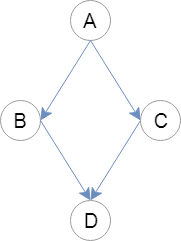
The dependency graph

Introduction

The onyx dependency graph (from now on simply referred to as graph) is a directed acyclic graph whose set of vertices is represented by a dictionary with node-ids as keys and node-instances as values. In general terms, a node is uniquely identified by the tuple (object-name, attribute-name). More about it in the following sections.

The set of edges is represented by a dictionary mapping the id of each node in the graph to the ids of its children. The topology of the graph is determined by the children whereas its state (value of each node) is determined by the nodes themselves. Parents of each node are dynamically determined from the topology of the graph.

We call descendants of a node the set of its children, grandchildren, etc; we call ancestors of a node the set of its parents, grandparents, etc.

In the example of figure 1 node A has {B, C} as children and it has no parents, node B has {D} as child and {A} as parent, node D has {B, C} as parents and no children (it’s called a leaf node or simply leaf). Node A has {A, C, D} as descendants, node D has {A, B, C} as ancestors.

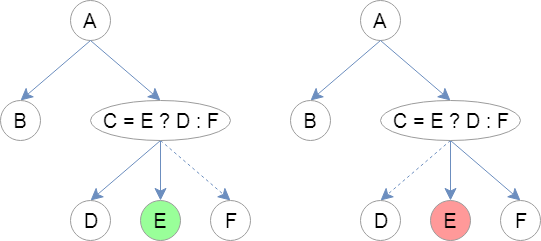
The value of a node is uniquely determined by the value of its children. This is true for each node except for the leaves, whose value (having no children) must have been set *off-graph*. Therefore, the value of each non-leaf node ends up being a (pure) function of the values of all and only the leaf nodes that are its descendants.

The topology and/or state of the of graph can be altered at any time if the value of a leaf node is changed. Nodes that in the current topology of the graph are not leaf nodes become leaves if their value is set “manually” (i.e. *off-graph*).

The set of children of a node is dynamically determined every time the value of that node is calculated. Knowledge of the children set is not needed for calculation itself but rather the children become known only when calculation takes place. The children are only needed when traversing the graph.

The onyx dependency graph is used to enable efficient (read minimal) calculations. To achieve this the graph is designed along the following principles:

1. The graph is pure, there are never side effects.
2. The graph has memory: the value of each node is calculated once and remembered for as long as the node is valid.
3. The graph is lazy: the topology of the graph remains undefined until calculation takes place. Nodes themselves are not created until their value is requested, either explicitly or by a parent.
4. Changing the value of a node triggers invalidation of its the ancestors. Invalidation of a node requires not only setting its state to invalid but also invalidating the topology of the graph by dropping all its children.

To understand the implications of invalidation let’s analyse the graphs of figure 2. In this example the value of node C is determined by that of node D if node E is true and by the value of node F if E is false. In other words, not only the value but also the topology of the graph depends on the value of node E. Invalidation needs to capture this, making sure that when the value of node E is switched from true to false node D is dropped from the set of children of node C and node F is added to it. The easiest way of achieving this is first dropping all children of node C. The new children of node C are then determined once its value requested (being now invalid, it will have to be re-calculated).

One of the key features of the graph is the ability of traversing it. For instance, it can be useful to know the set of all leaves that are descendants of a given node. A more common scenario is looking for the set of all descendants of a given node that satisfy certain requirements (such as referring to a specific object or attribute). Any implementation of the graph must ensure that the graph can be traverse at any time. Traversing is very efficient if the topology of the graph is known, otherwise it first requires calculating the values of all the relevant nodes.

Graph implementation

GraphScope context manager

The GraphScope replaces the active graph which becomes a fall back for as long as the GraphScope is active, i.e. from when \_\_enter\_\_ is executed to when \_\_exit\_\_ is executed.

The underlying graph must remain untouched so that all that happens inside the context manager is forgotten once we exit and the underlying graph becomes the active graph once more. This includes the state and topology of the graph.

To achieve this:

1. nodes that don’t exists in the underlying graph are created within the GraphScope,
2. when changing the value of a node that already exists in the underlying graph, such node as well as all its ancestors are copied in the GraphScope and set to invalid. Finally, the value of the relevant node is changed and the node is set to valid,