jpienaar Rename to underlying type rather than alias ... ✓ History

16 contributors +4

709 lines (636 sloc) | 24.7 KB ...

```
1  /* Copyright 2015 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
16 #include "tensorflow/core/framework/attr_value_util.h"
17
18 #include <string>
19 #include <unordered_map>
20 #include <vector>
21
22 #include "absl/strings/escaping.h"
23 #include "tensorflow/core/framework/attr_value.pb_text.h"
24 #include "tensorflow/core/framework/tensor.pb_text.h"
25 #include "tensorflow/core/framework/tensor_shape.pb.h"
26 #include "tensorflow/core/framework/types.h"
27 #include "tensorflow/core/framework/types.pb_text.h"
28 #include "tensorflow/core/lib/core/errors.h"
29 #include "tensorflow/core/lib/core/stringpiece.h"
```

```cpp
#include "tensorflow/core/lib/hash/hash.h"
#include "tensorflow/core/lib/strings/proto_serialization.h"
#include "tensorflow/core/lib/strings/str_util.h"
#include "tensorflow/core/platform/protobuf.h"

namespace tensorflow {
namespace {

// Do not construct large tensors to compute their hash or compare for equality.
constexpr int kMaxAttrValueTensorByteSize = 32 * 1024 * 1024;  // 32mb

// Limit nesting of tensors to 100 deep to prevent memory overflow.
constexpr int kMaxTensorNestDepth = 100;

// Return the size of the tensor represented by this TensorProto. If shape is
// not fully defined return -1.
int64_t TensorByteSize(const TensorProto& t) {
  // num_elements returns -1 if shape is not fully defined.
  int64_t num_elems = TensorShape(t.tensor_shape()).num_elements();
  return num_elems < 0 ? -1 : num_elems * DataTypeSize(t.dtype());
}

// Compute TensorProto hash by creating a Tensor, serializing it as tensor
// content, and computing a hash of it's string representation. This is unsafe
// operation, because large tensors can be represented as TensorProto, but can't
// be serialized to tensor content.
uint64 TensorProtoHash(const TensorProto& tp) {
  Tensor tensor(tp.dtype());
  bool success = tensor.FromProto(tp);
  DCHECK(success);
  TensorProto p;
  tensor.AsProtoTensorContent(&p);
  return DeterministicProtoHash64(p);
}

// Do not create large tensors in memory, compute hash based on TensorProto
// string representation. Tensors with identical content potentially can have a
// different hash code if they are defined with different TensorProto
// representations.
uint64 FastTensorProtoHash(const TensorProto& tp) {
  if (TensorByteSize(tp) > kMaxAttrValueTensorByteSize) {
    return DeterministicProtoHash64(tp);
  } else {
    return TensorProtoHash(tp);
  }
}

bool AreTensorProtosEqual(const TensorProto& lhs, const TensorProto& rhs,
                          bool allow_false_negatives) {
```

```
  // A small TensorProto can expand into a giant Tensor.  So we avoid
  // conversion to an actual Tensor if we can quickly rule out equality
  // by comparing the Tensor size since different sized Tensors are definitely
  // different.
  const int64_t lhs_tensor_bytes = TensorByteSize(lhs);
  const int64_t rhs_tensor_bytes = TensorByteSize(rhs);
  if (lhs_tensor_bytes != rhs_tensor_bytes) {
    return false;
  }

  // If the TensorProto representation expands into a much bigger Tensor,
  // we have a fast-path that first compares the protos.
  const int64_t lhs_proto_bytes = lhs.ByteSizeLong();
  const bool large_expansion =
      (lhs_proto_bytes < 512 && lhs_tensor_bytes > 4096);

  // If the tensor is very large, we'll only compare the proto representation if
  // false negatives are allowed. This may miss some equivalent tensors whose
  // actual tensor values are the same but which are described by different
  // TensorProtos. This avoids construction of large protos in memory.
  const bool only_compare_proto =
      (allow_false_negatives && lhs_tensor_bytes > kMaxAttrValueTensorByteSize);
  if (large_expansion || only_compare_proto) {
    if (AreSerializedProtosEqual(lhs, rhs))
      return true;
    else if (only_compare_proto)
      return false;
  }

  // Finally, compare them by constructing Tensors and serializing them back.
  // There are multiple equivalent representations of attr values containing
  // TensorProtos. Comparing Tensor objects is pretty tricky. This is unsafe
  // operation, because large tensors can be represented as TensorProto, but
  // can't be serialized to tensor content.
  Tensor lhs_t(lhs.dtype());
  bool success = lhs_t.FromProto(lhs);
  if (!success) {
    return false;
  }

  Tensor rhs_t(rhs.dtype());
  success = rhs_t.FromProto(rhs);
  if (!success) {
    return false;
  }

  TensorProto lhs_tp;
  lhs_t.AsProtoTensorContent(&lhs_tp);

```

```
  TensorProto rhs_tp;
  rhs_t.AsProtoTensorContent(&rhs_tp);

  return AreSerializedProtosEqual(lhs_tp, rhs_tp);
}

using TensorProtoHasher = std::function<uint64(const TensorProto&)>;

uint64 AttrValueHash(const AttrValue& a, const TensorProtoHasher& tensor_hash) {
  if (a.has_tensor()) return tensor_hash(a.tensor());

  if (a.has_func()) {
    const NameAttrList& func = a.func();
    uint64 h = Hash64(func.name());
    std::map<string, AttrValue> map(func.attr().begin(), func.attr().end());
    for (const auto& pair : map) {
      h = Hash64(pair.first.data(), pair.first.size(), h);
      h = Hash64Combine(AttrValueHash(pair.second, tensor_hash), h);
    }
    return h;
  }

  // If `a` is not a tensor or func, get a hash of serialized string.
  return DeterministicProtoHash64(a);
}

string SummarizeString(const string& str) {
  string escaped = absl::CEscape(str);

  // If the string is long, replace the middle with ellipses.
  constexpr int kMaxStringSummarySize = 80;
  if (escaped.size() >= kMaxStringSummarySize) {
    StringPiece prefix(escaped);
    StringPiece suffix = prefix;
    prefix.remove_suffix(escaped.size() - 10);
    suffix.remove_prefix(escaped.size() - 10);
    return strings::StrCat("\"", prefix, "...", suffix, "\"");
  } else {
    return strings::StrCat("\"", escaped, "\"");
  }
}

string SummarizeTensor(const TensorProto& tensor_proto) {
  Tensor t;
  if (!t.FromProto(tensor_proto)) {
    return strings::StrCat(
        "<Invalid TensorProto: ", tensor_proto.ShortDebugString(), ">");
  }
  return t.DebugString();
```

```
}

string SummarizeFunc(const NameAttrList& func) {
  std::vector<string> entries;
  for (const auto& p : func.attr()) {
    entries.push_back(
        strings::StrCat(p.first, "=", SummarizeAttrValue(p.second)));
  }
  std::sort(entries.begin(), entries.end());
  return strings::StrCat(func.name(), "[", absl::StrJoin(entries, ", "), "]");
}

bool ParseAttrValueHelper_TensorNestsUnderLimit(int limit, string to_parse) {
  int nests = 0;
  int maxed_out = to_parse.length();
  int open_curly = to_parse.find('{');
  int open_bracket = to_parse.find('<');
  int close_curly = to_parse.find('}');
  int close_bracket = to_parse.find('>');
  if (open_curly == -1) {
    open_curly = maxed_out;
  }
  if (open_bracket == -1) {
    open_bracket = maxed_out;
  }
  int min = std::min(open_curly, open_bracket);
  do {
    if (open_curly == maxed_out && open_bracket == maxed_out) {
      return true;
    }
    if (min == open_curly) {
      nests += 1;
      open_curly = to_parse.find('{', open_curly + 1);
      if (open_curly == -1) {
        open_curly = maxed_out;
      }
    } else if (min == open_bracket) {
      nests += 1;
      open_bracket = to_parse.find('<', open_bracket + 1);
      if (open_bracket == -1) {
        open_bracket = maxed_out;
      }
    } else if (min == close_curly) {
      nests -= 1;
      close_curly = to_parse.find('}', close_curly + 1);
      if (close_curly == -1) {
        close_curly = maxed_out;
      }
    } else if (min == close_bracket) {
```

```
      nests -= 1;
      close_bracket = to_parse.find('>', close_bracket + 1);
      if (close_bracket == -1) {
        close_bracket = maxed_out;
      }
    }
    min = std::min({open_curly, open_bracket, close_curly, close_bracket});
  } while (nests < 100);
  return false;
}

}  // namespace

string SummarizeAttrValue(const AttrValue& attr_value) {
  switch (attr_value.value_case()) {
    case AttrValue::kS:
      return SummarizeString(attr_value.s());
    case AttrValue::kI:
      return strings::StrCat(attr_value.i());
    case AttrValue::kF:
      return strings::StrCat(attr_value.f());
    case AttrValue::kB:
      return attr_value.b() ? "true" : "false";
    case AttrValue::kType:
      return EnumName_DataType(attr_value.type());
    case AttrValue::kShape:
      return PartialTensorShape::DebugString(attr_value.shape());
    case AttrValue::kTensor:
      return SummarizeTensor(attr_value.tensor());
    case AttrValue::kList: {
      std::vector<string> pieces;
      if (attr_value.list().s_size() > 0) {
        for (int i = 0; i < attr_value.list().s_size(); ++i) {
          pieces.push_back(SummarizeString(attr_value.list().s(i)));
        }
      } else if (attr_value.list().i_size() > 0) {
        for (int i = 0; i < attr_value.list().i_size(); ++i) {
          pieces.push_back(strings::StrCat(attr_value.list().i(i)));
        }
      } else if (attr_value.list().f_size() > 0) {
        for (int i = 0; i < attr_value.list().f_size(); ++i) {
          pieces.push_back(strings::StrCat(attr_value.list().f(i)));
        }
      } else if (attr_value.list().b_size() > 0) {
        for (int i = 0; i < attr_value.list().b_size(); ++i) {
          pieces.push_back(attr_value.list().b(i) ? "true" : "false");
        }
      } else if (attr_value.list().type_size() > 0) {
        for (int i = 0; i < attr_value.list().type_size(); ++i) {
```

```
          pieces.push_back(EnumName_DataType(attr_value.list().type(i)));
        }
      } else if (attr_value.list().shape_size() > 0) {
        for (int i = 0; i < attr_value.list().shape_size(); ++i) {
          pieces.push_back(
              TensorShape::DebugString(attr_value.list().shape(i)));
        }
      } else if (attr_value.list().tensor_size() > 0) {
        for (int i = 0; i < attr_value.list().tensor_size(); ++i) {
          pieces.push_back(SummarizeTensor(attr_value.list().tensor(i)));
        }
      } else if (attr_value.list().func_size() > 0) {
        for (int i = 0; i < attr_value.list().func_size(); ++i) {
          pieces.push_back(SummarizeFunc(attr_value.list().func(i)));
        }
      }
      constexpr int kMaxListSummarySize = 50;
      if (pieces.size() >= kMaxListSummarySize) {
        pieces.erase(pieces.begin() + 5, pieces.begin() + (pieces.size() - 6));
        pieces[5] = "...";
      }
      return strings::StrCat("[", absl::StrJoin(pieces, ", "), "]");
    }
    case AttrValue::kFunc: {
      return SummarizeFunc(attr_value.func());
    }
    case AttrValue::kPlaceholder:
      return strings::StrCat("$", attr_value.placeholder());
    case AttrValue::VALUE_NOT_SET:
      return "<Unknown AttrValue type>";
  }
  return "<Unknown AttrValue type>";  // Prevent missing return warning
}

Status AttrValueHasType(const AttrValue& attr_value, StringPiece type) {
  int num_set = 0;

#define VALIDATE_FIELD(name, type_string, oneof_case)                         \
  do {                                                                        \
    if (attr_value.has_list()) {                                              \
      if (attr_value.list().name##_size() > 0) {                              \
        if (type != "list(" type_string ")") {                                \
          return errors::InvalidArgument(                                     \
              "AttrValue had value with type 'list(" type_string ")' when '", \
              type, "' expected");                                            \
        }                                                                     \
        ++num_set;                                                            \
      }                                                                       \
    } else if (attr_value.value_case() == AttrValue::oneof_case) {            \
```

```
324       if (type != type_string) {                                          \
325         return errors::InvalidArgument(                                   \
326             "AttrValue had value with type '" type_string "' when '", type,   \
327             "' expected");                                                \
328       }                                                                   \
329       ++num_set;                                                          \
330     }                                                                     \
331   } while (false)
332
333   VALIDATE_FIELD(s, "string", kS);
334   VALIDATE_FIELD(i, "int", kI);
335   VALIDATE_FIELD(f, "float", kF);
336   VALIDATE_FIELD(b, "bool", kB);
337   VALIDATE_FIELD(type, "type", kType);
338   VALIDATE_FIELD(shape, "shape", kShape);
339   VALIDATE_FIELD(tensor, "tensor", kTensor);
340   VALIDATE_FIELD(func, "func", kFunc);
341
342 #undef VALIDATE_FIELD
343
344   if (attr_value.value_case() == AttrValue::kPlaceholder) {
345     return errors::InvalidArgument(
346         "AttrValue had value with unexpected type 'placeholder'");
347   }
348
349   // If the attr type is 'list', we expect attr_value.has_list() to be
350   // true.  However, proto3's attr_value.has_list() can be false when
351   // set to an empty list for GraphDef versions <= 4. So we simply
352   // check if has_list is false and some other field in attr_value is
353   // set to flag the error.  This test can be made more strict once
354   // support for GraphDef versions <= 4 is dropped.
355   if (absl::StartsWith(type, "list(") && !attr_value.has_list()) {
356     if (num_set) {
357       return errors::InvalidArgument(
358           "AttrValue missing value with expected type '", type, "'");
359     } else {
360       // Indicate that we have a list, but an empty one.
361       ++num_set;
362     }
363   }
364
365   // Okay to have an empty list, but not to be missing a non-list value.
366   if (num_set == 0 && !absl::StartsWith(type, "list(")) {
367     return errors::InvalidArgument(
368         "AttrValue missing value with expected type '", type, "'");
369   }
370
371   // Ref types and DT_INVALID are illegal, and DataTypes must
372   // be a valid enum type.
```

```
373   if (type == "type") {
374     if (!DataType_IsValid(attr_value.type())) {
375       return errors::InvalidArgument("AttrValue has invalid DataType enum: ",
376                                      attr_value.type());
377     }
378     if (IsRefType(attr_value.type())) {
379       return errors::InvalidArgument(
380           "AttrValue must not have reference type value of ",
381           DataTypeString(attr_value.type()));
382     }
383     if (attr_value.type() == DT_INVALID) {
384       return errors::InvalidArgument("AttrValue has invalid DataType");
385     }
386   } else if (type == "list(type)") {
387     for (auto as_int : attr_value.list().type()) {
388       const DataType dtype = static_cast<DataType>(as_int);
389       if (!DataType_IsValid(dtype)) {
390         return errors::InvalidArgument("AttrValue has invalid DataType enum: ",
391                                        as_int);
392       }
393       if (IsRefType(dtype)) {
394         return errors::InvalidArgument(
395             "AttrValue must not have reference type value of ",
396             DataTypeString(dtype));
397       }
398       if (dtype == DT_INVALID) {
399         return errors::InvalidArgument("AttrValue contains invalid DataType");
400       }
401     }
402   }
403
404   return Status::OK();
405 }
406
407 bool ParseAttrValue(StringPiece type, StringPiece text, AttrValue* out) {
408   // Parse type.
409   string field_name;
410   bool is_list = absl::ConsumePrefix(&type, "list(");
411   if (absl::ConsumePrefix(&type, "string")) {
412     field_name = "s";
413   } else if (absl::ConsumePrefix(&type, "int")) {
414     field_name = "i";
415   } else if (absl::ConsumePrefix(&type, "float")) {
416     field_name = "f";
417   } else if (absl::ConsumePrefix(&type, "bool")) {
418     field_name = "b";
419   } else if (absl::ConsumePrefix(&type, "type")) {
420     field_name = "type";
421   } else if (absl::ConsumePrefix(&type, "shape")) {
```

```
422       field_name = "shape";
423     } else if (absl::ConsumePrefix(&type, "tensor")) {
424       field_name = "tensor";
425     } else if (absl::ConsumePrefix(&type, "func")) {
426       field_name = "func";
427     } else if (absl::ConsumePrefix(&type, "placeholder")) {
428       field_name = "placeholder";
429     } else {
430       return false;
431     }
432     if (is_list && !absl::ConsumePrefix(&type, ")")) {
433       return false;
434     }
435 
436     // Construct a valid text proto message to parse.
437     string to_parse;
438     if (is_list) {
439       // TextFormat parser considers "i: 7" to be the same as "i: [7]",
440       // but we only want to allow list values with [].
441       StringPiece cleaned = text;
442       str_util::RemoveLeadingWhitespace(&cleaned);
443       str_util::RemoveTrailingWhitespace(&cleaned);
444       if (cleaned.size() < 2 || cleaned[0] != '[' ||
445           cleaned[cleaned.size() - 1] != ']') {
446         return false;
447       }
448       cleaned.remove_prefix(1);
449       str_util::RemoveLeadingWhitespace(&cleaned);
450       if (cleaned.size() == 1) {
451         // User wrote "[]", so return empty list without invoking the TextFormat
452         // parse which returns an error for "i: []".
453         out->Clear();
454         out->mutable_list();
455         return true;
456       }
457       to_parse = strings::StrCat("list { ", field_name, ": ", text, " }");
458     } else {
459       to_parse = strings::StrCat(field_name, ": ", text);
460     }
461     if (field_name == "tensor") {
462       if (!ParseAttrValueHelper_TensorNestsUnderLimit(kMaxTensorNestDepth,
463                                                       to_parse)) {
464         return false;
465       }
466     }
467     return ProtoParseFromString(to_parse, out);
468   }
469 
470   void SetAttrValue(const AttrValue& value, AttrValue* out) { *out = value; }
```

```
471
472 #define DEFINE_SET_ATTR_VALUE_ONE(ARG_TYPE, FIELD) \
473   void SetAttrValue(ARG_TYPE value, AttrValue* out) { out->set_##FIELD(value); }
474
475 #define DEFINE_SET_ATTR_VALUE_LIST(ARG_TYPE, FIELD)                      \
476   void SetAttrValue(ARG_TYPE value, AttrValue* out) {                    \
477     out->mutable_list()->Clear(); /* create list() even if value empty */ \
478     for (const auto& v : value) {                                        \
479       out->mutable_list()->add_##FIELD(v);                               \
480     }                                                                    \
481   }
482
483 #define DEFINE_SET_ATTR_VALUE_BOTH(ARG_TYPE, FIELD) \
484   DEFINE_SET_ATTR_VALUE_ONE(ARG_TYPE, FIELD)        \
485   DEFINE_SET_ATTR_VALUE_LIST(gtl::ArraySlice<ARG_TYPE>, FIELD)
486
487 DEFINE_SET_ATTR_VALUE_ONE(const string&, s)
488 DEFINE_SET_ATTR_VALUE_LIST(gtl::ArraySlice<string>, s)
489 DEFINE_SET_ATTR_VALUE_BOTH(const char*, s)
490 DEFINE_SET_ATTR_VALUE_BOTH(int64_t, i)
491 DEFINE_SET_ATTR_VALUE_BOTH(int32_t, i)
492 DEFINE_SET_ATTR_VALUE_BOTH(float, f)
493 DEFINE_SET_ATTR_VALUE_BOTH(double, f)
494 DEFINE_SET_ATTR_VALUE_BOTH(bool, b)
495 DEFINE_SET_ATTR_VALUE_LIST(const std::vector<bool>&, b)
496 DEFINE_SET_ATTR_VALUE_LIST(std::initializer_list<bool>, b)
497 DEFINE_SET_ATTR_VALUE_BOTH(DataType, type)
498
499 void SetAttrValue(const tstring& value, AttrValue* out) {
500   out->set_s(value.data(), value.size());
501 }
502
503 void SetAttrValue(gtl::ArraySlice<tstring> value, AttrValue* out) {
504   out->mutable_list()->Clear();
505   for (const auto& v : value) {
506     out->mutable_list()->add_s(v.data(), v.size());
507   }
508 }
509
510 void SetAttrValue(StringPiece value, AttrValue* out) {
511   out->set_s(value.data(), value.size());
512 }
513
514 void SetAttrValue(const gtl::ArraySlice<StringPiece> value, AttrValue* out) {
515   out->mutable_list()->Clear();  // Create list() even if value empty.
516   for (const auto& v : value) {
517     out->mutable_list()->add_s(v.data(), v.size());
518   }
519 }
```

```
520
521 void MoveAttrValue(std::vector<string>&& value, AttrValue* out) {
522   out->mutable_list()->Clear();  // Create list() even if value empty.
523   for (auto& v : value) {
524     out->mutable_list()->add_s(std::move(v));
525   }
526 }
527
528 void SetAttrValue(const TensorShape& value, AttrValue* out) {
529   value.AsProto(out->mutable_shape());
530 }
531
532 void SetAttrValue(const TensorShapeProto& value, AttrValue* out) {
533   *out->mutable_shape() = value;
534 }
535
536 void SetAttrValue(const PartialTensorShape& value, AttrValue* out) {
537   value.AsProto(out->mutable_shape());
538 }
539
540 void SetAttrValue(const gtl::ArraySlice<TensorShape> value, AttrValue* out) {
541   out->mutable_list()->Clear();  // Create list() even if value empty.
542   for (const auto& v : value) {
543     v.AsProto(out->mutable_list()->add_shape());
544   }
545 }
546
547 void SetAttrValue(gtl::ArraySlice<TensorShapeProto> value, AttrValue* out) {
548   out->mutable_list()->Clear();  // Create list() even if value empty.
549   for (const auto& v : value) {
550     *out->mutable_list()->add_shape() = v;
551   }
552 }
553
554 void SetAttrValue(const gtl::ArraySlice<PartialTensorShape> value,
555                   AttrValue* out) {
556   out->mutable_list()->Clear();  // Create list() even if value empty.
557   for (const auto& v : value) {
558     v.AsProto(out->mutable_list()->add_shape());
559   }
560 }
561
562 void SetAttrValue(const Tensor& value, AttrValue* out) {
563   if (value.NumElements() > 1) {
564     value.AsProtoTensorContent(out->mutable_tensor());
565   } else {
566     value.AsProtoField(out->mutable_tensor());
567   }
568 }
```

```
569
570 void SetAttrValue(const gtl::ArraySlice<Tensor> value, AttrValue* out) {
571   out->mutable_list()->Clear();  // Create list() even if value empty.
572   for (const auto& v : value) {
573     if (v.NumElements() > 1) {
574       v.AsProtoTensorContent(out->mutable_list()->add_tensor());
575     } else {
576       v.AsProtoField(out->mutable_list()->add_tensor());
577     }
578   }
579 }
580
581 void SetAttrValue(const TensorProto& value, AttrValue* out) {
582   *out->mutable_tensor() = value;
583 }
584
585 void SetAttrValue(const gtl::ArraySlice<TensorProto> value, AttrValue* out) {
586   out->mutable_list()->Clear();  // Create list() even if value empty.
587   for (const auto& v : value) {
588     *out->mutable_list()->add_tensor() = v;
589   }
590 }
591
592 void SetAttrValue(const NameAttrList& value, AttrValue* out) {
593   *out->mutable_func() = value;
594 }
595
596 void SetAttrValue(gtl::ArraySlice<NameAttrList> value, AttrValue* out) {
597   out->mutable_list()->Clear();  // Create list() even if value empty.
598   for (const auto& v : value) {
599     *out->mutable_list()->add_func() = v;
600   }
601 }
602
603 bool AreAttrValuesEqual(const AttrValue& a, const AttrValue& b,
604                         bool allow_false_negatives) {
605   if (a.type() != b.type()) {
606     return false;
607   } else if (a.type() != DT_INVALID && b.type() != DT_INVALID) {
608     return a.type() == b.type();
609   }
610
611   if (a.has_tensor() != b.has_tensor()) {
612     return false;
613   } else if (a.has_tensor() && b.has_tensor()) {
614     return AreTensorProtosEqual(a.tensor(), b.tensor(), allow_false_negatives);
615   }
616
617   // `func` field contains a nested AttrValue. Compare such AttrValues
```

```
  // recursively.
  if (a.has_func() != b.has_func()) {
    return false;
  } else if (a.has_func() && b.has_func()) {
    const NameAttrList& af = a.func();
    const NameAttrList& bf = b.func();
    if (af.name() != bf.name()) return false;
    std::unordered_map<string, AttrValue> am(af.attr().begin(),
                                             af.attr().end());
    for (const auto& bm_pair : bf.attr()) {
      const auto& iter = am.find(bm_pair.first);
      if (iter == am.end()) return false;
      if (!AreAttrValuesEqual(iter->second, bm_pair.second,
                              allow_false_negatives))
        return false;
      am.erase(iter);
    }
    if (!am.empty()) return false;
    return true;
  }

  // All other fields in AttrValue have deterministic representations.
  // It is safe to compare their serialized strings.
  return AreSerializedProtosEqual(a, b);
}

uint64 AttrValueHash(const AttrValue& a) {
  return AttrValueHash(a, TensorProtoHash);
}

uint64 FastAttrValueHash(const AttrValue& a) {
  return AttrValueHash(a, FastTensorProtoHash);
}

bool HasPlaceHolder(const AttrValue& val) {
  switch (val.value_case()) {
    case AttrValue::kList: {
      for (const NameAttrList& func : val.list().func()) {
        for (const auto& p : func.attr()) {
          if (HasPlaceHolder(p.second)) {
            return true;
          }
        }
      }
      break;
    }
    case AttrValue::kFunc:
      for (const auto& p : val.func().attr()) {
        if (HasPlaceHolder(p.second)) {
```

```
667           return true;
668         }
669       }
670       break;
671     case AttrValue::kPlaceholder:
672       return true;
673     default:
674       break;
675   }
676   return false;
677 }
678
679 bool SubstitutePlaceholders(const SubstituteFunc& substitute,
680                             AttrValue* value) {
681   switch (value->value_case()) {
682     case AttrValue::kList: {
683       for (NameAttrList& func : *value->mutable_list()->mutable_func()) {
684         for (auto& p : *func.mutable_attr()) {
685           if (!SubstitutePlaceholders(substitute, &p.second)) {
686             return false;
687           }
688         }
689       }
690       break;
691     }
692     case AttrValue::kFunc:
693       for (auto& p : *(value->mutable_func()->mutable_attr())) {
694         if (!SubstitutePlaceholders(substitute, &p.second)) {
695           return false;
696         }
697       }
698       break;
699     case AttrValue::kPlaceholder:
700       return substitute(value->placeholder(), value);
701     case AttrValue::VALUE_NOT_SET:
702       return false;
703     default:
704       break;
705   }
706   return true;
707 }
708
709 }  // namespace tensorflow
```