NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Recursive Function Definitions in Static Dataflow Graphs and their implementation in Tensorflow

Calliope P. Kostopoulou

Supervisors: **Panos Rondogiannis,** Professor NKUA
Angelos Charalambidis, Researcher NCSR

ATHENS

SEPTEMBER 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αναδρομικοί ορισμοί συναρτήσεων σε στατικούς dataflow γράφους και η υλοποίηση τους στο Tensorflow

Καλλιόπη Π. Κωστοπούλου

Επιβλέποντες: **Παναγιώτης Ροντογιάννης,** Καθηγητής ΕΚΠΑ
Άγγελος Χαραλαμπίδης, Ερευνητής ΕΚΕΦΕ

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2018

BSc THESIS

Recursive Function Definitions in Static Dataflow Graphs and their implementation in Tensorflow

Calliope P. Kostopoulou

S.N.: 1115201200084

SUPERVISORS: **Panos Rondogiannis,** Professor NKUA
Angelos Charalambidis, Researcher NCSR

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αναδρομικοί ορισμοί συναρτήσεων σε στατικούς dataflow γράφους και η υλοποίηση τους στο Tensorflow

Καλλιόπη Π. Κωστοπούλου

Α.Μ.: 1115201200084

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Παναγιώτης Ροντογιάννης,** Καθηγητής ΕΚΠΑ
Άγγελος Χαραλαμπίδης, Ερευνητής ΕΚΕΦΕ

ABSTRACT

Dataflow programming paradigm suggests that a program is represented as a graph. This representation could be thought of, perhaps, as an enhanced version of the conventional/ classic one because it contains information about both the computations that need to take place and their underlying ordering dependencies. A dataflow execution model can use this information to execute all the non-dependent code segments out of order and in parallel which increases the degree of parallelism to the maximum possible level (lack of resources may be the only limitation). However, embedding dynamic control flow features such as if-else structures, iteration or even recursive function definitions inside those graphs is not as trivial as we would like it to be. Two main approaches have been suggested in the dataflow community, so far. The first one (dynamic) leads to the creation of graphs that transform themselves on demand, during runtime, while the second one (static) proposes the creation of static, non-transforming graphs that retain their initial form throughout the whole execution. The second approach is quite more profound, as it needs the introduction of a concept called “tagging” in order to work. The various dataflow systems, that are being developed nowadays, follow either the first or the second approach when they need to support such dynamic control flow features. None of them (too cocky? is it even true?), however, does follow the second approach when it comes to supporting recursive function definitions, as the tagging mechanism that needs to be employed is considered to be rather complex and possibly is avoided. The subject of this thesis, is the proposal of a systematic way to embed recursive function definitions in static dataflow graphs, based on ideas expressed during the extended research that already has been made regarding this topic, at the past. A great part of this thesis, has also been the implementation of these ideas in a famous, rapidly growing, dataflow-based framework called Tensorflow, which was made by Google for Machine Learning purposes.

SUBJECT AREA: Dataflow Model

KEYWORDS: dataflow, tensorflow, static dataflow graphs, recursive function definitions, recursion, distributed computing, dynamic control flow

ΠΕΡΙΛΗΨΗ

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Dataflow Μοντέλο

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: dataflow, tensorflow, στατικοί dataflow γράφοι, αναδρομικοί ορισμοί συναρτήσεων, αναδρομή, κατανεμημένα συστήματα, δυναμική ροή δεδομένων

ACKNOWLEDGEMENTS

I would like to thank both Panos Rondogiannis and Angelos Charalambidis for giving me the chance to work on a challenging project and for coming up with this interesting subject on the first place.

I am especially grateful to Angelos who has been helping (baby-sitting) me for the last few months and making me consider him more of a teammate than a simple supervisor. His special psychic abilities, that make him come up with accurate solutions, provided me with great comfort when I'd see my intuition leading me to all the wrong places.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The objective of the task that was initially proposed to me, as a subject for this bachelor thesis, was to investigate ways of implementing and integrating recursion in Tensorflow by creating dataflow graphs that are static and therefore, retain their initial form at runtime, opposite to Google's approach, which relies on graphs that expand themselves during execution. This work is meant to demonstrate that the idea for producing static dataflow graphs from recursive function definitions, expressed in this [?] prior research work can be easily applied for data driven models and still be feasible. Another huge motivation, was the fact that recursion, as a feature in dataflow systems, is highly desired by people who are active in the machine learning field, as the additional expressiveness possibly leads to superior performance results in cases of recursive neural networks. A great amount of time, during my engagement with this project, was dedicated to studying the dataflow programming paradigm in general and an even greater one was spent on digging around TensorFlow's core code for acquiring the necessary knowledge as much as the confidence to add the appropriate code and implement the aforementioned ideas.

1. INTRODUCTION

1.1 Main Concept

In the conventional von Neumann model, a program is represented as a sequence of instructions, whose order of execution is implicitly described by the programmer. Memory is being treated as a stack during the execution and the data stored in there is mutable (side effects). The execution of a code segment, representing a recursive function, in the above architecture, is handled with the help of a register, called program counter, whose value always points to the address of the instruction that is meant to be executed next (instruction pointer). This code segment is supposed to be executed every time the recursive function calls itself which is a decision that, in most cases, can only be made at runtime (dynamic control flow). Every recursive call initiates the creation of an activation record (frame) which is "pushed" in stack and holds all the data that are local to that specific call (and thus "related" to each other), together. Code is executed sequentially, which causes memory to expand linearly and that is what truly allows the isolation of the data which is critical for avoiding mixing up variables that were derived by different function calls during the computations.

As opposed to the model described above, dataflow programming paradigm adopts a data driven computational model which allows the instructions to be executed out of order and in parallel and exploits the maximum parallelism inherent in a program. In dataflow, a program is represented as a graph where nodes are the operations/ instructions and edges are the flowing data. The next set of operations that are to be executed, at any given moment, comprises those that have all their inputs available, that is, nodes whose incoming edges have all received data generated by the computation of previous operations. In such a model, we have neither stack-behaved memory nor side effects (all data is immutable) as every "variable" is instantly propagated to all the operations that depend on its current value, eliminating the need for a stateful environment. So, representing recursive function definitions (or any dynamic control flow feature for that matter) in static, non-transforming dataflow graphs and executing them is not that easy to achieve. If a subgraph (set of nodes), inside the dataflow graph, represents the computations that compose the function's body, then, intuitively, we would expect that one or more of those nodes (depending on the number of times this recursive function calls itself) would send their output to the subgraph itself, creating a graph cycle, so that they can possibly re-trigger its execution. What happens though, when we have more than one calls of the same function, executing concurrently? We have multiple instances of the same operations, awaiting to be triggered by the arrival of data, at the same time. Nothing can guarantee, in this scenario, that an operation instance won't be triggered by the arrival of data derived by different function calls, hence, the correct execution cannot be ensured and the given results cannot be trusted.

While classic model is treating this problem by exploiting the linear memory expansion and making data of the same function calls reside on the same, confined memory locations, dataflow cannot follow, of course, the exact same approach. The equivalent of that solution, would be, to make the flowing data itself hold a piece of information that would indicate the function call from which they were derived. That piece of information is called "tag" in the dataflow community [?], and is introduced as the means for overcoming the difficulties described above.

The subject of this thesis, was to upgrade TensorFlow so that it can support the definition

of recursive functions inside static dataflow graphs as well as their execution, for which a tagging mechanism, as described above, needed to be deployed.

1.2 Motivation

1.2.1 Improved Expressiveness

Black Box – yet to be opened

1.2.2 Better Distribution

Google's approach for integrating recursion in TensorFlow is a lot different than the one implemented in this project. Their solution is actually based on the idea that the executed dataflow graphs can be transformed at runtime and expand themselves on demand. Each time another recursive call occurs, the executor is actually copying, repeatedly, the body of the callee function at the corresponding call site. This approach, however, has a major drawback. Dataflow's nature allows the partitioning of the main graph into multiple subgraphs that can then be deployed in distributed, heterogeneous systems and get executed in parallel without any conflicts. In Google's provided solution, the initial graph that gets partitioned does not include the bodies of the functions that will be possibly called during the execution. The "calling" of a function is represented with a special node whose main operation is to replace itself with the subgraph that corresponds to the body of that function. Thus, it is inevitable that all the recursive calls of a function will be executed, essentially, in only one machine. In cases where function bodies/subgraphs are very big or number of occurring recursive calls is huge, the performance results, theoretically, are expected to be rather inferior to the ones yielded by the "static" approach.

Even when execution takes place in only one, local machine and not a cluster, the dynamic approach has, in fact, proven itself to be marginally worse contrary to the static one, as will be discussed in the "Evaluation" section. That could possibly be caused by the additional overhead that causes the repetitive creation/initialization of the same graph.

1.2.3 Research Purposes

TensorFlow already implements other dynamic control flow features such as if-else structures and iteration with static dataflow graphs, so, their solution for recursion is actually inconsistent with those. It is interesting as a topic, for research purposes to compare the two different implementations and draw conclusions regarding their performance.

1.3 Structure of Thesis

The rest of the thesis is organized as follows:

Background is supposed to provide basic, fundamental knowledge about the concepts that are being dealt with throughout the entire thesis.

Approach provides a general description of the followed approach, in a theoretical, abstract level.

Implementation includes more technical information about the implementation that concerns mostly the TensorFlow core and its infrastructure.

Related work is the section where we examine the approaches followed by other existing dataflow systems regarding this topic.

Future work discusses ideas that could possibly extend and improve our current work.

Evaluation presents the performing results that occurred from the comparison of the two approaches (dynamic vs static one).

Conclusions provides review on the results and some final conclusions

2. BACKGROUND

2.1 Dataflow Programming Paradigm

A brief introduction of the dataflow computational model has already been given in previous sections. However, it is mandatory to provide a more formal and detailed one so that the content of this thesis is fully comprehended by those who are less familiar with the concept.

Dataflow programming paradigm suggests that the set of computations that need to take place in order to carry out a task is specified as a directed, possibly cyclic graph. In these dataflow graphs the nodes represent the computations and the edges represent the data dependencies. The execution of a node is triggered by the arrival of data at all its inputs. That means, that, at any given moment, there can be multiple nodes that are ready to “fire” and execute in parallel. This program representation captures and depicts all the existing data interdependencies and allows for the dataflow execution model to adopt an instruction ordering policy that exploits the full parallelism potential that is inherent in the program. This set of dependencies comprises the minimum number of ordering constraints that the execution model needs to respect in order to generate the correct, expected results. [?]

It is easy to infer that a model with such characteristics is ideal for distributed computing systems. A dataflow graph can easily be partitioned into multiple subgraphs, which can then be deployed in any number of different and potentially heterogeneous machines that belong in a cluster, and get executed simultaneously. The problem of transferring data between nodes that have resided on different machines, can be easily treated by the serialization of those data tokens and their transmission over the network. There is no doubt why people, in domains that deal with massive data management, are highly interested in the parallelism opportunities that this model unlocks for them. [?]

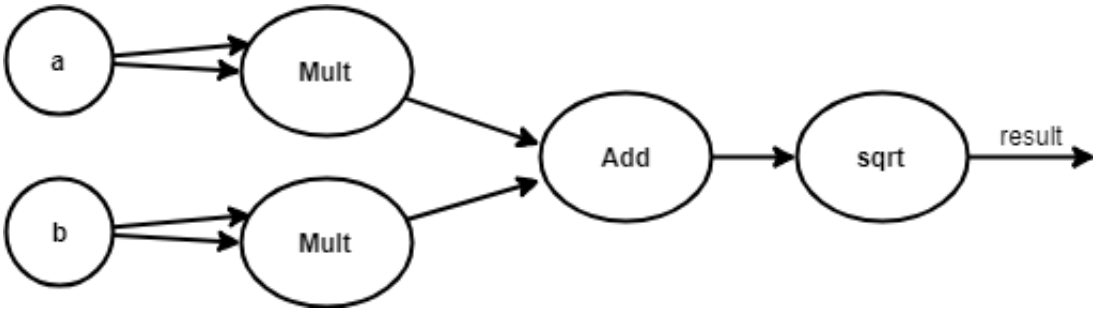

Figure 2.1: Pythagorean equation as a Dataflow graph

2.2 Dynamic Control Flow in Dataflow

It is highly desirable, as a feature for every dataflow-based system, to be able to embed dynamic control flow constructs inside the dataflow graphs and have an execution model that can handle them at runtime. The alternative to this, is an “out of graph”, client-side approach which will be better explained below. [?]

2.2.1 Out-of-Graph

Current dataflow-based frameworks, allow their clients to specify their desired computations as dataflow graphs via their APIs and provide an execution engine as well, in order to run them. The “out of graph” approach, meets the client’s requirements for expressing dynamic control flow constructs (e.g. while loops) by using the dynamic control flow support of the given client-side language. That means, that, if the framework’s API is accessible via a common, imperative language such as Python, then the client may construct a dataflow graph that corresponds to the body of a while loop and depend on the underlying python execution engine, to run repeatedly the API command that triggers the execution of the graph by the framework’s dataflow executor. This approach, of course, has some disadvantages, as it does not exploit the fact, that the graph has already travelled once, all the way, from the core’s surface to the execution engine, and has already been under many optimization or initialization processes that can be useful throughout all the iteration steps of the while loop. Some known frameworks that follow this approach are PyTorch [?] or DyNet [?].

2.2.2 In-Graph

In-graph approach suggests that the dynamic control flow constructs are embedded inside the dataflow graphs. That means, that the dataflow executor is responsible for managing all the iteration steps of the while loop as well as the logistics that will, eventually, decide its termination, without the intervention of the client’s underlying execution engine.

2.2.3 Dynamic vs Static Dataflow Graphs

Those who might consider integrating an “in-graph” approach in a dataflow-based system, must concern themselves with two critical matters. First, they need to figure out a way to depict the dynamic control flow constructs in the graph, during its creation. Then, they must make all the appropriate adjustments to the system’s execution engine, so that it knows how to properly treat that graph during its execution.

There are two discrete ways that are being used by various, modern dataflow systems for embedding dynamic control flow inside dataflow graphs.

First one suggests, that inside the graphs created by the client, any dynamic control flow construct is represented as a node whose operation is rather complex compared to all the other primitive operations. Such nodes are responsible for transforming and expanding the dataflow graphs at runtime which is the only period when the dynamic control flow decisions (logistics) can be actually resolved. They are, essentially, copying or adding the demanded subgraphs to the main graph the moment it is clear that they need to be re-computed.

Note that, in this particular approach, graphs do not contain cycles, as a cycle is nothing but an edge leading back to an already computed subgraph re-triggering its execution. In this solution, every node executes only once.

The equivalent of that solution, in the classic, control flow model, would be to modify and maybe expand the executable code (e.g. bytecode) at runtime. It is only then, in dynamic control flow cases, that the instruction to be executed next can be actually determined. That is, after the computation of the corresponding logistics has taken place. So, now, instead of updating appropriately the program counter and jumping to the right instruction, wherever that may be, we insert the correct instruction, dynamically to the default, following address, where the PC is normally supposed to point next (PC+4 in most cases).

The second approach, results in the creation of static dataflow graphs whose form will not be changed by any special, non-primitive nodes during the execution. Any set of nodes corresponding to the body of a while-loop, the body of a recursive function or the bodies of an if-else's branches, is now integrated in the graphs along with an additional set of primitive operators that constitutes the tagging mechanism that will ensure the correct flow of execution. The notion of "tags", as already mentioned before, serves as a way for specifying the context under which every node/operator is executed. Contrary to the previous approach, this one allows the creation of cycles as there would be no other way, to express the need for re-computing a certain subgraph in an environment where graphs remain static. This introduces the problem of mixed-up data, in cases of multiple instances of same operations executing simultaneously (overlapping recursive function calls or iterations) and is resolved by that employed mechanism that tags the data and makes them uniquely identified. A computation will now take place, only the moment when all the node's incoming edges have received data that belong in the same context. The tag can be anything one desires, as long as it can uniquely "paint" the data, based on the context under which they were derived.

Iteration and recursion are nothing but two different mechanisms for triggering the repetitive execution of a certain code-segment (here subgraph). One can say that the first one is, essentially, a subset of the second one, as every possible iteration-loop can be expressed as a tail recursive function.

In the dynamic approach, we would expect that an iterative construct would trigger the linear expansion of the graph, whereas, a recursive one, would cause the graph to expand in a tree-like way. That said, we can now have a sense, perhaps, of how the tag, in each case, must be represented.

Non-formally speaking, in the simple case of one iteration loop, we have multiple instances of the same subgraph (loop's body) firing whenever they are triggered, and creating a "chain" as the loop keeps unfolding. In that case, a simple number belonging in the set of natural numbers would suffice as a tag that would make one chain piece distinguishable from the others. However, if we are able, at each given moment, to trigger the simultaneous execution of that particular subgraph an arbitrary amount of times by different call sites, as happens with recursion, that representation would not be satisfactory. Intuitively we would need a more profound way to depict the topological position of each subgraph-instance inside the "function-calls tree" and that would be by using lists of natural numbers with unspecified length. [?] (Is it too much intuition?)

2.3 Yaghi's Transformation

The problem of how recursion can be expressed in static dataflow graphs and the tagging mechanism that needs to be employed in order to track the depth and path of each recursive function call, has already been resolved in past research work and a formal basis for the aforementioned matters has already been provided. [?]

There has been described, a systematic way for producing static dataflow graphs from potentially recursive, function definitions based on an algorithm called "Yaghi's transformation", proposed by A. Yaghi as a subject for his PhD dissertation, in 1984.

This algorithm provides a method for transforming first-order functional programs into intentional ones which, as the conducted research suggests, can be easily deployed and executed in dataflow architectures running a demand-driven computational model, called Eduction.

The above work, introduces the idea of expressing tags as integer-lists of arbitrary length and essentially describes the additional, primitive, context switching operators that implement the tagging mechanism we need for the static approach, that was mentioned earlier. Applying the ideas that were expressed in that research, to dataflow systems that run a data-driven execution model, was rather easy, but a formal, theoretical proof of the data-driven "version" should, definitely, be provided.

Blah blah some more.

2.4 TensorFlow

TensorFlow is a rapidly-growing, open-sourced project that is mainly introduced as a system for large-scale machine learning. It is based on the dataflow computational model, which unlocks high parallelism and distribution opportunities and makes it an ideal framework for domains that deal with complex and costly mathematical computations, in general. It was created by Google Brain Team's researchers and it is now being receiving contributions from all over the world.

It is one of the numerous dataflow-based systems, created by the Machine Learning and Big Data communities, where they are in great need for new, sophisticated frameworks, that will allow them to fully exploit all of their resources in order to manage huge amount of data in a reasonable amount of time.

TensorFlow is, basically, a library, whose API allows the users to represent their desired computations as dataflow graphs and set up their execution. They are able to configure the cluster on which the graph will be deployed, specify any node placement constraints that will affect the graph partitioning, select the graph-optimizations that will be, eventually, applied and choose between many more useful setting options.

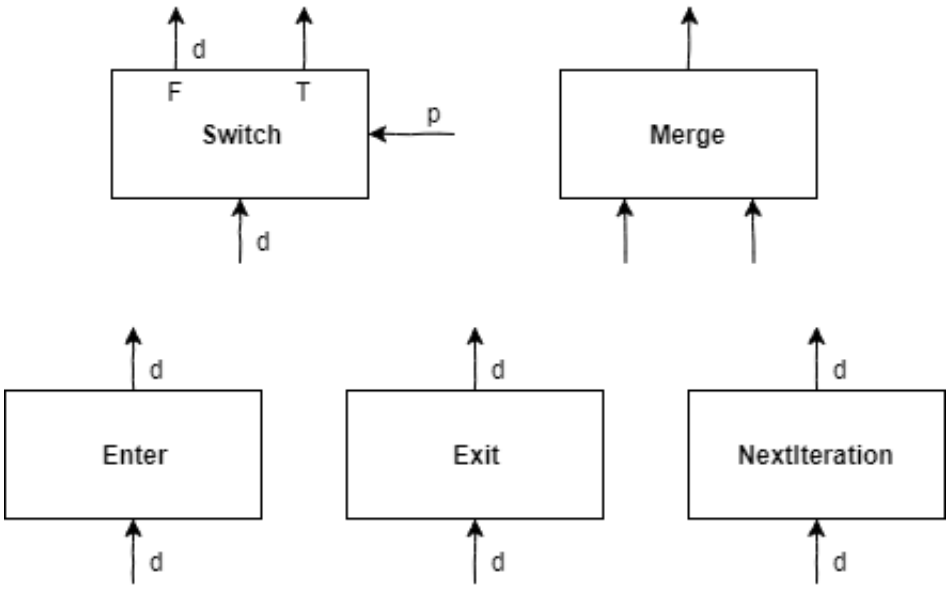

Figure 2.2: Dynamic Control Flow Operators in TensorFlow

2.4.1 Dynamic Control Flow in TensorFlow

Google favors the in-graph approach for integrating dynamic control flow features in TensorFlow and argues that this implementation yields much better performance results compared to the alternative, mainly because it enables the performing of whole-graph optimizations. In addition to that, they have chosen to implement the iteration and if-else features by following the ”static” approach that was described before. That means, that they are embedding while-loops and if-else constructs inside non-transforming, static graphs and have created a special set of primitive operators that will ensure the correct flow of execution. [?]

These operators, as well as their semantics, are described below:

- ***Enter:*** TensorFlow introduces the concept of ”frames”, which is nothing but a way for describing the context under which a set of computations, composing the body of an iterative construct, will actually take place. Frames are groups of node-instances that are identified by the same shared tag and execute only once under that context. Enter is, essentially, a context-switching operator that forwards its input to a child frame. It is possible that more than one Enter operators may send their inputs to the same frame and in that case only the first one will take the responsibility of creating and initializing it.
- ***NextIteration:*** It is there to create the graph-cycle needed in iteration, in order to express the re-triggering of a while-body’s execution. It is similar to the Enter operator, as it also forwards its input to a uniquely identified frame. Their difference essentialy lies in the way those two operators update their inputs’ tags.
- ***Exit:*** It is responsible for revoking the actions of the corresponding Enter operator, by forwarding its input to the outer-frame. Again, multiple ”Exit” operators may exist inside the current-frame, so the last of them that is being triggered is also taking the responsibility of destroying the frame in which it belongs.

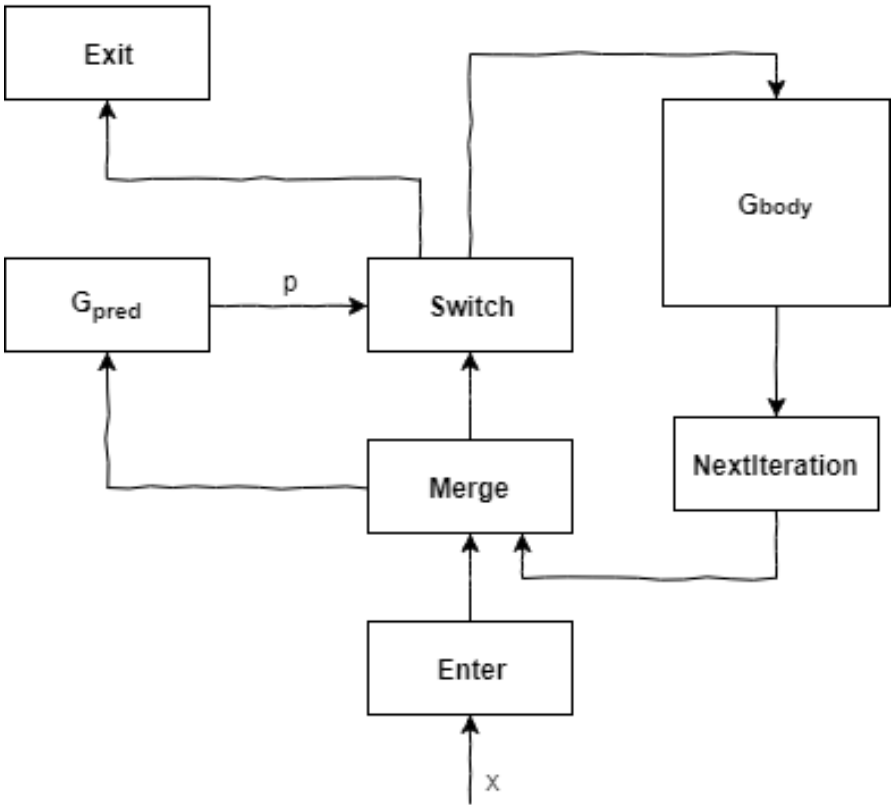

Figure 2.3: Dataflow graph for a while loop in TensorFlow

- ***Switch:*** It is used for the representation of if-else constructs and it's job is to ”direct” the execution to the correct branch. It receives two inputs of which the first one is the data that needs to be propagated to the correct subgraph and the second one is the result of the computed if-else logistics (predicate). Depending on the predicate's truth value the switch operator sends the data to either its first or its second output.

  In fact, for reasons that concern the support of distributed execution and will be better explained below, the data is actually being sent to both branches, equally, except that, the correct output is forwarding the propagated data as ”alive” and the wrong one as ”dead”, which is a phenomenon inofficially described as ”deadness propagation”.

- ***Merge:*** It goes as a pair with the Switch operator and is added to the place where the two branches join, in order to collect the if-else' s generated output. It expects two inputs, of which only one will be valid and forwarded, the one generated by the taken-branch. Merge operator, however, is playing another, significant role in the representation of iterative constructs, i.e. while loops. It is placed as a successor to all the existing Enter/NextIteration operators, so that it can distinguish between the first time we execute the body of a while loop and all the following ones. That is, so that it will always provide the correct input to the while's body-subgraph.

```
Eval(Switch(p, d), c) = (r1, r2), where
  r1 = (value(d), p || is_dead(d), tag(d))
  r2 = (value(d), !p || is_dead(d), tag(d))

Eval(Merge(d1, d2), c) = r, where
  r = if is_dead(d1) then d2 else d1

Eval(Enter(d, name), c) = r, where
  r = (value(d), is_dead(d), tag(d)/name/0)

Eval(Exit(d), c) = r, where
  r = (value(d), is_dead(d), c.parent.tag)

Eval(NextIteration(d), c) = r, where
  tag(d) = tag1/name/n
  r = (value(d), is_dead(d), tag1/name/(n+1))

Eval(Op(d1, ..., dm), c) = (r1, ..., rn), where
  value(r1), ..., value(rn) =
    Op(value(d1), ..., value(dm))
  is_dead(ri) =
    is_dead(d1) || ... || is_dead(dm), for all i
  tag(ri) = tag(d1), for all i
```

Figure 2.4: Evaluation rules for control-flow operators in TensorFlow

Figures ***2.2***, ***2.3*** and ***2.4*** can be found in the ”Dynamic Control Flow in Large-Scale Machine Learning” [?] where the dynamic control flow support in TensorFlow is being discussed in much more detail.

As for the case of recursion, TensorFlow’s support is still at a rudimentary level. They built up a way for allowing users to define recursive function definitions via the python API, and supported it up until their r.1.4 TensorFlow release. They removed it, however, later on, possibly termporarily, due to conflicts occured in their future development. Contrary to the other two dynamic control flow features, Google chooses to support functions and thus, recursion, by following the ”dynamic approach”. They have created an operator named ”Call”, that is placed as a node inside the graph and represents the calling of a function. Every such node contains important information that associates it with the function definition that is being called. ”Call” node’ s job is, essentially, to replace itself with the body of that function, causing the dynamic expansion of the graph, as many times as needed.

The disantvantages of this particular approach, have been already discussed in the ”Motivation” subsection.

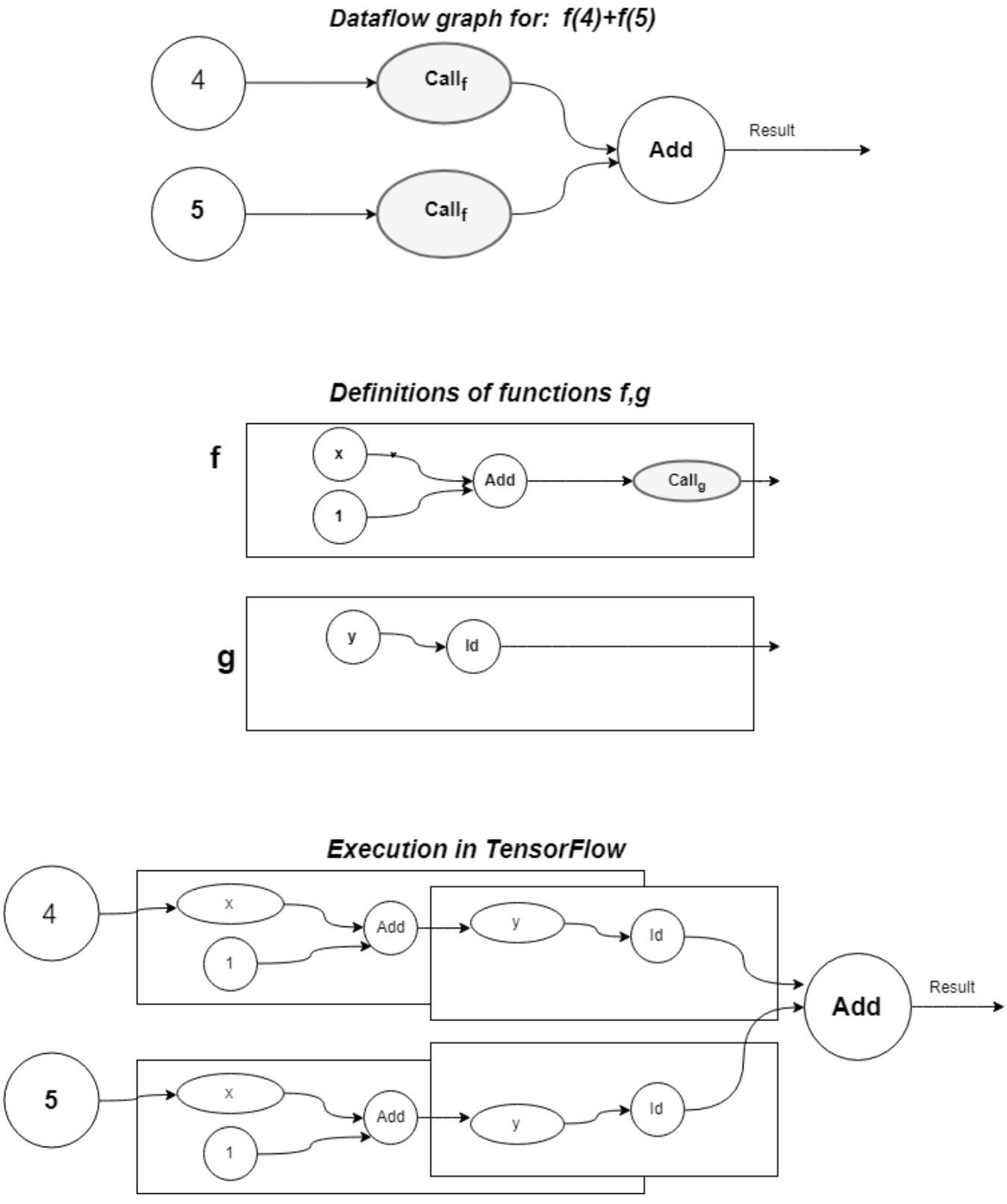

Figure 2.5: Execution of functions in TensorFlow

3. APPROACH

4. IMPLEMENTATION

4.1 Graph Transformation

4.2 Local Execution

4.3 Distributed Execution

5. RELATED WORK

6. FUTURE WORK

6.1 Higher Order Functions

6.2 Automatic Differentiation

7. EVALUATION

8. CONCLUSIONS

ABBREVIATIONS - ACRONYMS

RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
OWL	Web Ontology Language
OGC	Open Geospatial Consortium

APPENDIX A. FIRST APPENDIX