

a1320ec1ea ▾

...

tensorflow / tensorflow / core / framework / full_type_util.cc



Dan Moldovan Define type information for mutex ops. ... ✖

History

1 contributor

132 lines (105 sloc) | 4.15 KB

...

```

1  /* Copyright 2020 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/full_type_util.h"
17
18 #include "tensorflow/core/framework/attr_value.pb.h"
19 #include "tensorflow/core/framework/full_type.pb.h"
20 #include "tensorflow/core/framework/node_def.pb.h"
21 #include "tensorflow/core/framework/node_def_util.h"
22 #include "tensorflow/core/framework/op_def.pb.h"
23 #include "tensorflow/core/framework/types.h"
24 #include "tensorflow/core/platform/statusor.h"
25
26 namespace tensorflow {
27
28 namespace full_type {
29
```

```

30 OpTypeConstructor Nullary(FullTypeId t) {
31     return [t](OpDef* op_def) {
32         FullTypeDef* tdef =
33             op_def->mutable_output_arg(0)->mutable_experimental_full_type();
34         tdef->set_type_id(t);
35         return Status::OK();
36     };
37 }
38
39 OpTypeConstructor Unary(FullTypeId t, const string& var_name) {
40     return [t, var_name](OpDef* op_def) {
41         FullTypeDef* tdef =
42             op_def->mutable_output_arg(0)->mutable_experimental_full_type();
43         tdef->set_type_id(t);
44
45         FullTypeDef* arg = tdef->add_args();
46         arg->set_type_id(TFT_VAR);
47         arg->set_s(var_name);
48
49         return Status::OK();
50     };
51 }
52
53 OpTypeConstructor UnaryGeneric(FullTypeId t) {
54     return [t](OpDef* op_def) {
55         FullTypeDef* tdef =
56             op_def->mutable_output_arg(0)->mutable_experimental_full_type();
57         tdef->set_type_id(t);
58
59         FullTypeDef* arg = tdef->add_args();
60         arg->set_type_id(TFT_ANY);
61
62         return Status::OK();
63     };
64 }
65
66 OpTypeConstructor UnaryTensorContainer(FullTypeId t, FullTypeId dtype) {
67     return [t, dtype](OpDef* op_def) {
68         FullTypeDef* tdef =
69             op_def->mutable_output_arg(0)->mutable_experimental_full_type();
70         tdef->set_type_id(t);
71
72         FullTypeDef* arg = tdef->add_args();
73         arg->set_type_id(TFT_TENSOR);
74         FullTypeDef* targ = arg->add_args();
75         targ->set_type_id(dtype);
76
77         return Status::OK();
78     };

```

```

79 }
80
81 StatusOr<FullTypeDef> SpecializeType(const AttrSlice& attrs,
82                                     const OpDef& op_def) {
83     FullTypeDef ft;
84     ft.set_type_id(TFT_PRODUCT);
85
86     for (int i = 0; i < op_def.output_arg_size(); i++) {
87         auto* t = ft.add_args();
88
89         *t = op_def.output_arg(i).experimental_full_type();
90
91         // Resolve dependent types. The convention for op registrations is to use
92         // attributes as type variables.
93         // See https://www.tensorflow.org/guide/create_op#type_polymorphism.
94         // Once the op signature can be defined entirely in FullType, this
95         // convention can be deprecated.
96         //
97         // Note: While this code performs some basic verifications, it generally
98         // assumes consistent op defs and attributes. If more complete
99         // verifications are needed, they should be done by separately, and in a
100        // way that can be reused for type inference.
101        for (int j = 0; j < t->args_size(); j++) {
102            auto* arg = t->mutable_args(i);
103            if (arg->type_id() == TFT_VAR) {
104                const auto* attr = attrs.Find(arg->s());
105                DCHECK(attr != nullptr);
106                if (attr->value_case() == AttrValue::kList) {
107                    const auto& attr_list = attr->list();
108                    arg->set_type_id(TFT_PRODUCT);
109                    for (int i = 0; i < attr_list.type_size(); i++) {
110                        map_dtype_to_tensor(attr_list.type(i), arg->add_args());
111                    }
112
113                    } else if (attr->value_case() == AttrValue::kType) {
114                        map_dtype_to_tensor(attr->type(), arg);
115
116                    } else {
117                        return Status(error::UNIMPLEMENTED,
118                                    absl::StrCat("unknown attribute type",
119                                                  attrs.DebugString(), " key=", arg->s()));
120                    }
121
122                    arg->clear_s();
123                }
124            }
125        }
126
127        return ft;

```

```
128 }  
129  
130 } // namespace full_type  
131  
132 } // namespace tensorflow
```