



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

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

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2018

BSc THESIS

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

Calliope P. Kostopoulou

S.N.: 1115201200084

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

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

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

Καλλιόπη Π. Κωστοπούλου

A.M.: 1115201200084

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

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: Μοντέλο Ροής Δεδομένων

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

1	INTRODUCTION	13
1.1	Main Concept	13
1.2	Motivation	14
1.2.1	Improved Expressiveness	14
1.2.2	Better Distribution	14
1.2.3	Research Purposes	14
1.3	Structure of Thesis	14
2	BACKGROUND	16
2.1	Dataflow Programming Paradigm	16
2.2	Dynamic Control Flow in Dataflow	17
2.2.1	Out-of-Graph	17
2.2.2	In-Graph	17
2.2.3	Dynamic vs Static Dataflow Graphs	17
2.3	Yaghi's Transformation	19
2.4	TensorFlow	19
2.4.1	Dynamic Control Flow in TensorFlow	20
3	APPROACH	24
4	IMPLEMENTATION	27
4.1	Graph Transformation	27
4.2	Local Execution	30
4.3	Distributed Execution	31
5	RELATED WORK	34
6	FUTURE WORK	35
6.1	Higher Order Functions	35
6.2	Automatic Differentiation	35
7	EVALUATION	36
8	CONCLUSIONS	37

ABBREVIATIONS - ACRONYMS**38****BIBLIOGRAPHY****39**

LIST OF FIGURES

2.1	Pythagorean equation as a Dataflow graph	16
2.2	Dynamic Control Flow Operators in TensorFlow	20
2.3	Dataflow graph for a while loop in TensorFlow	21
2.4	Evaluation rules for control-flow operators in TensorFlow	22
2.5	Calling functions in TensorFlow	23
3.1	One call to a simple, non-recursive function	25
3.2	Two calls to a simple, non-recursive function	26
3.3	Factorial	26
4.1	Function Transformation	28
4.2	TensorFlow graphs for factorial in "static" and "dynamic" approach	29
4.3	Distributed execution of a while-loop in TensorFlow	32

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 different locations from which 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 sub-graphs 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 performance 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 the previous section. 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.

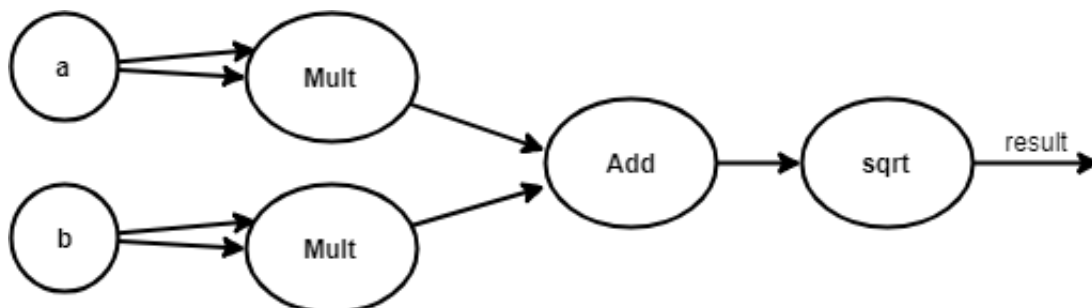


Figure 2.1: Pythagorean equation as a Dataflow graph

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 [?] 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 when multiple instances of same operations are being executed 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 resulting “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 Education.

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.

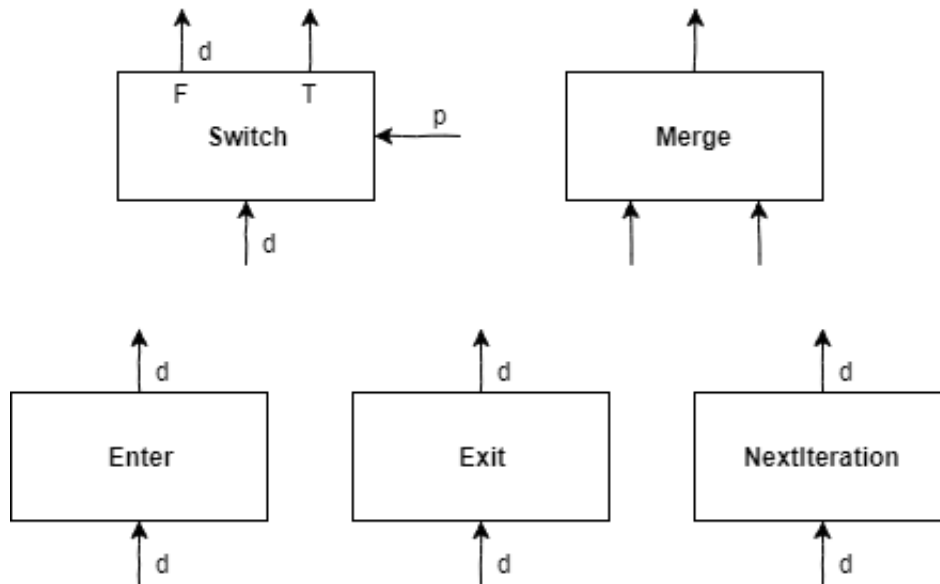


Figure 2.2: Dynamic Control Flow Operators in TensorFlow

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 new, uniquely identified context. Their difference essentially lies in the way those two operators update the tags of their inputs.
- **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.

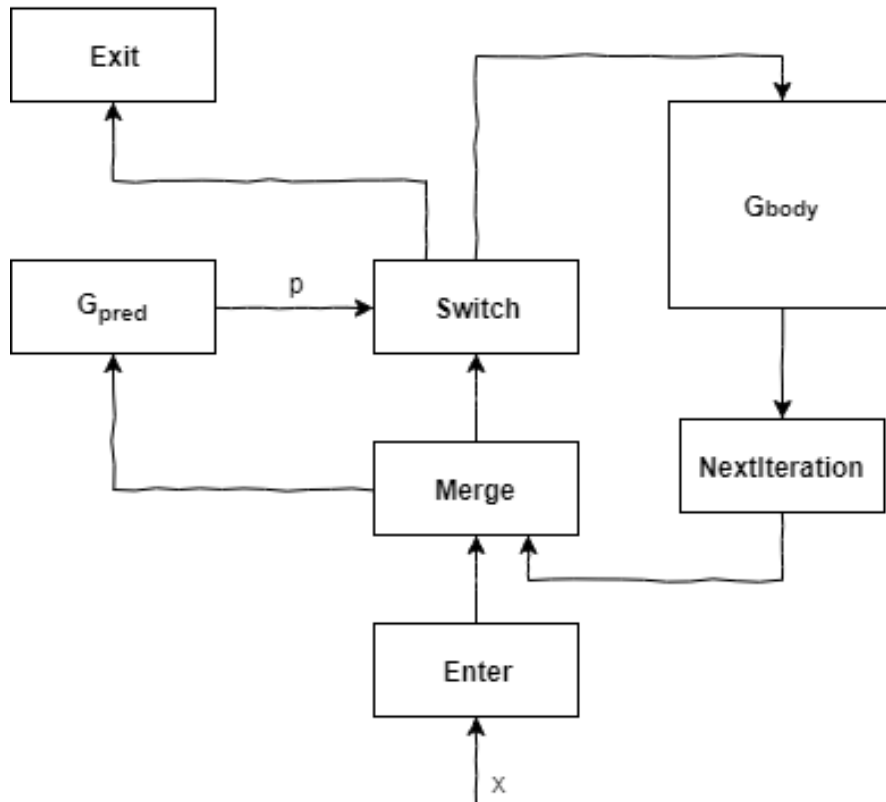


Figure 2.3: Dataflow graph for a while loop in TensorFlow

- **Switch:** It is used for the representation of if-else constructs and its 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 in the following sections, 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 unofficially 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" paper [?] 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 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 temporarily) due to conflicts occurred 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 definition of the function 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 real implementation might be a slight derivation of what was described above but the essence of the approach is exactly the same.

The disadvantage of this particular approach, has been already discussed in the "Motivation" subsection.

Below, we present an example of calling functions, expressed as a functional program and the way it will be converted into a dataflow graph and get executed in TensorFlow:

$result = f(4) + f(5)$

$$f(x) = g(x + 1)$$

$$g(y) = y$$

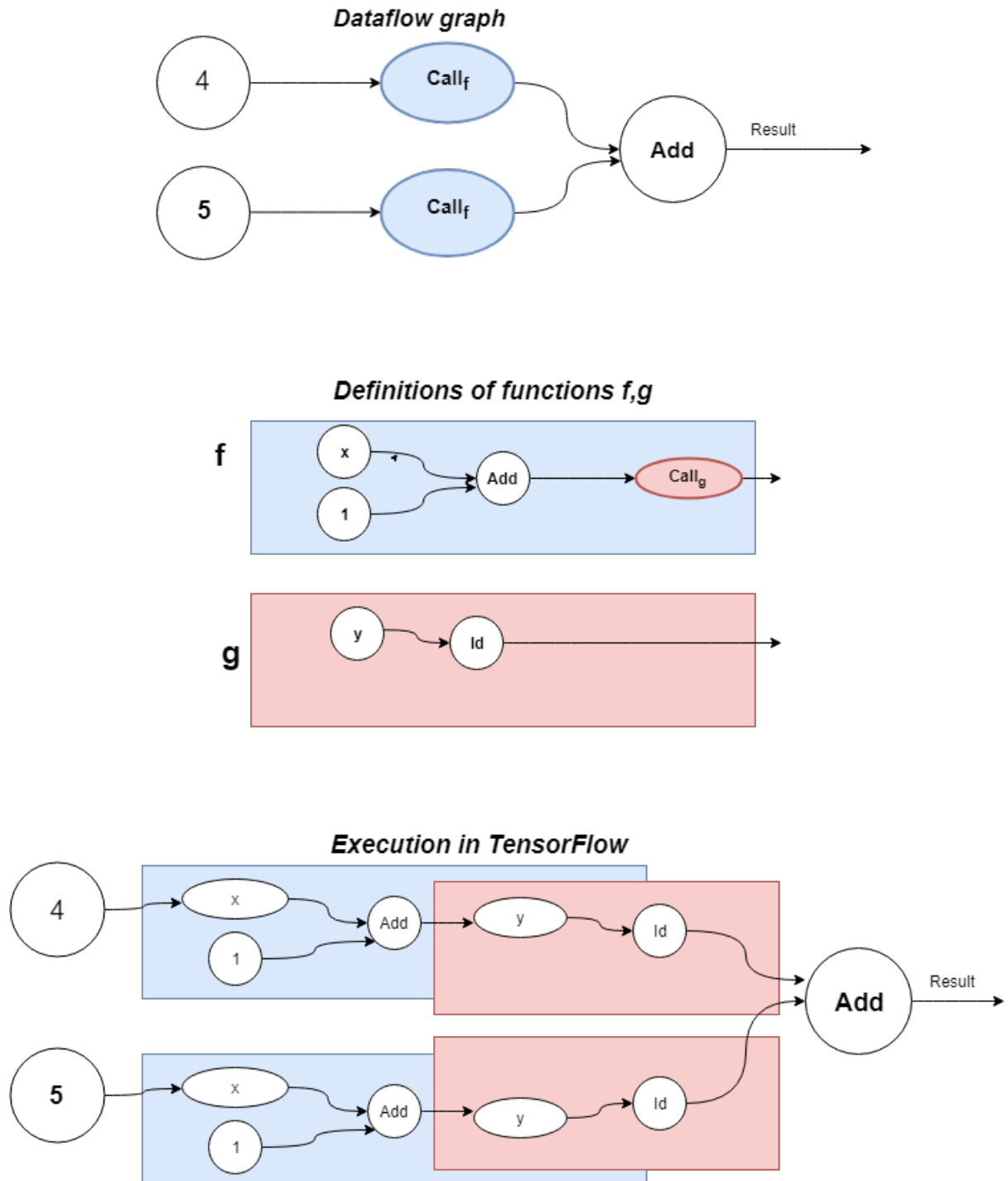


Figure 2.5: Calling functions in TensorFlow

3. APPROACH

We can resolve the problem of integrating recursion in dataflow-based systems, without allowing any graph-transformations at runtime ("static" approach), by first addressing the following critical matters.

We, first, need to establish a way for representing recursive function definitions as dataflow graphs. That can be easily treated, as already mentioned before, by allowing the occurrence of cycles inside those graphs. If a subgraph (set of nodes), represents the computations that compose the body of a function, then we represent a recursive call to that function by creating a cycle between the node that corresponds to the call site and the subgraph itself. That edge unlocks the possibility of re-triggering the subgraph's computation and thus, re-executing the function as many times as needed, without dynamically transforming the main executable graph.

Then, we must ensure that, if a function is being called, at least once, inside the program, then the subgraph that constitutes the representation of that function will have to be statically integrated in the main subgraph. Every call to that function from the outer-main graph, will be represented from now on, as an edge, connecting the node that corresponds to the call site to the entering point of the callee function/subgraph.

The form of the resulting graph, however, is not complete yet. We also have to specify an additional set of two primitive operators that will constitute the tagging mechanism that needs to be employed. These operators will also be added to the graph and will ensure the correct flow of execution.

We call these two operators "Call" and "Return" and their function is to guard the entrance/exit of the data to/from the function's scope, respectively.

- **Call:** This operator is placed as a guard to all the entry points that lead to the body of a function. It receives, as input, the function's arguments, as those are yielded by the caller and it updates their tags appropriately, before forwarding them to the correct function nodes. This generated tag will be uniquely identifying, at runtime, that specific function instance and will constitute the context under which all the entering data will eventually be executed. This ensures that data derived by different, overlapping calls to the same function won't get mixed up during the execution.
- **Return:** This operator is placed as a guard to all the function's exit points. It receives, as input, data generated by the "returning nodes" of that function (function's outputs) and it updates their tags appropriately, before forwarding them to the caller (outer-graph). The "Return" operator updates the tag by revoking the changes the corresponding "Call" operator previously made, retrieving that way the correct "parent-context".

All the different call-sites from where all the possible function calls may occur at runtime, are statically depicted in the initial graph and therefore, can be enumerated. Every such call-site is uniquely associated to a specific pair of "Call/Return" operators. These operators keep internally stored the call-site's numbering and they will use that information, at runtime, for creating globally unique tags.

For reasons we mentioned again earlier, tag could be an integer-list of unspecified length. This representation is satisfactory for tracking the depth and path of all the (potentially recursive) function calls.

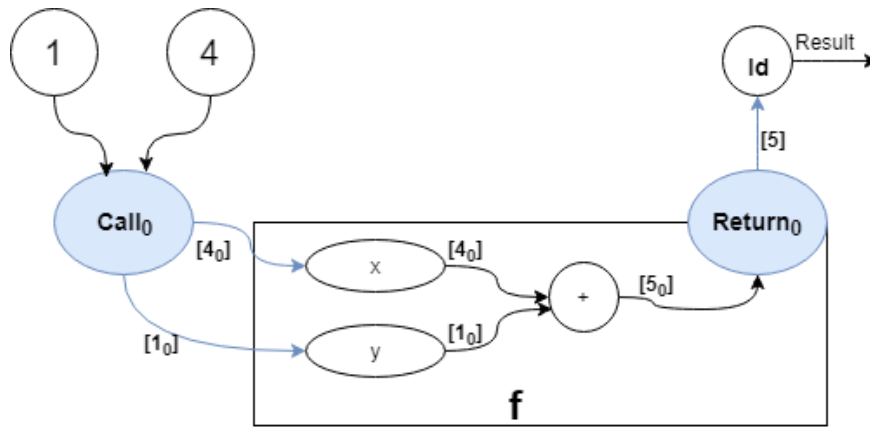


Figure 3.1: One call to a simple, non-recursive function

In the simple example, presented in **Figure 3.1**, function f is being called only once, from the main graph. We have one call-site numbered with 0. The corresponding "Call/Return" pair has internally stored that number. "Call" operator tags with it its incoming data, and "Return" operator extracts it from its input's tag.

Everything works fine in this scenario, but what happens when we have more than one calls to the same function? The "returning node" (+) of function f is connected to two different "Return" operators, as shown in **Figure 3.2** and must decide, each time, where is the correct place to send its generated output.

For treating this problem, we are extending the functionality of "Return" operator. Function f 's returning node will propagate its data to every possible "Return" operator that lives inside the body of all the potential callers and a "Return" operator will operate on an incoming input only if it decides that the input's tag concerns it.

That said, we can now be more specific about the functionality of "Call" and "Return" operators and the way they update the tags of their inputs:

- **Call** operator appends its internally stored number to the list/tag of its input and then forwards that input to its corresponding output. The generated tag is guaranteed to be globally unique.
- **Return** operator examines the last element (integer) of its input's list/tag. If it matches its internally stored number, it officially extracts it from the tag and forwards the input to its output. If not, it performs no further actions.

Note that, the dataflow graph in **Figure 3.2** is still incomplete, as it needs an additional operator that will help distinguish between the inputs provided by the different calls. The classic "Merge" operator whose semantics are described in the majority of the existing "dataflow" papers (it was also mentioned in the "Dynamic Control Flow in TensorFlow" subsection as it is used in TensorFlow, too), is ideal for this job. It is placed as a successor to all the "Call" operators and it chooses, each time, to accept only one input.

Note also, that everything discussed above, applies to cases where we have mutually recursive function calls, too.

In **Figure 3.3**, we present a more complex and interesting example of the case we are more concerned with in this particular project. That is, recursion. The example depicts the dataflow graph of a simple program where **factorial** gets called once.

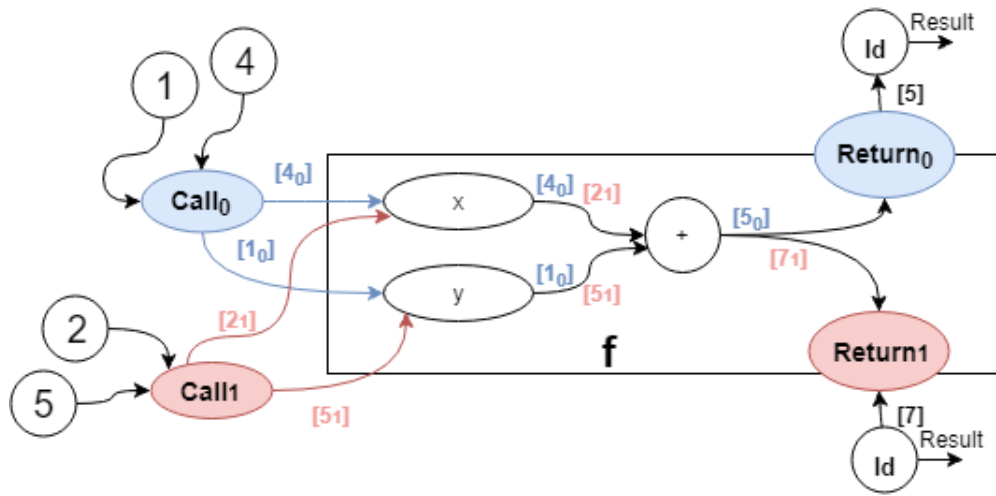


Figure 3.2: Two calls to a simple, non-recursive function

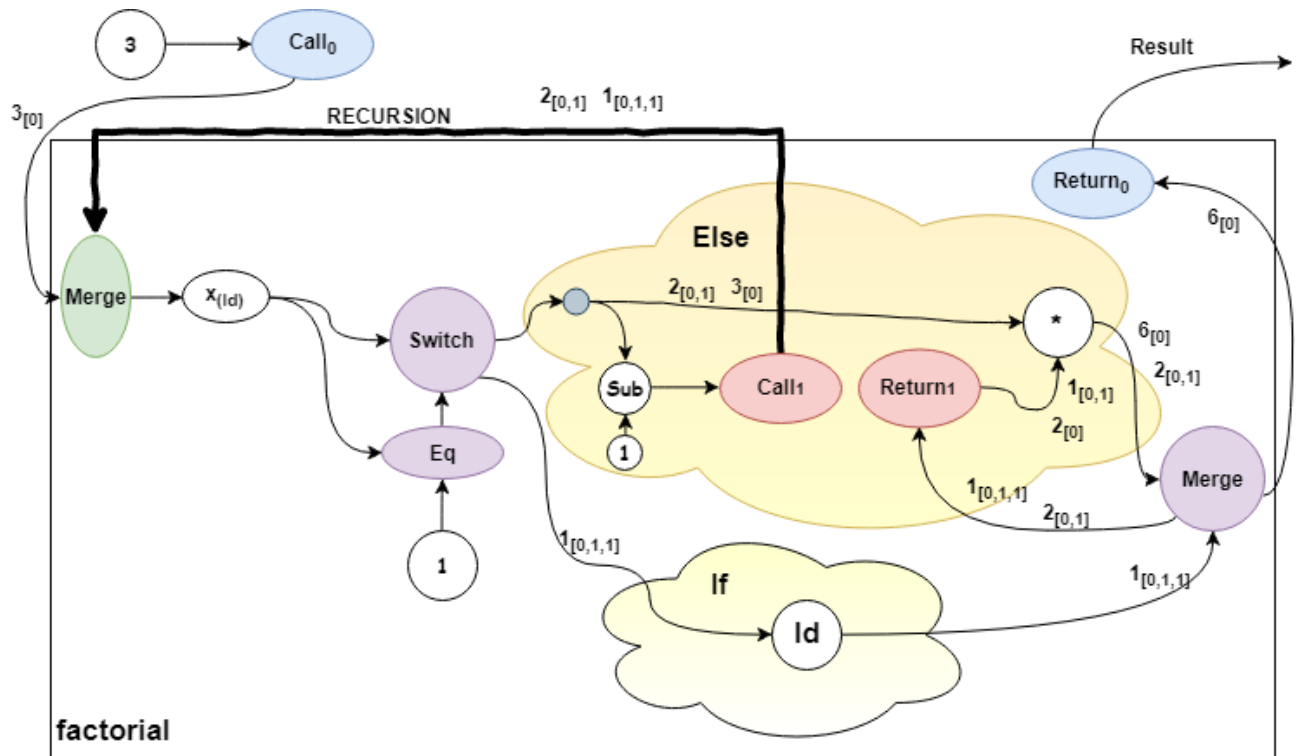


Figure 3.3: Factorial

4. IMPLEMENTATION

In the previous section, our aim was to provide a more general and abstract description of the "static" approach and to present the fundamental basis of our implementation so that it can be, theoretically, applied to all the existing dataflow frameworks.

In this section, however, we present all the technical details that concern the integration of that approach in TensorFlow's infrastructure.

Our main goal was to integrate our implementation as an additional feature to the framework. That means that we did not perform any changes to Google's already existing code but we, instead, built on top of it so that we can provide an additional method for representing and executing functions, in general. Clients can deliberately choose, via the API, whether they want to apply our approach or ignore it completely in which case Google's default way for supporting functions will prevail.

We chose to base our implementation on TensorFlow's r.1.4 release in which the definition of recursive functions via the Python API was still supported. That way we could relieve ourselves from all the dirty, interface-related details and make the most out of exploiting TensorFlow's existing infrastructure.

4.1 Graph Transformation

TensorFlow is an extremely flexible framework that can be easily extended by the users, so that they can perform their own experiments on it. People may want to implement and test their ideas either for improving the framework's performance or for research purposes, in general. For that, they are allowed to define their own optimization techniques, implement better node placement algorithms, create new additional operators or even add their own, personal execution engines (executors).

Once a dataflow graph is constructed, it passes through a number of different stages before it is ready for execution. One such stage is where it undergoes many transformation procedures that are meant to optimize its form. We managed to add a new optimization algorithm, called "function-transformation", that takes place before the graph's partitioning and changes its form, in case it detects any function-call occurrences. As already mentioned before, TensorFlow's way of dealing with functions and thus, recursion is based on the "dynamic" approach. That means that every function call is represented, inside the main graph, with a special node, whose job, when it gets executed, is to replace itself with the body of that function. The "function-transformation" algorithm manages to transform this specific representation to the one we want and described in the "Approach" section.

As shown in **Figure 4.1**, when this algorithm detects a special node calling a function, it replaces it with a new pair of "Call/Return" operators whose semantics are identical to those we described above. The algorithm must ensure that the bodies of all the callee functions are embedded inside the graph at least once (inlining). When a "Call/Return" pair takes the place of a special node, we must attach all the incoming edges of that node to the inputs of the "Call" operator. Respectively, all the outgoing edges of that node will constitute the outputs of the corresponding "Return" operator. Function transformation also adds the "Merge" operators that are needed in case we might have multiple calls to the same function, from different call-sites. It places them between the "Call" operators and the nodes that expect the arguments of the function, for reasons already mentioned.

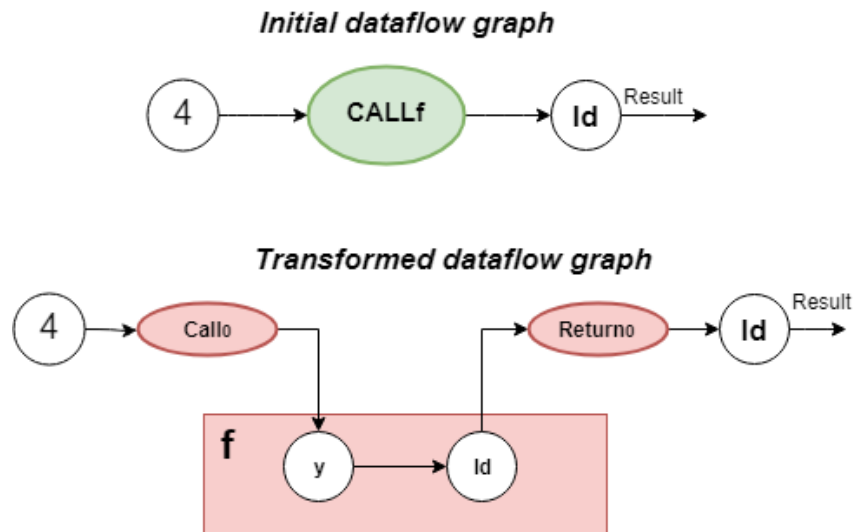


Figure 4.1: Function Transformation

Every "Call" operator, as described until now, must first gather all the function's arguments before propagating them to the nodes that need them inside the body of that function. Respectively, "Return" operator is waiting to gather all the function's returning nodes before propagating them to the outer-graph. This is, however, a limiting factor for the potential parallelism that can be achieved. Similar to what Google already does for the "Enter/Exit" operators to cure this problem, we break every "Call" and "Return" operator into multiple ones, with each one of them corresponding to a single/unique function argument/returning node. That way we can increase the number of overlapping function calls, during the execution.

In **Figure 4.2** we present the dataflow graphs of a simple program where **factorial** gets called once, as generated by TensorBoard. The graph on the left is the one generated after the "function-transformation" took place, whereas the one on the right is the initial graph generated by Google's "dynamic" approach.

TensorBoard is a visualization tool, built by Google. It has several utilities but we used it extensively for visualizing the generated TensorFlow graphs which was extremely helpful for studying them and debugging our code.

blabla a bit more maybe

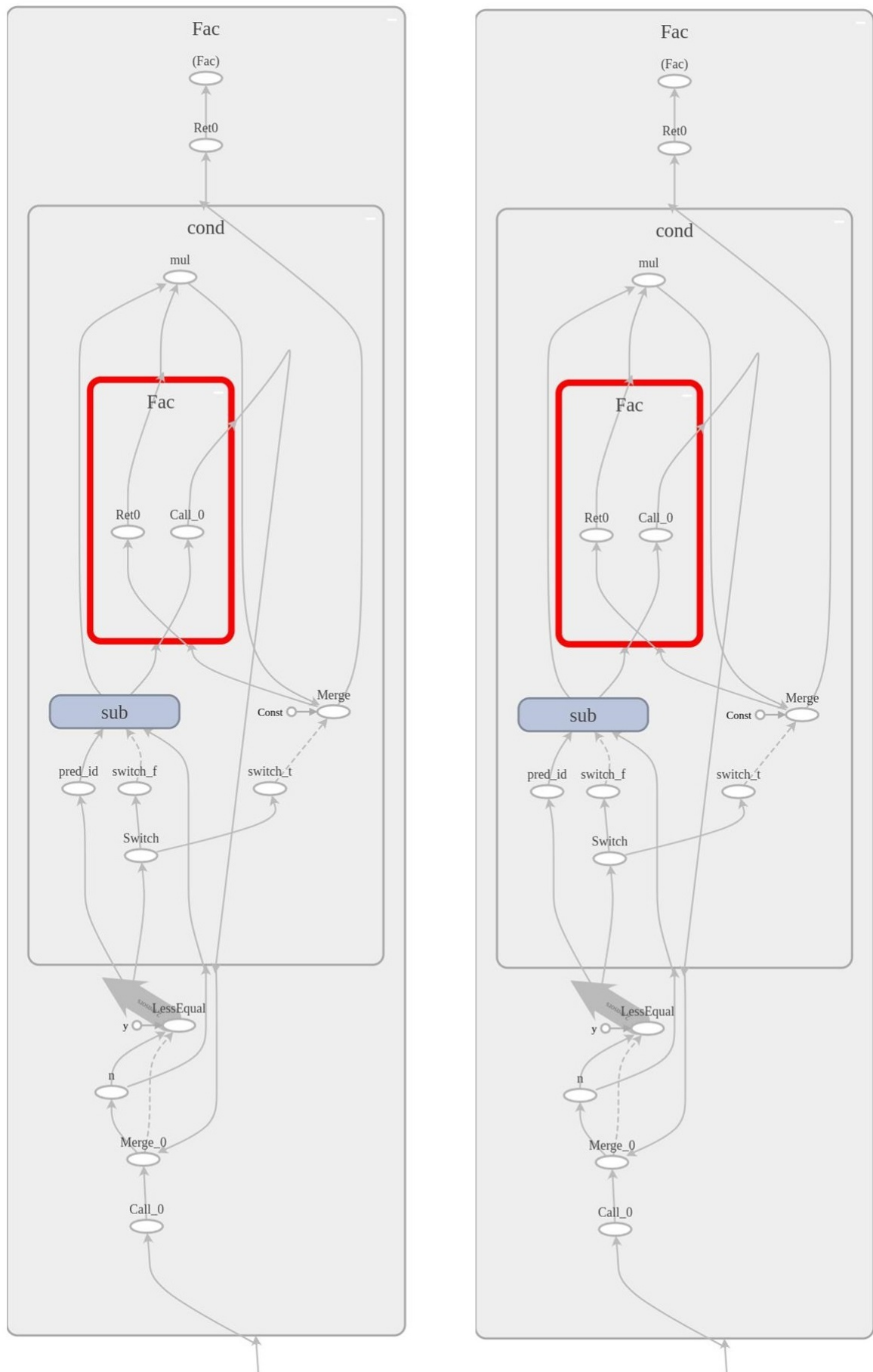


Figure 4.2: TensorFlow graphs for factorial in "static" and "dynamic" approach

4.2 Local Execution

After successfully transforming the graph, its resulting form can gracefully capture and depict in its entirety, every possible recursive mechanism defined by the client, in a way that relieves us from the need of using non-primitive, special operators with such costly and complex kernels. We, essentially, eliminate every such special node as well as the additional overhead that it imposes by replacing it with two other, less costly, primitive operators (Call/Return) that only forward their inputs and manage frames. The overhead added by the management of tags is not insignificant either, but with the appliance of the appropriate optimization techniques it can be limited significantly.

Of course, our work doesn't end with the transformation. We may have succeeded so far, in ending up with representations that favor the "static" approach but we haven't ensured, yet, that the execution engine recognizes the introduced operators and thus, understands how to properly treat the graph at runtime. Consequently, the next matter we needed to attend to was the registration of the Call/Return operators and the implementation of their kernels. For the sake of consistency, we tried to do that by constantly consulting Google's pre-existing code for iteration, as Call and Return operators do not differ from Enter and Exit that much, after all. In fact, one can say that the first two are nothing but a generalization of the second ones.

As a continuation to Google's attempt at describing their operators' semantics, we provide below, in a similar vein, the evaluation rules of our newly introduced operators:

$$\begin{aligned} Eval(Call(d, name), c) &= r, \text{ where} \\ r &= (value(d), is_dead(d), tag(d)/name) \end{aligned}$$

$$\begin{aligned} Eval(Return(d, name), c) &= r, \text{ where} \\ tag(d) &= tag1/name \\ r &= (value(d), is_dead(d), tag1) \end{aligned}$$

Note that, the evaluation of "Return" is not defined for the case where it receives data from returning nodes that have been executed under a different context.

After registering Call and Return in TensorFlow and making the appropriate adjustments to the execution engine, the semantics of these two operators are now fully conveyed to TensorFlow's infrastructure and the implementation for "local execution" is considered to be complete. From now on, any example of simple or mutual recursion can be executed via the "static" approach, too, if the client chooses so.

Something that should be noted is, that, contrary to the case of iteration, the execution of non-tail recursive functions imposes what might constitute a significant performance issue. We have no control of parallelism. That is, we cannot impose a limit to how many calls of a recursive function can execute in the machine, at the same time. It is easy to specify a maximum number of overlapping iteration "steps" because the execution of an iteration step that exceeds the limit can simply be stalled (TensorFlow does such thing for limiting parallelism). However, in cases where we are dealing with non-tail recursive functions, the execution of one function-call depends on that of another one and those dependencies between parent-children function calls go on until we reach a base case. Therefore, imposing similar parallelism limitations will most probably cause a deadlock.

4.3 Distributed Execution

Executing graphs without deploying them on clusters is not wise in terms of performance. Using dataflow systems only for 'local exeuction' not only does not exploit their immense parallelism/distribution potential but also yields slower performance results compared to other classic systems, as the overhead that they impose by trying to emulate the dataflow computational model is very significant. That is why, implementing a "dynamic control flow" approach that favors the distributed execution is extremely important.

We already saw that, the resulting graphs in "static" approach contain the bodies of any existing, callee functions. That means that, contrary to TensorFlow's approach, we are now able to include these subgraphs in the partitioning and assign the function nodes to be executed on different machines, in ways that will improve performance the most.

The partitioning of a recursive function definition, however, as well as that of any other dynamic control flow construct when we follow the "static" approach, is not that easy to perform in TensorFlow. TensorFlow's design allows the execution of each partition to make progress independently, without conforming to constraints imposed by a centralized coordinator [?]. That maximizes the potential for better performance, but makes the partitioning a challenging procedure.

Remember that, in static approach, any subgraph corresponding to the body of a while loop or a recursive function, is provided with a tagging mechanism that will create/destroy frames and produce different contexts under which those computations will take place. This mechanism is nothing but an additional set of primitive, context switching operators which are, also, subject to partitioning. A partition, however, cannot include any partially composed "mechanisms" as this would cause the incorrect generation of frames/tags, inside a machine, during execution. We have to make sure that, every partition that contains nodes that will be executed under different contexts/frames, will also contain all the necessary mechanisms for generating that frames.

In order to add support for distributed execution, in our implementation for recursion, we adopted a similar solution to the one Google already proposes for the case of iteration, yet more generalized. Google's solution keeps track of all the iterative constructs that are embedded in a graph as well as their mechanisms for generating and destroying frames. Whenever a loop is "shared" between multiple partitions, the corresponding mechanism, referred to as control-loop state machine, will be added to all of them. A simple control-loop state machine consists of an Enter, a NextIteration, a Merge and a Switch operator as shown in **Figure 4.3** where we present an example also found in the "Dynamic Control Flow in Large-Scale Machine Learning" paper [?].

However, in case of recursion, the form of a "control-recursion" state machine is slightly more complex. Here, we may have multiple conditional branches, with each one of them, leading either to a base case or a simple/mutually recursive call or a simple call to a different possibly recursive function e.t.c. What we do in order to extract the correct form of the state machines in every case, is traversing the graph and capturing all the special operators that constitute the dynamic control flow's backbone as well as the way they are connected to each other. When traversing the graph we ignore common and non context-switching operators as they will neither be included in the eventual state machine nor they will give us important information about the dynamic control flow construct's structure. We take into account the Call, Return, Switch, Merge operators and all of them, with the exception of "Return" nodes, will be added to the state machine that will eventually be copied to all the appropriate partitions. The way these operators will be connected to

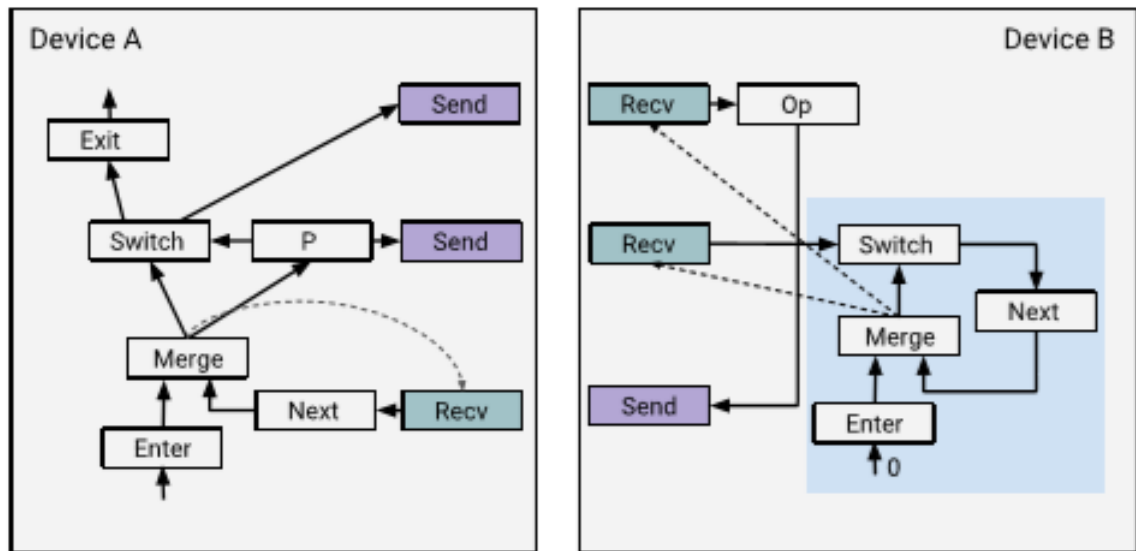


Figure 4.3: Distributed execution of a while-loop in TensorFlow

each other, inside the state machine, is inferred during the graph's traversal.

As shown in **Figure 4.3**, whenever two connected nodes reside on two different partitions, TensorFlow creates and places between them, two additional special operators that handle the transition of the data via the network. These nodes, named "Send" and "Receive", get connected to the nodes that send the data and receive it, respectively and each one of them resides on the same partition as its connected node does.

We already made a reference, in a previous section, to a phenomenon called deadness propagation. In cases where a conditional branch is not taken, the nodes that compose it are being executed anyway. The difference is that these nodes do not perform computations. They are considered to be "dead" and all the outputs they generate are propagated as "dead", too. The reason why this is happening, is because "Receive" operators do not formally have any incoming edges and that makes them qualified for immediate execution. Their computation is terminated the moment they receive data from its corresponding "Send" operators that belong in other partitions. However, there may exist "Receive" operators, that have been triggered to wait for data generated by nodes that belong in non-taken branches. Normally, those operators would block indefinitely, as they would never receive any data, but now, with deadness propagation, they will accept data tokens indicated as dead and their execution will be officially terminated.

In the case we are dealing with, in our implementation, we may have conditional branches that contain simple/mutually recursive calls. The nodes of these branches are also expected to generate dead data in case the branches are not taken, so we have to ensure somehow, that deadness will manage to get propagated through the callee recursive functions, too. This, however, constitutes an issue, as the recursive "Call" operators might cause the propagation to get stuck within an endless loop or freeze in general. We are treating this by inserting a control-edge that directly connects the "Call" operator to the corresponding "Return". So, now, "Call" manages to propagate deadness immediately through the "Return" operator.

Control edges are edges that do not transfer useful data. They are only created between two nodes in order to impose ordering constraints.

Figure with factorial's partition goes here

5. RELATED WORK

[?]

korean guys

other frameworks

6. FUTURE WORK

6.1 Higher Order Functions

[?]

6.2 Automatic Differentiation

7. EVALUATION

8. CONCLUSIONS

ABBREVIATIONS - ACRONYMS

NKUA	National and Kapodistrian University of Athens
NCSR	National Centre for Scientific Research
PC	Program Counter

BIBLIOGRAPHY