

a1320ec1ea ▾

...

tensorflow / tensorflow / core / framework / shape_inference.h



jpienaar Document ShapeHandle & DimensionHandle ownership ... ✓

History

18 contributors



902 lines (761 sloc) | 36.6 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 #ifndef TENSORFLOW_CORE_FRAMEWORK_SHAPE_INFERENCE_H_
16 #define TENSORFLOW_CORE_FRAMEWORK_SHAPE_INFERENCE_H_
17
18 #include <vector>
19
20 #include "absl/memory/memory.h"
21 #include "tensorflow/core/framework/full_type.pb.h"
22 #include "tensorflow/core/framework/node_def_util.h"
23 #include "tensorflow/core/framework/tensor.h"
24 #include "tensorflow/core/lib/core/errors.h"
25 #include "tensorflow/core/lib/core/status.h"
26 #include "tensorflow/core/lib/gtl/inlined_vector.h"
27 #include "tensorflow/core/platform/macros.h"
28
29 namespace tensorflow {

```

```

30
31 namespace grappler {
32 class GraphProperties;
33 class SymbolicShapeManager;
34 } // namespace grappler
35
36 namespace shape_inference {
37
38 struct DimensionOrConstant;
39 class InferenceContext;
40
41 // This header contains the InferenceContext that is used to infer the shape of
42 // the results of an operation or flag an operation with invalid inputs (e.g.,
43 // mismatched shapes for elementwise operation) by ShapeRefiner. The shape of an
44 // operation is computed using the OpShapeInferenceFn set via SetShapeFn in op
45 // registration. The OpShapeInferenceFn uses a per op InferenceContext populated
46 // with input shapes to compute resultant shape (including resource shapes).
47 //
48 // The shapes created in the InferenceContext are bound to the lifetime of the
49 // InferenceContext in which it was created. E.g., in
50 //
51 // ```c++
52 // InferenceContext c;
53 // // Below a ShapeHandle is returned by MakeShape, while UnknownDim returns a
54 // // DimensionHandle.
55 // ShapeHandle in0 = c.MakeShape({10, c.UnknownDim()});
56 // ```
57 //
58 // the ShapeHandle `in0` (and the nested unknown dim inside) is only valid while
59 // `c` is in scope, as ShapeHandle and DimensionHandle are effectively
60 // wrappers around pointers stored inside the context with the lifetime of the
61 // value pointed to managed by the context. The result from one operation's
62 // inference context will be passed as input to the inference of consumer
63 // operations. Hence it is possible for ShapeHandles produced by inference on a
64 // node to consist of ShapeHandles owned by different InferenceContexts. While
65 // inferring the shapes of a Graph, the InferenceContext of all nodes/operations
66 // in the Graph remain resident for the lifetime of the Graph (e.g, there is a
67 // map from each node to its InferenceContext, technically its
68 // ExtendedInferencContext which additionally stores the element types of inputs
69 // & outputs, which remains resident).
70 //
71 // For functions, the body of the function is instantiated as a Graph while
72 // inferring the result shapes of a function call node. The rules above apply
73 // while the function's shape is being inferred, but the contexts associated
74 // with nodes in the function body are released once the function call's
75 // resultant shapes are inferred. The shapes of results returned by a function
76 // are propagated to the InferenceContext of the function call's op (which is
77 // associated with a Graph of nodes whose shape is being inferred) as the return
78 // values of a function call node are the inputs of its consumer, but the return

```

```

79 // values are produced by nodes inside the function whose InferenceContexts
80 // (which owns the values pointed to by ShapeHandle and DimensionHandle) are
81 // reclaimed after inferring function result shapes. Recursive user-defined
82 // function are not supported hence inference of functions are fully nested with
83 // the InferenceContext's of function calls forming a stack.
84 //
85 // For example, consider the following call and function:
86 //
87 // ```python
88 // @tf.function
89 // def g(st):
90 //     d = tf.add(st, st)
91 //     return d
92 //
93 // @tf.function
94 // def f():
95 //     st = tf.A()
96 //     result = g(st)
97 //     return h(result)
98 // ```
99 //
100 // During inference of f, the shape of `A` will be inferred and the results from
101 // its InferenceContext used as inputs to function call `g(st)`. The call node
102 // will have an InferenceContext created (call it outer context) and the graph
103 // corresponding to function `g` will be instantiated. The result shape of the
104 // Arg nodes of the function will be associated with input from outer context.
105 // During inference of `g` (for the callsite `g(st)` in `f`), the
106 // InferenceContext of all nodes inside `g` will remain alive. Thus, when shape
107 // of `tf.add` is computed it may rely on all inputs. Once the RetVal nodes of a
108 // function is reached, we know the shape of its input may correspond to a shape
109 // queried in the outer context and it is explicitly copied to outer context. In
110 // this case that means that the shape of `d` is copied to the InferenceContext
111 // of `g(st)` and so when `h(result)` is executed this shape may be queried.
112 // Furthermore, no shapes computed due to call `g(st)` can be queried post this
113 // point and, as the RetVal shapes have been copied into outer context, all
114 // InferenceContexts associated with nodes in function `g` instantiated for
115 // `g(st)` may be and are released.
116
117 // Dimension values are accessed through InferenceContext.
118 class Dimension {
119     private:
120         Dimension();
121         Dimension(int64_t value);
122         ~Dimension() {}
123
124     const int64_t value_;
125
126     friend class InferenceContext;
127     friend class ShapeManager;

```

```

128     TF_DISALLOW_COPY_AND_ASSIGN(Dimension);
129 };
130
131 class DimensionHandle {
132 public:
133     DimensionHandle() {}
134     bool SameHandle(DimensionHandle d) const { return ptr_ == d.ptr_; }
135     std::size_t Handle() const { return reinterpret_cast<std::size_t>(ptr_); }
136
137 private:
138     DimensionHandle(const Dimension* dim) { ptr_ = dim; }
139
140     const Dimension* operator->() const { return ptr_; }
141     bool IsSet() const { return ptr_ != nullptr; }
142
143     const Dimension* ptr_ = nullptr;
144
145     friend struct DimensionOrConstant;
146     friend class InferenceContext;
147     friend class ShapeInferenceTest;
148     friend class ShapeInferenceTestutil;
149     friend class ::tensorflow::grappler::GraphProperties;
150     friend class ::tensorflow::grappler::SymbolicShapeManager;
151
152     // Intentionally copyable.
153 };
154
155 // Shape rank and dimensions are accessed through InferenceContext.
156 class Shape {
157 private:
158     Shape();
159     Shape(const std::vector<DimensionHandle>& dims);
160     ~Shape() {}
161
162     const int32 rank_;
163     const std::vector<DimensionHandle> dims_;
164
165     friend class InferenceContext;
166     friend class ::tensorflow::grappler::SymbolicShapeManager;
167
168     TF_DISALLOW_COPY_AND_ASSIGN(Shape);
169 };
170
171 class ShapeHandle {
172 public:
173     ShapeHandle() {}
174     bool SameHandle(ShapeHandle s) const { return ptr_ == s.ptr_; }
175     std::size_t Handle() const { return reinterpret_cast<std::size_t>(ptr_); }
176

```

```

177 private:
178     ShapeHandle(const Shape* shape) { ptr_ = shape; }
179     const Shape* operator->() const { return ptr_; }
180     bool IsSet() const { return ptr_ != nullptr; }
181
182     const Shape* ptr_ = nullptr;
183
184     friend class InferenceContext;
185     friend class ShapeInferenceTest;
186     friend class ShapeInferenceTestutil;
187     friend class ::tensorflow::grappler::SymbolicShapeManager;
188
189     // Intentionally copyable.
190 };
191
192 // Struct used to allow functions to take DimensionHandle or a dimension value.
193 // Not meant to be constructed directly.
194 struct DimensionOrConstant {
195     public:
196         // Intentionally not explicit.
197         DimensionOrConstant(DimensionHandle dim);
198
199         // val must be non-negative or InferenceContext::kUnknownDim.
200         DimensionOrConstant(int64_t val);
201
202         // dim takes precedence. If dim != nullptr, val is ignored.
203         DimensionHandle dim;
204         int64_t val;
205
206     private:
207         DimensionOrConstant();
208 };
209
210 struct ShapeAndType {
211     ShapeAndType() {}
212     ShapeAndType(ShapeHandle s, DataType t) : shape(s), dtype(t) {}
213     // TODO(mdan): Remove dtype from constructor, and use type_ instead.
214     // dtype is kept here for backward compatibility. Its information should
215     // be redundant to that in type;
216     ShapeAndType(ShapeHandle s, DataType t, FullTypeDef type_)
217         : shape(s), dtype(t), type(type_) {}
218
219     ShapeHandle shape;
220     DataType dtype = DT_INVALID;
221     FullTypeDef type;
222 };
223
224 // Shape inference functions registered on ops in REGISTER_OP implement
225 // their shape functions in terms of this InferenceContext. An InferenceContext

```

```

226 // is created by the framework and passed to a shape inference function. The
227 // shape inference function calls functions on the context, and should call
228 // set_output() to set the shape on all outputs.
229 //
230 // To infer shapes for user-defined functions see ShapeRefiner.
231 //
232 // All Shape* and Dimension* returned by functions of InferenceContext are owned
233 // by the InferenceContext.
234 class InferenceContext {
235 public:
236     static constexpr int64_t kUnknownDim = -1;
237     static constexpr int32_t kUnknownRank = -1;
238
239     // <input_tensors> is NULL-padded to be the same size as <input_shapes>.
240     //
241     // Elements of <input_tensors_as_shapes> are used for when a shape function
242     // makes a call to MakeShapeFromShapeTensor; in particular, when the
243     // input_tensors[i] is nullptr but the shape represented by it is partially
244     // known from analysis of the graph.
245     // <input_tensors_as_shapes> can have fewer elements than <input_shapes>.
246     // Values of <input_tensors_as_shapes> do not need to outlive the context.
247     InferenceContext(int graph_def_version, const AttrSlice& attrs,
248                     const OpDef& op_def,
249                     const std::vector<ShapeHandle>& input_shapes,
250                     const std::vector<const Tensor*>& input_tensors,
251                     const std::vector<ShapeHandle>& input_tensors_as_shapes,
252                     std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
253                     input_handle_shapes_and_types);
254
255     // <input_tensors> is NULL-padded to be the same size as <input_shapes>.
256     //
257     // Elements of <input_tensors_as_shapes> are used for when a shape
258     // function makes a call to MakeShapeFromShapeTensor; in particular, when
259     // the input_tensors[i] is nullptr but the shape represented by it is
260     // partially known from analysis of the graph. <input_tensors_as_shapes>
261     // can have fewer elements than <input_shapes>. Values of
262     // <input_tensors_as_shapes> do not need to outlive the context.
263     InferenceContext(
264         int graph_def_version, const AttrSlice& attrs, const OpDef& op_def,
265         const std::vector<PartialTensorShape>& input_shapes,
266         const std::vector<const Tensor*>& input_tensors,
267         const std::vector<PartialTensorShape>& input_tensors_as_shapes,
268         const std::vector<std::unique_ptr<
269             std::vector<std::pair<PartialTensorShape, DataType>>>>&
270             input_handle_shapes_and_types);
271
272     ~InferenceContext();
273
274     // Runs the shape inference function 'fn' with 'this' as the

```

```

275 // argument, returns the status of the inference.
276 //
277 // On error, additional context is provided in the error message.
278 Status Run(
279     const std::function<Status(shape_inference::InferenceContext* c)>& fn);
280
281 // Merge the stored shape of the input in position idx with <shape> according
282 // to the following rules:
283 //
284 // - If the ShapeHandles are the same or <shape> is unknown, there will be no
285 //   change. Otherwise if the stored shape is unknown, the new shape will be
286 //   <shape>.
287 // - If both shapes are known, then they must have the same rank.
288 // - For any one dimension, if the values for that dimension in both shapes
289 //   are known, then the values must match.
290 // - If one shape has equal or more information than the other shape in every
291 //   dimension, the new shape will become the shape with more information.
292 // - Example: merging [2,?] and [?,2] results in [2,2]
293 // - Example: [2,2] cannot be merged with [1,2]
294 //
295 // This requires idx to be in the [0, num_inputs) range. If the merge is
296 // successful, return true. Return false otherwise.
297 bool MergeInput(int idx, ShapeHandle shape) {
298     ShapeHandle new_shape;
299     if (!Merge(inputs_[idx], shape, &new_shape).ok()) return false;
300     inputs_[idx] = new_shape;
301     return true;
302 }
303
304 // Relax the stored shape of the input in position idx with <shape> according
305 // to the following rules:
306 //
307 // - If the ShapeHandles are the same then the stored shape will be returned.
308 // - If either of the ShapeHandles are unknown, then a new UnknownShape will
309 //   be returned. A new shape must be returned because we cannot claim that
310 //   the resulting shape is necessarily the same as either of the input
311 //   shapes.
312 // - If the shapes both have known ranks but their ranks are different, a new
313 //   UnknownShape will be returned.
314 // - For any one dimension, if the value for that dimension in either of the
315 //   shapes is unknown, a new shape will be returned with a new UnknownDim in
316 //   that dimension.
317 // - For any one dimension, if the values for that dimension in both shapes
318 //   are known but do not match, a new shape will be returned with a new
319 //   UnknownDim in that dimension.
320 // - If both shapes have the same known rank and match in every dimension,
321 //   the stored shape will be returned.
322 // - Example: relaxing [2,?] and [?,2] results in [?,?]
323 // - Example: relaxing [2,2] and [3,2] results in [?,2]

```

```

324 // - Example: relaxing [2,2] with [1,2,3] results in ?
325 //
326 // This requires idx to be in the [0, num_inputs) range. If the relax is
327 // successful and the new shape differs from the old one, store the new
328 // shape and return true. Return false otherwise.
329 bool RelaxInput(int idx, ShapeHandle shape) {
330     ShapeHandle new_shape;
331     Relax(inputs_[idx], shape, &new_shape);
332     if (inputs_[idx].SameHandle(new_shape)) {
333         return false;
334     }
335     inputs_[idx] = new_shape;
336     return true;
337 }
338
339 void SetInput(int idx, ShapeHandle shape) { inputs_[idx] = shape; }
340
341 ShapeHandle input(int64_t idx) const { return inputs_[idx]; }
342 Status input(StringPiece input_name, std::vector<ShapeHandle>* output) const;
343 int num_inputs() const { return inputs_.size(); }
344
345 // Returns the input tensor at index <idx>, or nullptr if the input tensor is
346 // not available at the time of shape inference.
347 const Tensor* input_tensor(int idx) {
348     // Mark that this idx was requested.
349     request_input_tensor(idx);
350     return input_tensors_[idx];
351 }
352
353 // Notifies the shape refiner that the value of the tensor at index <idx>
354 // is needed. The shape refiner tries to statically compute this tensor,
355 // and if successful re-runs the shape function with this tensor available
356 // in the call to 'input_tensor(idx)'.
357 void request_input_tensor(int idx) { requested_input_tensor_[idx] = true; }
358
359 // Returns true iff input_tensor(idx) was called by the shape function.
360 bool requested_input_tensor(int idx) const {
361     return requested_input_tensor_[idx];
362 }
363
364 // Notifies the shape refiner that the value of the tensor at index <idx>
365 // as a partial shape is needed. The shape refiner tries to statically compute
366 // this, and if successful re-runs the shape function with the
367 // computed PartialTensorShape available in the call to
368 // 'MakeShapeFromShapeTensor(idx, handle)' or
369 // 'MakeShapeFromShapeTensorTreatScalarAsUnknownShape(idx, handle)'.
370 void request_input_tensor_as_partial_shape(int idx) {
371     requested_input_tensor_as_partial_shape_[idx] = true;
372 }

```



```

373
374 // Returns true if MakeShapeFromInputTensor was called but the constant
375 // input_tensor was not present.
376 bool requested_input_tensor_as_partial_shape(int idx) const {
377     return requested_input_tensor_as_partial_shape_[idx];
378 }
379
380 void set_input_tensors(const std::vector<const Tensor*>& input_tensors) {
381     input_tensors_ = input_tensors;
382 }
383
384 void set_input_tensors_as_shapes(
385     const std::vector<ShapeHandle>& input_tensors_as_shapes) {
386     input_tensors_as_shapes_ = input_tensors_as_shapes;
387 }
388
389 const std::vector<ShapeHandle>& input_tensors_as_shapes() const {
390     return input_tensors_as_shapes_;
391 }
392
393 ShapeHandle output(int64_t idx) const { return outputs_.at(idx); }
394 void set_output(int idx, ShapeHandle shape) { outputs_.at(idx) = shape; }
395 Status set_output(StringPiece output_name,
396     const std::vector<ShapeHandle>& shapes);
397
398 int num_outputs() const { return outputs_.size(); }
399 ShapeHandle output(int idx) const { return outputs_.at(idx); }
400 Status output(StringPiece output_name,
401     std::vector<ShapeHandle>* output) const;
402
403 // Returns the value for attribute named `attr_name`.
404 Status GetAttr(StringPiece attr_name, const AttrValue** attr_value) const {
405     return attrs_.Find(attr_name, attr_value);
406 }
407 const AttrValue* GetAttr(StringPiece attr_name) const {
408     return attrs_.Find(attr_name);
409 }
410
411 const FullTypeDef& ret_types() const { return ret_types_; }
412
413 // idx can be negative for an offset from end of dimensions.
414 // idx must be in the range [-1 * s.rank, s.rank).
415 DimensionHandle Dim(ShapeHandle s, int64_t idx) {
416     if (!s.Handle() || s->rank_ == kUnknownRank) {
417         return UnknownDim();
418     }
419     return DimKnownRank(s, idx);
420 }
421 // As above, but asserts that the rank of the shape is known.

```

```

422 static DimensionHandle DimKnownRank(ShapeHandle s, int64_t idx) {
423     CHECK_NE(s->rank_, kUnknownRank);
424     if (idx < 0) {
425         return s->dims_[s->dims_.size() + idx];
426     }
427     return s->dims_[idx];
428 }
429
430 static int32 Rank(ShapeHandle s) {
431     return s.IsSet() ? s->rank_ : kUnknownRank;
432 }
433 static bool RankKnown(ShapeHandle s) {
434     return (s.IsSet() && (Rank(s) != kUnknownRank));
435 }
436 static inline int64_t Value(DimensionOrConstant d) {
437     return d.dim.IsSet() ? d.dim->value_ : d.val;
438 }
439 static inline bool ValueKnown(DimensionOrConstant d) {
440     return Value(d) != kUnknownDim;
441 }
442
443 // Fills the output proto with the shape defined by the handle.
444 // "proto" is expected to be empty prior to the call.
445 void ShapeHandleToProto(ShapeHandle handle, TensorShapeProto* proto);
446
447 // Returns true if the rank and all dimensions of the Shape are known.
448 bool FullyDefined(ShapeHandle s);
449
450 // Returns the total number of elements, or an unknown dimension for an
451 // incomplete shape.
452 DimensionHandle NumElements(ShapeHandle s);
453
454 std::string DebugString(ShapeHandle s);
455 std::string DebugString(DimensionHandle d);
456 std::string DebugString(const ShapeAndType& shape_and_type);
457 std::string DebugString(gtl::ArraySlice<ShapeAndType> shape_and_types);
458
459 // Describes the whole context, for debugging purposes.
460 std::string DebugString() const;
461
462 // If <shape> has rank <rank>, or its rank is unknown, return OK and return
463 // the shape with asserted rank in <*out>. Otherwise return an error.
464 //
465 // Note that <*out> may be set to <shape>.
466 Status WithRank(ShapeHandle shape, int64_t rank,
467                 ShapeHandle* out) TF_MUST_USE_RESULT;
468 Status WithRankAtLeast(ShapeHandle shape, int64_t rank,
469                        ShapeHandle* out) TF_MUST_USE_RESULT;
470 Status WithRankAtMost(ShapeHandle shape, int64_t rank,

```

```

471         ShapeHandle* out) TF_MUST_USE_RESULT;
472
473 // If <dim> has value <value>, or its value is unknown, returns OK and returns
474 // the dimension with asserted value in <*out>. Otherwise returns an error.
475 //
476 // Note that <*out> may be set to <dim>.
477 Status WithValue(DimensionHandle dim, int64_t value,
478                 DimensionHandle* out) TF_MUST_USE_RESULT;
479
480 // Merges <s0> and <s1> and returns the merged shape in <*out>. See
481 // 'MergeInput' function for full details and examples.
482 Status Merge(ShapeHandle s0, ShapeHandle s1,
483             ShapeHandle* out) TF_MUST_USE_RESULT;
484
485 // Asserts that <s>'s rank >= <prefix>'s rank, and the first
486 // <prefix.rank> dimensions of <s> are compatible with the dimensions of
487 // <prefix>.
488 // Returns the merged results in <*s_out> and <*prefix_out>.
489 Status MergePrefix(ShapeHandle s, ShapeHandle prefix, ShapeHandle* s_out,
490                  ShapeHandle* prefix_out) TF_MUST_USE_RESULT;
491
492 // Merges <d0> and <d1> and returns the merged dimension in <*out>. If <d0>
493 // and <d1> have incompatible values, returns an error.
494 //
495 // Note that <*out> may be set to <d0> or <d1>.
496 Status Merge(DimensionHandle d0, DimensionHandle d1,
497             DimensionHandle* out) TF_MUST_USE_RESULT;
498
499 // Returns in <*out> a sub-shape of <s> with dimensions [start:].
500 // <start> can be negative to index from the end of the shape. If <start> >
501 // rank of <s>, then an empty subshape is returned.
502 Status Subshape(ShapeHandle s, int64_t start,
503               ShapeHandle* out) TF_MUST_USE_RESULT;
504
505 // Returns in <*out> a sub-shape of <s>, with dimensions [start:end].
506 // <start> and <end> can be negative, to index from the end of the shape.
507 // <start> and <end> are set to the rank of <s> if > rank of <s>.
508 Status Subshape(ShapeHandle s, int64_t start, int64_t end,
509               ShapeHandle* out) TF_MUST_USE_RESULT;
510
511 // Returns in <*out> a sub-shape of <s>, with dimensions [start:end:stride].
512 // <start> and <end> can be negative, to index from the end of the shape.
513 // <start> and <end> are set to the rank of <s> if > rank of <s>.
514 // <stride> can be negative, to reverse the <s>.
515 Status Subshape(ShapeHandle s, int64_t start, int64_t end, int64_t stride,
516               ShapeHandle* out) TF_MUST_USE_RESULT;
517
518 // Returns in <*out> the result of appending the dimensions of <s2> to those
519 // of <s1>.

```

```

520 Status Concatenate(ShapeHandle s1, ShapeHandle s2,
521                    ShapeHandle* out) TF_MUST_USE_RESULT;
522
523 // Returns in <out> the shape from replacing <s.dim[dim_index]> with
524 // <new_dim>.
525 Status ReplaceDim(ShapeHandle s, int64_t dim_index, DimensionHandle new_dim,
526                  ShapeHandle* out) TF_MUST_USE_RESULT;
527
528 // Returns a new shape with the given dims. The returned value is owned by
529 // this context.
530 ShapeHandle MakeShape(const std::vector<DimensionHandle>& dims);
531 ShapeHandle MakeShape(std::initializer_list<DimensionOrConstant> dims);
532
533 // Returns a new unknown shape.
534 ShapeHandle UnknownShape();
535
536 // Returns a shape with specified rank but unknown dims.
537 ShapeHandle UnknownShapeOfRank(int64_t rank);
538
539 // Returns a new shape of zero dimensions.
540 ShapeHandle Scalar();
541
542 // Returns a new shape of one dimension.
543 ShapeHandle Vector(DimensionOrConstant dim);
544
545 // Returns a new shape of two dimensions.
546 ShapeHandle Matrix(DimensionOrConstant dim1, DimensionOrConstant dim2);
547
548 // Returns in <out> a new shape whose dimension sizes come from input tensor
549 // <input_idx>. The tensor must be a 1-dimensional int32 or int64 tensor. If
550 // the input tensor is NULL, then an unknown shape is returned.
551 Status MakeShapeFromShapeTensor(int input_idx, ShapeHandle* out);
552
553 // Like the function above, but treats scalar values as unknown
554 // shapes. **NOTE** If the scalar is statically known, its value
555 // must be -1 or an error is returned.
556 Status MakeShapeFromShapeTensorTreatScalarAsUnknownShape(int input_idx,
557                                                          ShapeHandle* out);
558
559 // Returns in <out> a new shape corresponding to <proto>.
560 Status MakeShapeFromShapeProto(const TensorShapeProto& proto,
561                               ShapeHandle* out);
562
563 // Returns in <out> a new shape corresponding to <partial_shape>.
564 Status MakeShapeFromPartialTensorShape(
565     const PartialTensorShape& partial_shape, ShapeHandle* out);
566
567 // Returns in <out> a new shape corresponding to <shape>.
568 Status MakeShapeFromTensorShape(const TensorShape& shape, ShapeHandle* out);

```

```

569
570 // Returns a new dimension of the given size. The returned value is owned by
571 // this context.
572 inline DimensionHandle MakeDim(DimensionOrConstant d) {
573     return shape_manager_.MakeDim(d);
574 }
575
576 inline DimensionHandle UnknownDim() { return MakeDim(kUnknownDim); }
577
578 // Returns in <val> a scalar value from an input tensor <t>. The input tensor
579 // must be a 0-dimensional int32 or int64 tensor. Caller must ensure that the
580 // input tensor is not NULL.
581 Status GetScalarFromTensor(const Tensor* t, int64_t* val);
582
583 // Returns in <val> a scalar value from a 1D input tensor <t> with int32 or
584 // int64 elements. Caller must ensure that the input tensor is not NULL.
585 Status GetScalarFromTensor(const Tensor* t, int64_t idx, int64_t* val);
586
587 // Returns a new dimension whose value is given by a scalar input tensor.
588 // The input tensor must be in host memory, since it is dereferenced to get
589 // the value.
590 Status MakeDimForScalarInput(int idx, DimensionHandle* out);
591
592 // Returns a new dimension whose value is given by a scalar input tensor.
593 // This allows for a negative input dimension given the rank of a separate
594 // tensor. This rank can be negative if unknown.
595 // The input tensor must be in host memory, since it is dereferenced to get
596 // the value.
597 Status MakeDimForScalarInputWithNegativeIndexing(int idx, int input_rank,
598                                                  DimensionHandle* out);
599
600 // Look up the attr being evaluated with name attr_name and set *value to its
601 // value. If no attr with attr_name is found in def(), or the attr does not
602 // have a matching type, a non-ok status will be returned.
603 template <class T>
604 Status GetAttr(StringPiece attr_name, T* value) const;
605
606 // Returns in <out> the result of dividing <dividend> by <divisor>.
607 // Returns an error if <divisor> is not positive or if <evenly_divisible>
608 // and <divisor> does not evenly divide <dividend>.
609 Status Divide(DimensionHandle dividend, DimensionOrConstant divisor,
610              bool evenly_divisible, DimensionHandle* out);
611
612 // Returns in <out> the sum of <first> and <second>.
613 Status Add(DimensionHandle first, DimensionOrConstant second,
614            DimensionHandle* out);
615
616 // Returns in <out> the dimension that is <first> minus <second>.
617 Status Subtract(DimensionHandle first, DimensionOrConstant second,

```

```

618         DimensionHandle* out);
619
620 // Returns in <out> the product of <first> and <second>.
621 Status Multiply(DimensionHandle first, DimensionOrConstant second,
622               DimensionHandle* out);
623
624 // Returns in <out> the minimum of <first> and <second>. If either <first> or
625 // <second> is zero the results is zero. Otherwise, if either <first> or
626 // <second> is unknown the results is unknown.
627 Status Min(DimensionHandle first, DimensionOrConstant second,
628           DimensionHandle* out);
629
630 // Returns in <out> the maximum of <first> and <second>. If either <first> or
631 // <second> is unknown the results is unknown.
632 Status Max(DimensionHandle first, DimensionOrConstant second,
633           DimensionHandle* out);
634
635 Status construction_status() const { return construction_status_; }
636
637 // Methods to propagate shape and dtype on edges of handles. Handles are the
638 // dtype DT_RESOURCE which can be used to access state stored in a
639 // ResourceManager. When ops (such as variables) consume these handles to
640 // produce tensors they might need to know side-information about the shapes
641 // and dtypes of tensors which can be accessed via the handle. These methods
642 // propagate that information. Output handle dtypes and shapes are ignored if
643 // the output tensor is not of type DT_RESOURCE.
644
645 // Merge the stored shapes and types corresponding to the input handle in
646 // position idx with the specified shapes and types. This requires idx to be
647 // in the [0, num_inputs) range.
648 //
649 // If the merge is successful and any of the new shapes differs from the old
650 // one, or any of the old dtypes was DT_INVALID, store the new shapes and
651 // return true. Return false otherwise.
652 //
653 // See 'MergeInput' function for full details and examples.
654 bool MergeInputHandleShapesAndTypes(
655     int idx,
656     const std::vector<ShapeAndType>& shapes_and_types) TF_MUST_USE_RESULT;
657
658 // As MergeInputHandleShapesAndTypes, but for an output.
659 bool MergeOutputHandleShapesAndTypes(
660     int idx,
661     const std::vector<ShapeAndType>& shapes_and_types) TF_MUST_USE_RESULT;
662
663 // Relaxes the stored shapes and types corresponding to the input handle in
664 // position idx with the specified shapes and types. This requires idx to be
665 // in the [0, num_inputs) range.
666 //

```

```

667 // If the relax is successful (sizes are the same, old dtypes match new ones
668 // or are DT_INVALID), then store the relaxed shapes and return true.
669 // Return false otherwise.
670 //
671 // See 'RelaxInput' function for full details and examples.
672 bool RelaxInputHandleShapesAndMergeTypes(
673     int idx,
674     const std::vector<ShapeAndType>& shapes_and_types) TF_MUST_USE_RESULT;
675
676 // As RelaxInputHandleShapesAndTypes, but for an output.
677 bool RelaxOutputHandleShapesAndMergeTypes(
678     int idx,
679     const std::vector<ShapeAndType>& shapes_and_types) TF_MUST_USE_RESULT;
680
681 void set_input_handle_shapes_and_types(
682     int idx, const std::vector<ShapeAndType>& shapes_and_types) {
683     input_handle_shapes_and_types_[idx] =
684         absl::make_unique<std::vector<ShapeAndType>>(shapes_and_types);
685 }
686
687 // Returns the output handle shapes and types, for the resource tensor output
688 // at index <idx>. Returns NULL if the shape and types were never set.
689 const std::vector<ShapeAndType>* output_handle_shapes_and_types(int idx) {
690     return output_handle_shapes_and_types_[idx].get();
691 }
692
693 // Returns the inputs handle shapes and types, for the resource tensor output
694 // at index <idx>. Returns NULL if the shape and types were not available.
695 const std::vector<ShapeAndType>* input_handle_shapes_and_types(int idx) {
696     return input_handle_shapes_and_types_[idx].get();
697 }
698
699 void set_output_handle_shapes_and_types(
700     int idx, const std::vector<ShapeAndType>& shapes_and_types) {
701     output_handle_shapes_and_types_[idx].reset(
702         new std::vector<ShapeAndType>(shapes_and_types));
703 }
704
705 // Note that shape functions should usually call MakeShapeFromShapeTensor,
706 // as it does more analysis to provide partial shapes.
707 //
708 // Returns in <out> a new shape whose dimension sizes come from tensor <t>.
709 // The tensor must be a 1-dimensional int32 or int64 tensor. If <t> is NULL,
710 // then an unknown shape is returned.
711 Status MakeShapeFromTensor(const Tensor* t, ShapeHandle tensor_shape,
712                             ShapeHandle* out);
713
714 int graph_def_version() const { return graph_def_version_; }
715

```

```

716     const std::vector<std::pair<ShapeHandle, ShapeHandle>>& MergedShapes() const {
717         return merged_shapes_;
718     }
719     const std::vector<std::pair<DimensionHandle, DimensionHandle>>& MergedDims()
720         const {
721         return merged_dims_;
722     }
723
724     // Adds new outputs; useful when mutating the graph.
725     Status ExpandOutputs(int new_output_size);
726
727 private:
728     // Creates and stores shapes for use in InferenceContext.
729     class ShapeManager {
730     public:
731         ShapeManager();
732         ~ShapeManager();
733
734         // Returns a new shape with the given dims. The returned value is owned by
735         // this class.
736         ShapeHandle MakeShape(const std::vector<DimensionHandle>& dims);
737
738         // Returns a new unknown shape.
739         ShapeHandle UnknownShape();
740
741         // Returns a new dimension of the given size. The returned value
742         // is owned by this class.
743         inline DimensionHandle MakeDim(DimensionOrConstant d) {
744             if (d.dim.IsSet()) {
745                 return d.dim;
746             } else {
747                 all_dims_.push_back(new Dimension(d.val));
748                 return all_dims_.back();
749             }
750         }
751
752     private:
753         std::vector<Shape*> all_shapes_;    // values are owned.
754         std::vector<Dimension*> all_dims_;  // values are owned.
755     };
756
757     friend class ::tensorflow::grappler::GraphProperties;
758
759     friend class ShapeInferenceTest;        // For testing Relax functions.
760     friend class ShapeInferenceTestutil;    // For testing shapes.
761
762     // Shared initialization across the two constructors. Remove
763     // once we get rid of one of them.
764     void PreInputInit(const OpDef& op_def,

```



```

765         const std::vector<const Tensor*>& input_tensors,
766         const std::vector<ShapeHandle>& input_tensors_as_shapes);
767 void PostInputInit(std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
768         input_handle_data);
769
770 Status ReturnUnknownShape(ShapeHandle* out) {
771     *out = UnknownShape();
772     return Status::OK();
773 }
774 Status ReturnCreatedShape(const std::vector<DimensionHandle>& dims,
775         ShapeHandle* out) {
776     *out = MakeShape(dims);
777     return Status::OK();
778 }
779
780 // Adds additional context to the given status.
781 Status AttachContext(const Status& status);
782
783 // Relaxes an existing value <d_old> with a new value <d_new> and returns the
784 // relaxed dimension in <*out>. If <d_old> and <d_new> have incompatible
785 // values, returns an error.
786 //
787 // Note that <*out> may be set to <d_old> or <d_new>.
788 void Relax(DimensionHandle d_old, DimensionHandle d_new,
789         DimensionHandle* out);
790 // Relaxes an existing shape <s_old> with a new shape <s_new> and returns the
791 // relaxed shape in <*out>. See 'RelaxInput' function for full details and
792 // examples.
793 void Relax(ShapeHandle s_old, ShapeHandle s_new, ShapeHandle* out);
794
795 // Used to implement MergeInputHandleShapesAndTypes and
796 // MergeOutputHandleShapesAndTypes.
797 bool MergeHandleShapesAndTypes(
798     const std::vector<ShapeAndType>& shapes_and_types,
799     std::vector<ShapeAndType>* to_update) TF_MUST_USE_RESULT;
800 // Used to implement RelaxInputHandleShapesAndMergeTypes and
801 // RelaxOutputHandleShapesAndMergeTypes.
802 bool RelaxHandleShapesAndMergeTypes(
803     const std::vector<ShapeAndType>& shapes_and_types,
804     std::vector<ShapeAndType>* to_update) TF_MUST_USE_RESULT;
805
806 // Forget all the previous merged shapes and dims.
807 void ForgetMerges() {
808     merged_shapes_.clear();
809     merged_dims_.clear();
810 }
811
812 // Helper method for MakeShapeFromTensor and MakeShapeFromShapeTensor.
813 Status InternalMakeShapeFromTensor(

```

```

814     bool treat_unknown_scalar_tensor_as_unknown_shape, const Tensor* t,
815     ShapeHandle tensor_shape, ShapeHandle* out);
816
817 ShapeManager shape_manager_;
818
819 // inputs_, outputs_, and input_tensors_as_shapes_ refer to values from
820 // `shape_manager`.
821 std::vector<ShapeHandle> inputs_;
822 std::vector<const Tensor*> input_tensors_;
823 std::vector<bool> requested_input_tensor_;
824 std::vector<ShapeHandle> outputs_;
825 // Can have fewer elements than inputs_.
826 std::vector<ShapeHandle> input_tensors_as_shapes_;
827 std::vector<bool> requested_input_tensor_as_partial_shape_;
828
829 // input_handle_shapes_and_types_[i] is the list of shape/type pairs available
830 // through the resource handle passed along input i of the node.
831 //
832 // Values may be NULL.
833 std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
834     input_handle_shapes_and_types_;
835
836 // output_handle_shapes_and_types_[i] is the list of shape/type pairs
837 // available through the resource handle passed along output i of the node.
838 //
839 // Values may be NULL.
840 std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
841     output_handle_shapes_and_types_;
842
843 // Return types for the node this context is associated with. This information
844 // is to eventually consolidate all the dtype and shape info, allowing for
845 // output_handle_shapes_and_types_ to be removed.
846 FullTypeDef ret_types_;
847
848 const int graph_def_version_;
849 AttrSlice attrs_;
850 NameRangeMap input_name_map_;
851 NameRangeMap output_name_map_;
852
853 // An error set during construction. TODO(cwhipkey): remove when test
854 // constructor is removed.
855 Status construction_status_;
856
857 // Pair of shape or dim handles that are equivalent, ie that represent the
858 // same underlying shape of dimension. Note that for each pair at least one of
859 // the handles must contain an unknown shape, since we don't keep track of
860 // known shapes or dims here.
861 std::vector<std::pair<ShapeHandle, ShapeHandle>> merged_shapes_;
862 std::vector<std::pair<DimensionHandle, DimensionHandle>> merged_dims_;

```

```

863
864     TF_DISALLOW_COPY_AND_ASSIGN(InferenceContext);
865 };
866
867 // -----
868 // Template and inline method implementations, please ignore
869
870 inline Dimension::Dimension() : value_(InferenceContext::kUnknownDim) {}
871 inline Dimension::Dimension(int64_t value) : value_(value) {
872     DCHECK(value >= 0 || value == InferenceContext::kUnknownDim)
873         << "Dimension must be non-negative or equal to "
874         << "InferenceContext::kUnknownDim but got "
875         << value;
876 }
877
878 inline Shape::Shape() : rank_(InferenceContext::kUnknownRank) {}
879 inline Shape::Shape(const std::vector<DimensionHandle>& dims)
880     : rank_(dims.size()), dims_(dims) {}
881
882 inline DimensionOrConstant::DimensionOrConstant(DimensionHandle dim)
883     : dim(dim) {
884     DCHECK(dim.IsSet()) << "Internal error: Got nullptr for Dimension.";
885 }
886
887 inline DimensionOrConstant::DimensionOrConstant(int64_t val) : val(val) {
888     DCHECK(val >= 0 || val == InferenceContext::kUnknownDim)
889         << "Dimension must be non-negative or equal to "
890         << "InferenceContext::kUnknownDim but got "
891         << val;
892 }
893
894 template <class T>
895 Status InferenceContext::GetAttr(StringPiece attr_name, T* value) const {
896     return GetNodeAttr(attrs_, attr_name, value);
897 }
898
899 } // namespace shape_inference
900 } // namespace tensorflow
901
902 #endif // TENSORFLOW_CORE_FRAMEWORK_SHAPE_INFERENCE_H_

```