A Dynamic Pipeline Framework implemented in Haskell[⋆]

Juan Pablo Royo Sales[a], Edelmira Pasarella[a,*], Cristina Zoltan[a], Maria-Esther Vidal[b]

[a] *Computer Science Department, Universitat Politècnica de Catalunya, C/Jordi Girona, 1-3, 08034, Barcelona, Spain*

[b] *TIB/L3S Research Centre at the University of Hannover, Hannover, Germany*

Abstract

Streaming processing has given rise to new computation paradigms to provide effective and efficient data stream processing. The Dynamic Pipeline Paradigm is a computational model for stream processing suitable to solving problems where the incremental emission of results is a critical issue. In this paradigm computations are primary entities. Haskell is a pure functional programming language based on solid theoretical foundations. From a practical point of view, it has a robust set of tools for writing multithreading and parallel computations with optimal performance. In this work we tackle the problem of specifying a Dynamic Pipeline and implement it using Haskell. We first conduct and empirically assess a proof of concept to evaluate the suitability of (parallel) Haskell to compute and emit results incrementally. In particular, an *ad hoc* Dynamic Pipeline solution for computing weakly connected components incrementally is implemented. The results of the experiments are competitive with the baseline solution available in a graph algorithm Haskell library. Once established Haskell is suitable for implementing a Dynamic Pipeline Framework, it is implemented. Then, we conduct, analyze and report experiments to evaluate the performance of using this framework to compute weakly connected components incrementally w.r.t. the *ad hoc* Dynamic Pipeline solution. To assess the continuous generation of results, we measure the Diefficiency metrics, i.e., the continuous efficiency of the implementation of an algorithm for generating incremental results. The observed results satisfy our expectations. Moreover, they put in perspective the suitability of the Dynamic Pipeline Framework for implementing graph stream processing problems where it is critical that results are produced continuously.

Keywords: Stream Processing Frameworks, Dynamic Pipeline, Parallelism, Concurrency, Haskell

Contents

[⋆]This work was partially supported by MCIN/ AEI /10.13039/501100011033 [grant number PID2020-112581GB-C21].

[*]Corresponding author

Email addresses: `juan.pablo.royo@estudiantat.upc.edu` (Juan Pablo Royo Sales), `edelmira@cs.upc.edu` (Edelmira Pasarella), `zoltan@cs.upc.edu` (Cristina Zoltan), `maria.vidal@tib.eu` (Maria-Esther Vidal)

1. Introduction

Effective streaming processing of large amounts of data has been studied for several years [6, 14] as a critical factor providing fast and incremental results in big data algorithmic problems. Parallel techniques that exploit computational power as much as possible represent one of the most explored techniques, regardless of the approach. In that regard, the Dynamic Pipeline Paradigm (DPP) [26] has lately emerged as one of the models that exploit data streaming processing using a dynamic pipeline parallelism approach [14]. This computational model has been designed with a functional focus. The main components of this paradigm are functional stages or pipes that dynamically enlarge and shrink depending on incoming data.

One of the biggest challenges of implementing a Dynamic Pipeline Framework (DPF) is to find a proper set of tools and a programming language which can take advantage of both of its primary aspects: i) *fast parallel* processing and ii) *strong theoretical* foundations that manage computations as first-class citizens. Haskell is a statically typed pure functional language that has been designed and evolved on strong theoretical foundations where computations are primary entities. At the same time, Haskell makes available a powerful set of tools for writing multithreading and parallel programs with optimal performance [16, 19].

In the context of this research, we first assessed the suitability of Haskell to implement DPP solutions to stream processing problems. To be concrete, we conduct a proof of concept implementing a

Dynamic Pipeline in Haskell for solving a particular and very relevant problem as the computation/enumeration of the Weakly Connected Components of a graph. In particular, the main objective of our proof of concept is to study the critical features required in Haskell for a DPF implementation, the real possibilities of emitting incrementally results, and the performance of such kinds of implementations. Indeed, we explore the basis of an implementation of a DPF in a pure (parallel) functional language as Haskell. This is, we determine the particular features (i.e., versions and libraries) that will allow for an efficient implementation of a DPF. Moreover, we conduct an empirical evaluation to analyze the performance of the Dynamic Pipeline implemented in Haskell for enumerating WCC. To assess the incremental delivery of results, we measure the Diefficiency metrics [1], i.e., the continuous efficiency of the implementation of an algorithm for generating incremental results. Since continuous performance results are encouraging and the programming basis is clearly stated, we develop a general Dynamic Pipeline Framework written in Haskell Programming Language which allows for implementing algorithms under the Dynamic Pipeline Paradigm approach. Then, we conduct, analyze, and report experiments to measure the performance of using this framework to compute weakly connected components incrementally w.r.t. the *ad hoc* Dynamic Pipeline solution. Obtained results satisfy our expectations and encourage us to keep using the Dynamic Pipeline Framework for solving some families of graph stream processing problems where is critical not to have to wait until the whole results are emitted. Furthermore, the experiments give us insights about the characteristics these family of problems should meet.

Problem Research and Objective: The main objective of this work is to design and implement a Dynamic Pipeline Framework using Haskell as programming language. Through a particular and very relevant problem as the computation of the Weak Connected Components (WCC) of a graph, we study the critical features required in Haskell for a DPF implementation and set the basis of an implementation of a DPF in Haskell. This is, we determine the particular features (i.e., versions and libraries) of this language that will allow for an efficient implementation of the DPF.

Contributions: This paper is an extension to our work previously reported by Royo et al. [24]; we present a solution (a.k.a. $\mathsf{DP}_{\mathsf{WCC}}$) to the problem of enumerating the weakly connected components of a graph as a dynamic pipeline implemented in Haskell. We also empirically show the behavior of $\mathsf{DP}_{\mathsf{WCC}}$ concerning a solution for enumerating the weakly connected components using Haskell libraries; we name this solution Blocking WCC in Haskell `containers` ($\mathsf{BLH}_{\mathsf{WCC}}$). Additionally, we present the following novel contributions in this paper: i) the Dynamic Pipeline Framework (DPF-Haskell) implemented in Haskell. ii) A program for enumerating weakly connected component implemented on top of the Dynamic Pipeline Framework (DPF-Haskell). iii) An empirical evaluation, comparing the performance of $\mathsf{DP}_{\mathsf{WCC}}$ in DPF-Haskell ($\mathsf{DPFH}_{\mathsf{WCC}}$) with respect to $\mathsf{BLH}_{\mathsf{WCC}}$ and $\mathsf{DP}_{\mathsf{WCC}}$ baseline in Haskell ($\mathsf{DP}_{\mathsf{WCC}}$). The continuous performance of these implementations is measured in terms of the diefficiency metric dieft [27]. The results of this study suggests that Haskell is a suitable language for implementing DPP and the programming basis for implementing DP solutions is settled. Lastly, it has been shown that implementing $\mathsf{DPFH}_{\mathsf{WCC}}$ performs well with respect to $\mathsf{DP}_{\mathsf{WCC}}$. Hence, we envision that the overhead of DPF-Haskell does not degrade the performance of dynamic pipelines implemented on the framework.

The rest of this paper is organized as follows. The next section presents the basic notions used through this work. In subsection 3.2 the proof of concept is analyzed. To be concrete, in this section an algorithm for enumerating WCC using DPP and its implementation using parallel Haskell are introduced. Additionally,the experiments conducted to assess how Haskell supports the dynamic pipeline implementation for enumerating WCC are described and their results reported. In section 4 the system architecture of the Haskell Dynamic Pipeline Framework and the most relevant details of its implementation are deeply explained. In particular, all the Haskell data types and language techniques used in the implementation are detailed. The justification of the used external libraries for the runtime system are presented at the end of this section. In section 5 the implementation of the $\mathsf{DPFH}_{\mathsf{WCC}}$ is presented. In addition, the empirical evaluation that we have conducted to analyze the performance of this implementation is analyzed. In section 2 we present the related work and, finally, conclusions and further work are presented in section 7.

2. Related Work

Streaming in Haskell Language. Streaming computational models have been implemented in Haskell Programming Language during the last 10 years. One of the first libraries in the ecosystem was `conduit`[1] in 2011. After that, several efforts on improving streaming processing on the language has been made not only at abstraction level for the user but as well as performance execution improvements like `pipes` [2] and `streamly`[3] lately. Moreover, there is an empirical comparison between those three, where a benchmark analysis has been conducted[4].

Although most of those libraries offer the ability to implement DAP and PP, none of them provide clear abstractions to create DPP models because the setup of the stages should be provided beforehand. In the context of this work, we have done a proof of concept at the beginning, but it was not possible to adapt any of those libraries to implement properly DPP. The closest we have been to implement DPP with some of those libraries was when we explored `streamly`. In this case, there is a `foldrS` combinator that could have been proper for the purpose of generating a dynamic pipeline of stages based on the data flow. However, it was not possible to manipulate the channels between the stages to control the flow of the data. It is important to remark that, even though, the library `streamly` implements channels, they are hidden from the end-user, and there is not a clear way to manipulate them.

To the best of our knowledge, no similar library under the DPP approach has been written in Haskell Programming Language. One important motivation to develop our own framework is that we not only want to satisfy our research needs but, as a novel contribution, we aim at providing a DPF to the Haskell community as well. We hope this contribution encourages and helps writing algorithms under the Dynamic Pipeline Paradigm. Another kind of streaming implementation in Haskell is described in [16]. In that work, the author describes how to encode pipeline parallelism with `Par` *Monad*. Although this could have been a suitable alternative for implementing DP, the parallelization level used by `Par` *Monad* is sparks [18]. As we have explained in section **??**, we do not require to reach that level of parallelization in our current model.

In regards to other DP language implementations, a significant contribution on [21] has been done, where a DP implementation in Go Programming Language (Go) for counting triangles of graphs is compared against MapReduce. Those experiment results have shown how DP in Go improves the performance in terms of execution time and memory depending on the graph topology. It would be interesting and a matter of future work, to compare different language implementations of DPs, taking into consideration those promising results and the ones presented in this article.

[1]`https://hackage.haskell.org/package/conduit`
[2]`https://hackage.haskell.org/package/pipes`
[3]`https://hackage.haskell.org/package/streamly`
[4]`https://github.com/composewell/streaming-benchmarks`

3. Preliminaries

Before moving forward with the core of the research, we describe the fundamental concepts that support the different parts of our study. That is, stream processing in general, Dynamic Pipeline Paradigm, streaming processing in the context of Haskell and the metrics we use in the conducted experiments.

3.1. Basic Concepts

3.1.1. Streaming Processing

The development of streaming processing techniques have potentiated areas as massive data processing for data mining algorithms, big data analysis, IoT applications, etc. Data Streaming (DS) has been studied using different approaches [6, 14] allowing to process a large amount of data efficiently with an intensive level of parallelization. We can distinguished two different parallelization streaming computational models: Data Parallelism (DAP) and Pipeline Parallelism (PP).

Data Parallelism (DAP). The data is split and processed in parallel and, the computations that perform some action over that subset of data do not have any dependency with other parallel computation. A common model that has been proved successful over the last decade is MapReduce (MR) [4]. Different frameworks or tools like Hadoop [5], Spark [6], etc., support this computational model efficiently. One of the main advantages of this kind of model is the ability to implement stateless algorithms. Data can be split and treated in different threads or processors without the need for contextual information. On the other hand, when there is a need to be aware of the context, parallelization is penalized, each computational step should be fully calculated before proceeding with the others. For example, this is the case of `reduce` operation on many of the above-mentioned frameworks or tools.

Pipeline Parallelism (PP). It breaks the computation in a series of sequential stage, where each stage takes the result of the previous stage as an input and downstream its results to the next. Each *pipeline Stage* is parallelized and, it could potentially exist one stage per data item of the stream. The communication between stages takes place through some means, typically channels. One of the main advantages of this model is that the stages are non-blocking, meaning that there is no need to wait to process all data to run the next stage. This kind of paradigm enables computational algorithms that can generate incremental results, preventing the user waits until the end of the whole data stream processing to get a result. On the other hand, the disadvantage over DAP is that although pipeline stages are parallelized, some intensive computation in one stage might delay processing the next stage because of its sequential dependency nature. Therefore, the user must be sure each stage runs extremely fast computations on it.

The nature of our problem requires that results are output incrementally, i.e., the weakly connected components are generated incrementally. Additionally, data need to be aware of the context to compute the weakly connected components. Considering the stream processing models presented above, we have chosen PP. We think it is the model that better fits the requirements of the enumerating weak connected components incrementally. We are going to see in the next section what is the specific PP computation model used for that purpose.

[5]`https://hadoop.apache.org/`
[6]`https://spark.apache.org/`

3.1.2. Diefficiency Metrics

In this work, we use two important metrics to measure the diefficiency, i.e., continuous efficiency of a program to generate incremental results. The metrics to measure diefficiency are `dief@t` and `dief@k` [1]. The metric `dief@t` measures the continuous efficiency during the first t time units of execution regarding the results generated by the program. The higher value of the `dief@t` metric, the better the continuous behavior. The metric `dief@k` measures the continuous efficiency while producing the first k answers. The lower the value of the `dief@k` metric, the better the continuous behavior. Both metrics have been measured using `diefpy` Tool (`diefpy`) [27] and traces obtained by the execution of the experiment scenarios. Additionally, `diefpy` generates two different kinds of plots w.r.t. an experimental scenario: A bi-dimensional plot and a radial plot. In the bi-dimensional plot, the x-axis represents the time when answers are generated and the y-axis represents the number of generated answers. Points (x, y) are taken from traces. The radial plot contains the visual comparison of `dief@t` the metric with respects to other non-continuos metrics, such as **i)** Completeness (Comp) which is the total number of produced answers. **ii)** Time for the first tuple (TFFT) which measure the elapsed time spent to produce the first answer. **iii)** Execution Time (ET) which measures the elapsed time spent to complete the execution of a query. **iv)** Throughput (T) which measure the number of total answers produced after evaluating a query divided by its execution time ET .

3.1.3. Dynamic Pipeline Paradigm

The *Dynamic Pipeline Paradigm* (DPP) [26] is a PP computational model based on a one-dimensional and unidirectional chain of stages connected by means of channels synchronized by data availability. This chain of stages is a computational structure called *Dynamic Pipeline* (DP). A DP stretches and shrinks depending on the spawning and the lifetime of its stages, respectively. Modeling an algorithmic solution as a DP corresponds to define a dynamic computational structure in terms of four kinds of stages: *Source* (Sr), *Generator* (G), *Sink* (Sk) and *Filter* (F) stages. In particular, the specific behavior of each stage to solve a particular problem must be defined as well as the number and the type of channels connecting them. Channels are unidirectional according to the flow of the data. The *Generator* stage is in charge of spawning *Filter* stage instances. This particular behavior of the *Generator* gives the elastic capacity to DPs. *Filter* stage instances are stateful operators in the sense described in [23]. This is, *Filter* instances have a state. The deployment of a DP consists in setting up the initial configuration depicted in Figure 1. The activation of a DP starts

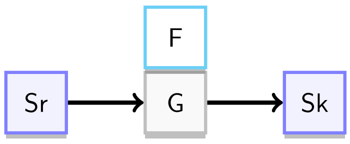


Figure 1: An initial DP consists of three stages: Sr, G together its filter parameter F, and Sk. These stages are connected through its channels –represented by right arrows– as shown in this figure.

when a stream of data items arrives at the initial configuration of the DP. In particular, when a data stream arrives to the *Source* stage. During the execution, the *Generator* stage spawns *Filter* stage instances according to incoming data and the *Generator* defined behavior. This evolution is illustrated in Figure 2. If the data stream is bounded, the computation finishes when the lifetime of all the stages of DP has finished. Otherwise, if the stream data is unbounded, the DP remains active and incremental results are output.

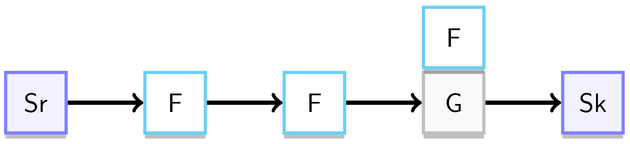


Figure 2: Evolution of a DP. After creating some filter instances (shadow Filter squares) of the filter parameter (light Filter square) in the Generator, the DP has stretched.

3.2. Weakly Connected Components of a Graph

One of the biggest challenges of implementing a Dynamic Pipeline is to find a programming language with a proper set of tools supporting both of the primary features of the DPP: i) *parallel* processing and ii) *strong theoretical* foundations to manage computations as first-class citizens. In this section, we introduce the DP for enumerating weakly connected components using, Haskell as well as the results obtained in the empirical evaluation of its implementation. More details of the empirical evaluation can be found in [24].

3.3. $\mathsf{DP_{WCC}}$ *Algorithm*

Let us consider the problem of computing/enumerating the (weak) connected components of a graph G using DPP. A connected component of a graph is a subgraph in which any two vertices are connected by paths. Thus, finding connected components of an undirected graph implies obtaining the minimal partition of the set of nodes induced by the relationship *connected*, i.e., there is a path between each pair of nodes. An example of that graph can be seen in Figure 3. The input of the Dynamic Pipeline for computing the WCC of a graph, $\mathsf{DP_{WCC}}$, is a sequence of edges ending with eof. In the source graph, there are neither isolated vertices nor loops. The connected components are output as soon as they are computed, i.e., they are produced incrementally. Roughly speaking, the idea of the algorithm is that the weakly connected components are built in two phases. In the first phase, filter instance stages receive the edges of the input graph and create sets of connected vertices. During the second phase, these filter instances construct maximal subsets of connected vertices, i.e., the vertices corresponding to (weakly) connected components. $\mathsf{DP_{WCC}}$ is defined in terms of the behavior of its four kinds of stages: *Source* ($\mathsf{Sr_{WCC}}$), *Generator* ($\mathsf{G_{WCC}}$), *Sink* ($\mathsf{Sk_{WCC}}$), and *Filter*($\mathsf{F_{WCC}}$) stages. Additionally, the channels connecting these stages must be defined. In $\mathsf{DP_{WCC}}$, stages are connected linearly and unidirectionally through the channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$. Channel $\mathsf{IC_E}$ carries edges, while channel $\mathsf{IC_{set(V)}}$ conveys sets of connected vertices. Both channels end by the eof mark. The behavior of $\mathsf{F_{WCC}}$ is given by a sequence of two actors (scripts). Each actor corresponds to a phase of the algorithm. In what follows, we denote these actors by $\mathsf{actor_1}$ and $\mathsf{actor_2}$, respectively. The script $\mathsf{actor_1}$ keeps a set of connected vertices (CV) in the state of the $\mathsf{F_{WCC}}$ instance. When an edge e arrives, if an endpoint of e is present in the state, then the other endpoint of e is added to CV. Edges without incident endpoints are passed to the next stage. When eof arrives at channel $\mathsf{IC_E}$, it is passed to the next stage, and the script $\mathsf{actor_2}$ starts its execution. If the script, $\mathsf{actor_2}$ receives a set of connected vertices CV in $\mathsf{IC_{set(V)}}$, it determines if the intersection between CV and the nodes in its state is not empty. If so, it adds the nodes in CV to its state. Otherwise, the CV is passed to the next stage. Whenever eof is received, $\mathsf{actor_2}$ passes–through $\mathsf{IC_{set(V)}}$– the set of vertices in its state and the eof mark to the next stage; then, it dies. The behavior of $\mathsf{Sr_{WCC}}$ corresponds to the identity transformation over the data stream of edges. As edges arrive, they are passed through $\mathsf{IC_E}$ to the next stage. When receiving eof on $\mathsf{IC_E}$, this mark is put on both channels. Then, $\mathsf{Sr_{WCC}}$ dies.

Let us describe this behavior with the example of the graph shown in Figure 3.

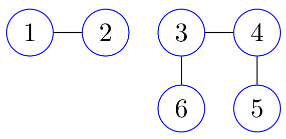


Figure 3: Example of a graph with two weakly connected components: $\{1, 2\}$ and $\{3, 4, 5, 6\}$

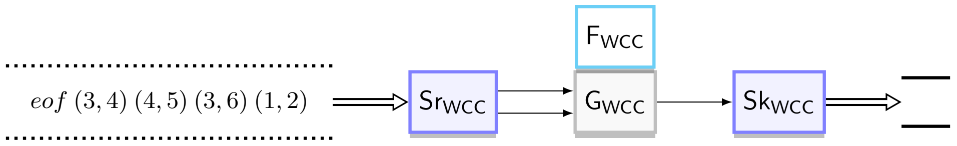
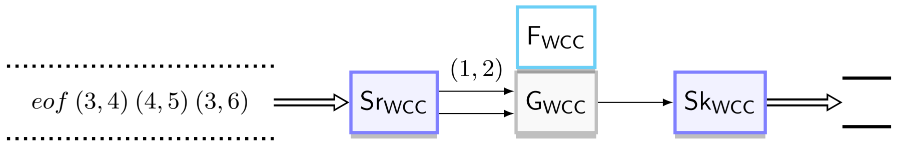
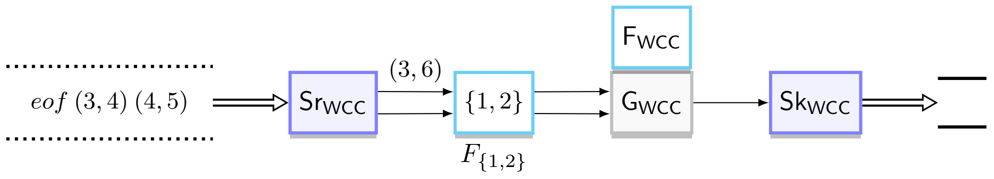


Figure 4: $\mathsf{DP_{WCC}}$ Initial setup. Stages Source, Generator, and Sink are represented by the squares labeled by $\mathsf{Sr_{WCC}}$, $\mathsf{G_{WCC}}$ and $\mathsf{Sk_{WCC}}$, respectively. The square $\mathsf{F_{WCC}}$ corresponding to the Filter stage template is the parameter of $\mathsf{G_{WCC}}$. Arrows $\rightrightarrows$ between represents the connection of stages through two channels, $\mathsf{IC_E}$, and $\mathsf{IC_{set(V)}}$. The arrow $\rightarrow$ represents the channel $\mathsf{IC_{set(V)}}$ connecting the stages $\mathsf{G_{WCC}}$ and $\mathsf{Sk_{WCC}}$. The arrow $\Longrightarrow$ stands for I/O data flow. Finally, the input stream comes between the dotted lines on the left and the WCC computed incrementally will be placed between the solid lines on the right.

Figure 4 depicts the initial configuration of $\mathsf{DP_{WCC}}$. The interaction of $\mathsf{DP_{WCC}}$ with the "external" world is done through the stages $\mathsf{Sr_{WCC}}$ and $\mathsf{Sk_{WCC}}$. Indeed, once activated the initial $\mathsf{DP_{WCC}}$, the input stream – consisting of a sequence containing all the edges in the graph in Figure 3 – feeds $\mathsf{Sr_{WCC}}$ while $\mathsf{Sk_{WCC}}$ emits incrementally the resulting weakly connected components. In what follows Figure 5, Figure 6, Figure 7, Figure 8 and Figure 9 depict the evolution of the $\mathsf{DP_{WCC}}$.



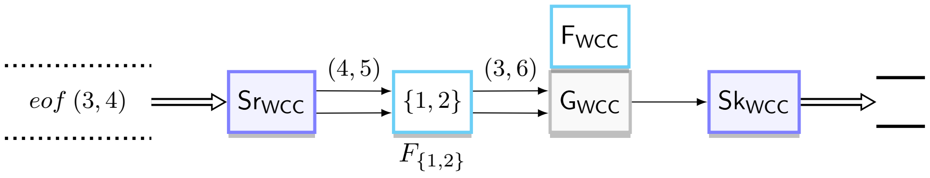
(a) The edge $(1, 2)$ is arriving to $\mathsf{G_{WCC}}$.



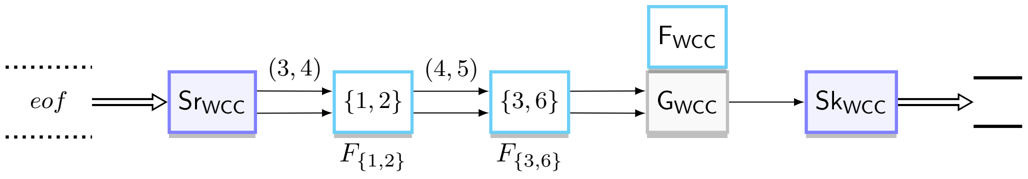
(b) When the edge $(1, 2)$ arrives to $\mathsf{G_{WCC}}$, it spawns a new instance of $\mathsf{F_{WCC}}$ before $\mathsf{G_{WCC}}$. Filter instance $F_{\{1,2\}}$ is connected to $\mathsf{G_{WCC}}$ through channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$. The state of the new filter instance $F_{\{1,2\}}$ is initialized with the set of vertices $\{1, 2\}$. The edge $(3, 6)$ arrives to the new filter instance $F_{\{1,2\}}$.

Figure 5: Evolution of the $\mathsf{DP_{WCC}}$: First state

It is important to highlight that during the states shown in Figure 5a, Figure 5b, Figure 6a, Figure 6b and Figure 7a the only actor executed in any filter instance is $\mathsf{actor_1}$ (constructing sets of connected

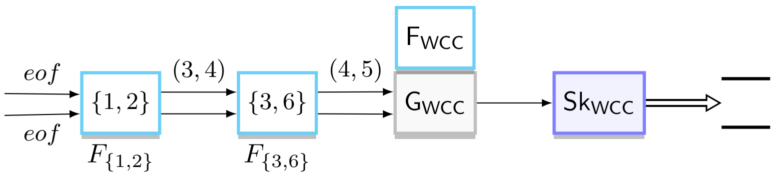


(a) None of the vertices in the edge $(3,6)$ is in the set of vertices $\{1,2\}$ in the state of $F_{\{1,2\}}$, hence it is passed through $\mathsf{IC_E}$ to $\mathsf{G_{WCC}}$.

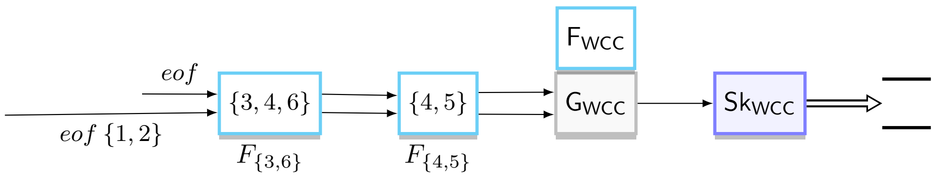


(b) When the edge $(3,6)$ arrives to $\mathsf{G_{WCC}}$, it spawns the filter instance $F_{\{3,6\}}$ between $F_{\{1,2\}}$ and $\mathsf{G_{WCC}}$. Filter instance $F_{\{1,2\}}$ is connected to the new filter instance $F_{\{3,6\}}$ and this one is connected to $\mathsf{G_{WCC}}$ through channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$. The state of the new filter instance $F_{\{3,6\}}$ is initialized with the set of vertices $\{3,6\}$. The edge $(3,4)$ arrives to $F_{\{1,2\}}$ and $\mathsf{Sr_{WCC}}$ is fed with the mark eof. Edges $(3,4)$ and $(4,5)$ remain passing through $\mathsf{IC_E}$.

Figure 6: Evolution of the $\mathsf{DP_{WCC}}$: Second state



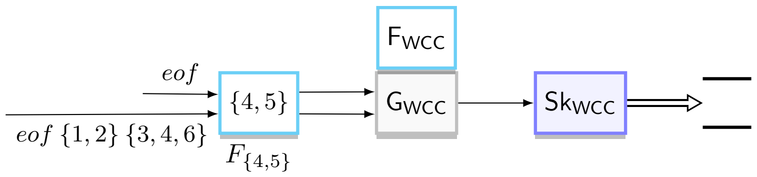
(a) $\mathsf{Sr_{WCC}}$ fed both, $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$, channels with the mark eof received from the input stream in previous state and then, it died. The edge $(4,5)$ is arriving to $\mathsf{G_{WCC}}$ and the edge $(3,4)$ is arriving to $F_{\{3,6\}}$.



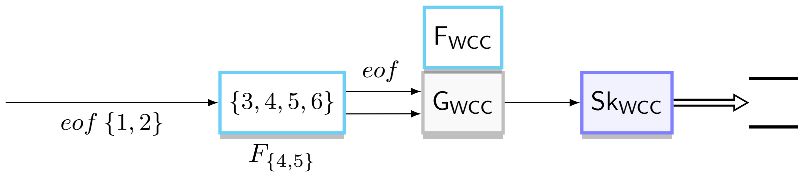
(b) When the edge $(4,5)$ arrives to $\mathsf{G_{WCC}}$, it spawns the filter instance $F_{\{4,5\}}$ between $F_{\{3,6\}}$ and $\mathsf{G_{WCC}}$. Filter instance $F_{\{3,6\}}$ is connected to the new filter instance $F_{\{4,5\}}$ and this one is connected to $\mathsf{G_{WCC}}$ through channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$. Since the edge $(3,4)$ arrived to $F_{\{3,6\}}$ at the same time and vertex 3 belongs to the set of connected vertices of the filter $F_{\{3,6\}}$, the vertex 4 is added to the state of $F_{\{3,6\}}$. Now, the state of $F_{\{3,6\}}$ is the connected set of vertices $\{3,4,6\}$. When the mark eof arrives to the first filter instance, $F_{\{1,2\}}$, through $\mathsf{IC_{set(V)}}$, this stage passes its partial set of connected vertices, $\{1,2\}$, through $\mathsf{IC_{set(V)}}$ and dies. This action will activate $\mathsf{actor_2}$ in next filter instances to start building maximal connected components. In this example, the state in $F_{\{3,6\}}$, $\{3,4,6\}$, and the arriving set $\{1,2\}$ do not intersect and, hence, both sets of vertices, $\{1,2\}$ and $\{3,4,6\}$ will be passed to the next filter instance through $\mathsf{IC_{set(V)}}$.

Figure 7: Evolution of the $\mathsf{DP_{WCC}}$: Third state

vertices). Afterwards, although $\mathsf{actor_1}$ can continue being executed in some filter instances, there are some instances that start executing $\mathsf{actor_2}$ (constructing sets of maximal connected vertices). This is shown from Figure 7a to Figure 9a.

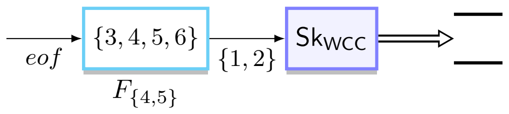


(a) The set of connected vertices $\{3,4,6\}$ is arriving to $F_{\{4,5\}}$. The mark eof continues passing to next stages through the channel $\mathsf{IC_E}$.

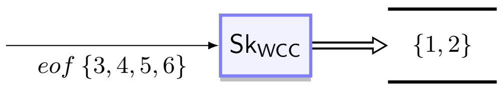


(b) Since the intersection of the set of connected vertices $\{3,4,6\}$ arrived to $F_{\{4,5\}}$ and its state is not empty, this state is enlarged to be $\{3,4,5,6\}$. The set of connected vertices $\{1,2\}$ is arriving to $F_{\{4,5\}}$

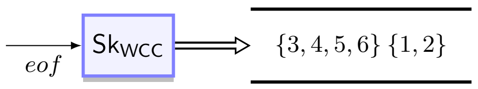
Figure 8: Evolution of the $\mathsf{DP_{WCC}}$: Fourth state



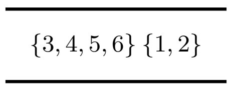
(a) $F_{\{4,5\}}$ has passed the set of connected vertices $\{1,2\}$ and it is arriving to $\mathsf{Sk_{WCC}}$. The mark eof is arriving to $F_{\{4,5\}}$ through $\mathsf{IC_{set(V)}}$.



(b) Since the mark eof arrived to $F_{\{4,5\}}$ through $\mathsf{IC_{set(V)}}$, it passes its state, the set $\{3,4,5,6\}$ through $\mathsf{IC_{set(V)}}$ to next stages and died. The set of connected vertices $\{1,2\}$ arrived to $\mathsf{Sk_{WCC}}$ and this implies that $\{1,2\}$ is a maximal set of connected vertices, i.e. a connected component of the input graph. Hence, $\mathsf{Sk_{WCC}}$ output this first weakly connected component.



(c) Finally, the set of connected vertices $\{3,4,5,6\}$ arrived to $\mathsf{Sk_{WCC}}$ and was output as a new weakly connected component. Besides, the mark eof also arrived to $\mathsf{Sk_{WCC}}$ through $\mathsf{IC_{set(V)}}$ and thus, it dies.



(d) The weakly connected component of in the graph Figure 3 such as they have been emitted by $\mathsf{DP_{WCC}}$.

Figure 9: Last states in the evolution of the $\mathsf{DP_{WCC}}$

4. Dynamic Pipeline Framework in Haskell

The design and implementation of Haskell Dynamic Pipeline Framework (DPF-Haskell) is the core of the present work. A general Dynamic Pipeline Framework written in Haskell Programming Language which allows for implementing algorithms under the Dynamic Pipeline Paradigm approach. In what follows, we present the design and the most relevant implementation details of DPF-Haskell[7].

4.1. Framework Design

A suitable framework should provide users with the right level of abstraction that removes and hides underlying complexity, allowing developers to focus on the problem to be solved. There are several design approaches to implement a framework: i) *Configuration Based* where users only focuses on completing a specific configuration either on a file or a database or both. This configuration is provided to the framework's runtime system in order to execute the program. An example of this could be WordPress[8], ii) *Convention over Configuration (CoC)* where users write code and definition following certain patterns in naming or source code location. Using the source code and content-specific information, the framework interprets the execution flow that needs to be executed. This technique has been deeply explored in the last 10 years. One of the first framework introducing this design paradigm was Ruby on Rails[9]. Other examples are Spring Boot[10] and Cake PHP[11], iii) *Application Programming Interface (API)* where the framework or library provides functions or interfaces. Users need to compose these functions to achieve the desired results. This has been a traditional design approach for building libraries or frameworks, and finally iv) *Domain-specific Language (DSL)* [5] where the framework or library provides a domain-oriented language. Solutions are written in terms of this DSL language. An example of this type of design is Hibernate Query Language[12] .

There exists two types of Domain-specific Languages [13]: External Domain-specific Language (DSL) and Embedded Domain-specific Language (EDSL) [10]. The purpose of DSL, is to create a completely new language and its interpreter. As their name indicates DSLs are not general-purpose languages. EDSL are syntactically embedded in the host language of the library, and users write in that host language, but restricted by the EDSL abstractions.

System Architecture. DPF-Haskell follows a EDSL approach taking advantage of the strong type Haskell system providing correctness at type-level [9]. The framework has three components:

DSL Component. This component allows users to specify the structure of the dynamic pipeline to be implemented on the DPF-Haskell. To this purpose, a syntactically embedded in the host language of the library has been defined (DP-EDSL). Additionally, users must provide a type-level specification about the channels communicating stages of the pipeline, as well as the data types these channels carry is provided.

[7]The DPF-Haskell is available in *Hackage: The Haskell Package Repository*, `https://hackage.haskell.org/package/dynamic-pipeline`. The source code is open and can be found on this Github Repository`https://github.com/jproyo/dynamic-pipeline`.

[8]`https://wordpress.com/`

[9]`https://rubyonrails.org/`

[10]`https://spring.io/projects/spring-boot`

[11]`https://cakephp.org/`

[12]`https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html`

Definition 1. The grammar of the Dynamic Pipeline Domain Specific Language (DP-EDSL) is formally defined as the 4-tuple $G_{dsl} = (N, \Sigma, DB, P)$ where

$$N = \{DP, S_r, S_k, G, F_b, CH, CH_s\},$$
$$\Sigma = \{\texttt{Source}, \texttt{Generator}, \texttt{Sink}, \texttt{FeedbackChannel}, \texttt{Type}, \texttt{Eof}, \texttt{:=>}, \texttt{:<+>}\},$$

$$\begin{aligned} P = \{ & \\ DP &\to S_r \texttt{ :=> } G \texttt{ :=> } S_k \mid S_r \texttt{ :=> } G \texttt{ :=> } F_b \texttt{ :=> } S_k, \\ S_r &\to \texttt{Source } CH_s, \\ G &\to \texttt{Generator } CH_s, \\ S_k &\to \texttt{Sink}, \\ F_b &\to \texttt{FeedbackChannel}CH, \\ CH_s &\to \texttt{Channel } CH, \\ CH &\to \texttt{Type :<+> } CH \mid \texttt{Eof}\} \end{aligned}$$

IDL Component. This component is the Interpreter of the DP-EDSL specification. Users interact with it to define the functions corresponding to each stage of the dynamic pipeline according to the structure of the dynamic pipeline provided in the EDSL language.

RS component. This is the Run System component. It is fed with the dynamic pipeline definition and the functions implementations to execute the program.

Figure 10 depicts the different the components of the DPF-Haskell and their organization as layers of the system.

4.2. Implementation

In this section, we describe the implementation details of each system layer: DSL, IDL and RS. For encoding G_{dsl} on the Haskell, we use an *Index type* [8] to keep track, at type-level, of the extra information required by the DPP definition such as channels and data types the channels carry.

```haskell
data Source (a :: Type)
data Generator (a :: Type)
data Sink
data Eof
data Channel (a :: Type)
data FeedbackChannel (a :: Type)
```

Source Code 4.1: This code is showing most of the data types that represent the same terminal symbols Σ in G_{dsl}. These types indexed by another kind `Type`, allows us to store information at type-level needed for interpret the DP-EDSL

In Source Code 4.1, there is an *Index type* for each element of Σ encoded in Haskell Types. The highlighted lines in Source Code 4.1 shows the terminal symbols Σ that are not indexed, because neither `Sink` nor `Eof` are carrying extra type-level information. In the case of `Sink`, since it is the last stage that does not connect further with any other stage, we do not need to indicate any channel

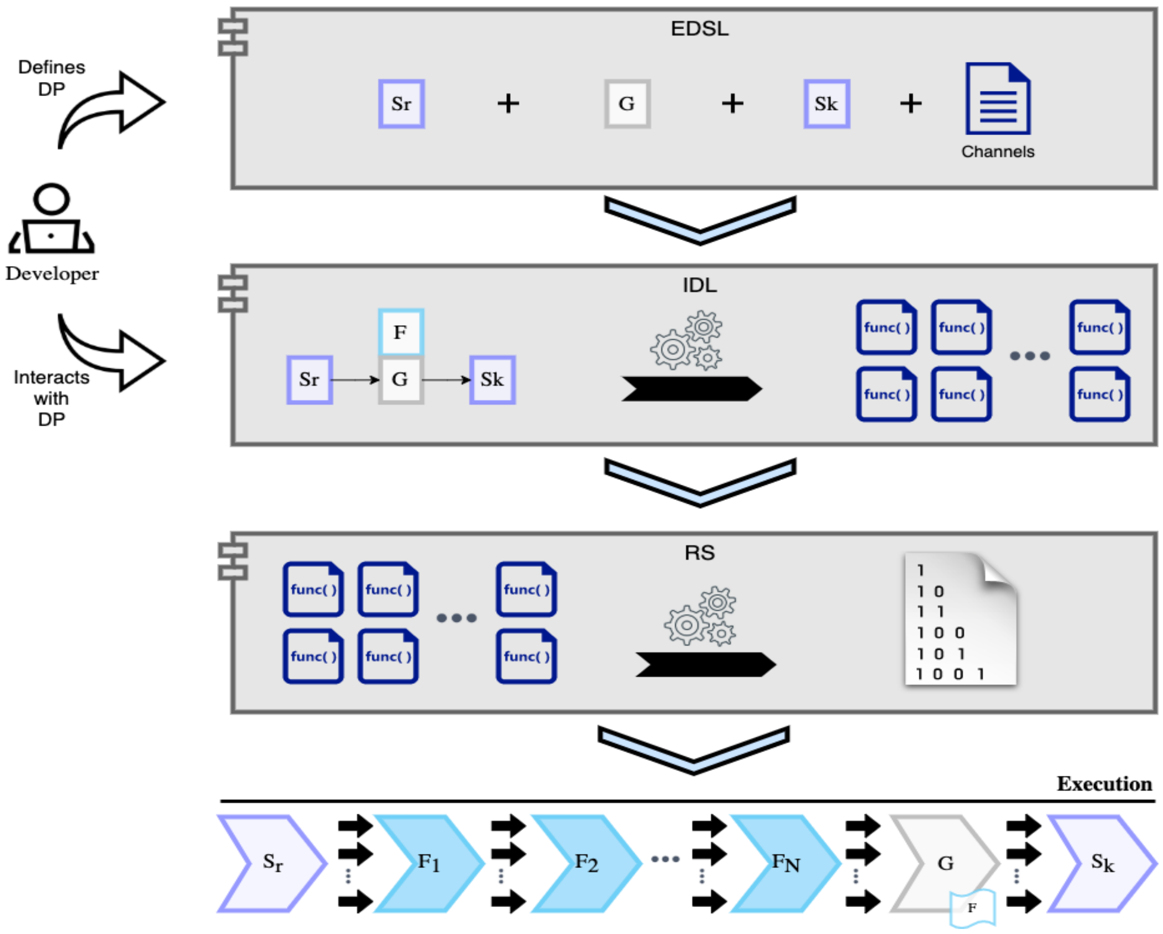


Figure 10: This diagram shows the system architecture of DPF-Haskell. DPF-Haskell is a DSL which is built on three main components: DSL, IDL and RS. In the DSL we can see how users can compose the main stages of the DPP. IDL is showing how the frameworks is helping users to transform that definition into real function or computations. Finally RS execute all that definition plus functions. Execution layer indicates an example of a DPP running after being executed.

information. `Eof` it is just a terminal type to disambiguate the `Channel` `(a :: Type)` of the tree branch. `Channel` can carry any type because it needs to be polymorphic to support a different number of channels and data types.

```haskell
    data chann1 :<+> chann2 = chann1 :<+> chann2
    deriving (Typeable, Eq, Show, Functor, Traversable, Foldable, Bounded)
    infixr 5 :<+>

    data a :=> b = a :=> b
    deriving (Typeable, Eq, Show, Functor, Traversable, Foldable, Bounded)
    infixr 5 :=>
```

Source Code 4.2: Special terminal symbols $\{$`:<+>`, `:=>`$\} \subset \Sigma$. This terminal symbols allows us to index two types in order to combine several of them and build a chain of stages (`:=>`) and a set of channels (`:<+>`).

There are two important terminal symbols in Σ: `:=>` and `:<+>`. In Source Code 4.2, the definition shows how `:=>` and `:<+>` can combine two types. The propose of writing `:=>` and `:<+>` as types is to have a syntactic sugar type combinator for writing the DSL according to the CFG. Moreover, having two distinguishable symbols in Σ allows to separate the encoding of the pipeline stages (Sr, G, Sk) from the encoding of channel composition in the same stages. This can be appreciate in Definition 1.

```haskell
type DPExample = Source (Channel (Int :<+> Eof))
              :=> Generator (Channel (Int :<+> Eof))
              :=> Sink
```

Source Code 4.3: This example shows the DSL encoding in DP-DSL for the problem of eliminating duplicated elements in a stream.

4.2.1. DSL Validation

The language generated by the grammar needs to be compiled to avoid errors or provide an incorrect dynamic pipeline definition. Fortunately, Haskell provides several Type-level techniques [22] which allows to verify properties of programs before running them, preventing users to introduce bugs, reducing errors. This verification done by the compiler establish a Curry-Howard Isomorphism [9], i.e. *Propositions as Types - Programs as Proof.* It is important to remark here that Haskell is not a theorem prover System like Coq[13], but some verifications, as we present in this work, can be done with GHC to ensure correctness on programs. Although Haskell provides tools to build advanced type-level verifications, all these techniques require the addition of *Haskell Language Extensions.*

Once we have the encoded the dynamic pipeline using the G_{dsl}, we proceed validating. The implementation of the validation is done using *Associated Type Families* [2].

[13]https://coq.inria.fr/

```haskell
type family And (a :: Bool) (b :: Bool) :: Bool where
    And 'True 'True = 'True
    And a b         = 'False


type family IsDP (dpDefinition :: k) :: Bool where
    IsDP (Source (Channel inToGen) :=> Generator (Channel genToOut) :=> Sink)
        = And (IsDP (Source (Channel inToGen))) (IsDP (Generator (Channel
        ↪  genToOut)))
    IsDP ( Source (Channel inToGen) :=> Generator (Channel genToOut) :=>
    ↪  FeedbackChannel toSource :=> Sink)
        = And (IsDP (Source (Channel inToGen))) (IsDP (Generator (Channel
        ↪  genToOut)))
    IsDP (Source (Channel (a :<+> more)))
        = IsDP (Source (Channel more))
    IsDP (Source (Channel Eof))                = 'True
    IsDP (Generator (Channel (a :<+> more)))  = IsDP (Generator (Channel more))
    IsDP (Generator (Channel a))               = 'True
    IsDP x                                     = 'False

type family ValidDP (a :: Bool) :: Constraint where
  ValidDP 'True = ()
  ValidDP 'False = TypeError
                    ( 'Text "Invalid Semantic for Building DP Program"
                     ':$$: 'Text "Language Grammar:"
                     ':$$: 'Text "DP       -> Source CHANS :=> Generator CHANS
                     ↪  :=> Sink"
                     ':$$: 'Text "DP       -> Source CHANS :=> Generator CHANS
                     ↪  :=> FEEDBACK :=> Sink"
                     ':$$: 'Text "CHANS    -> Channel CH"
                     ':$$: 'Text "FEEDBACK -> FeedbackChannel CH"
                     ':$$: 'Text "CH       -> Type :<+> CH | Eof"
                     ':$$: 'Text "Example: 'Source (Channel (Int :<+> Int)) :=>
                     ↪  Generator (Channel (Int :<+> Int)) :=> Sink'"
                    )
```

Source Code 4.4: Type Families `And`, `IsDP` and `ValidDP` which allows to perform a type-level validation over a DSL CFG definition.

In Source Code 4.4, there are 3(three) Type families that helps to validate the DSL CFG. `IsDP` associated type family is checking the production rules P of the grammar defined in Definition 1, returning a promoted data type [25] (not a boolean value) `'True` in case the production rule matches all the generated language, or `'False` otherwise. `ValidDP` is taking the result of `IsDP` type application, associating `'True` promoted boolean type to empty `()` constraint. An empty constraint is an indication of no restriction, meaning that if `ValidDP` is used as a constraint, and it is fully applied to `()`, it will give the compiler the evidence that there is no error at type-level. `ValidDP` is also associating `'False` to a custom `TypeError` which will appear at compilation time if the DPP DSL definition fully applies to that – a type checked error –.

```haskell
mkDP :: forall dpDefinition filterState filterParam st.
    ( ValidDP (IsDP dpDefinition)
    , DPConstraint dpDefinition filterState st filterParam)
 => Stage (WithSource dpDefinition (DP st))
 -> GeneratorStage dpDefinition filterState filterParam st
 -> Stage (WithSink dpDefinition (DP st))
 -> DP st ()
mkDP = ...

someFunc = mkDP @DPExample ...
```

Source Code 4.5: Definition of `mkDP` function of the Framework which uses type-level validation of the grammar `ValidDP (IsValid Type)`. Last line of the code is showing that using that function will compile-time check the definition of `DPExample` type.

4.2.2. Interpreter of DSL (IDL)

IDL component takes the dynamic pipeline specification made on with DSL component to interpret and generate the function definitions that needs to be implemented for solving a specific problem. In subsubsection 3.1.3, we have described what should be provided in a DPP algorithm: Sr, G, Sk, and the F with the non-empty set of Actors. The IDL generates the function definitions with an empty implementation to be implemented, ensuring that those functions will give "Proof" – in terms of Curry-Howard Correspondence [9] – of the "Propositions" defined on the DSL.

Similar techniques that we used on subsubsection 4.2.1 are also used in the IDL. On the first hand, *Type-level Defunctionalization* [3, 12] is used to let the compiler generates the signatures of the required functions. On the other hand, the framework uses *Term-level Defunctionalization* to interpret those functions. And finally, *Indexed Types* [8] and *Heterogeneous List* [11] are used to keep track of the dynamic number and polymorphic types of the functions parameters.

```haskell
withSource :: forall (dpDefinition :: Type) st. WithSource dpDefinition (DP st)
             -> Stage (WithSource dpDefinition (DP st))
withSource = mkStage' @(WithSource dpDefinition (DP st))

withGenerator :: forall (dpDefinition :: Type) (filter :: Type) st. WithGenerator
↪  dpDefinition filter (DP st)
                -> Stage (WithGenerator dpDefinition filter (DP st))
withGenerator = mkStage' @(WithGenerator dpDefinition filter (DP st))

withSink :: forall (dpDefinition :: Type) st. WithSink dpDefinition (DP st)
           -> Stage (WithSink dpDefinition (DP st))
withSink = mkStage' @(WithSink dpDefinition (DP st))
```

Source Code 4.6: This code is showing the different interpreters combinators to help users to generate the functions of the principal stages of the dynamic pipeline

In Source Code 4.6 we can appreciate the different combinators of the IDL that helps to interpret the DSL and generates the function definitions. `Stage` data type will be cover in Source Code 4.8, but it is a wrapper type of a pipeline stage – minimal unit of execution –, containing the function to be executed – here is the use *Term-level Defunctionalization* –. `withSource`, `withGenerator`, and `withSink` are aliases of the function `mkStage'` which is the combinator that is applying the *Associated Type* related to that stage. For example `withSource`, is equivalent to `mkStage' @(WithSource dpDefinition (DP st))`. For each *Associated Type Family* definition, there exist an equivalent term-level definition: `WithSource` type with `withSource` term , `WithGenerator` type with `withGenerator` term, and `WithSink` type with `withSink` term – notice the capital case letter "W" indicating the type and not the term –.

```haskell
type family WithSource (dpDefinition :: Type) (monadicAction :: Type -> Type) :: Type where
  WithSource (Source (Channel inToGen) :=> Generator (Channel genToOut) :=> Sink) monadicAction
      = WithSource (ChanIn inToGen) monadicAction
  WithSource (Source (Channel inToGen) :=> Generator (Channel genToOut) :=> FeedbackChannel toSource :=> Sink) monadicAction
      = WithSource (ChanOutIn toSource inToGen) monadicAction
  WithSource (ChanIn (dpDefinition :<+> more)) monadicAction
      = WriteChannel dpDefinition -> WithSource (ChanIn more) monadicAction
  WithSource (ChanIn Eof) monadicAction
      = monadicAction ()
  WithSource (ChanOutIn (dpDefinition :<+> more) ins) monadicAction
      = ReadChannel dpDefinition -> WithSource (ChanOutIn more ins) monadicAction
  WithSource (ChanOutIn Eof ins) monadicAction
      = WithSource (ChanIn ins) monadicAction
  WithSource dpDefinition _
      = TypeError
          ( 'Text "Invalid Semantic for Source Stage"
            ':$$: 'Text "in the DP Definition '"
            ':<>: 'ShowType dpDefinition
            ':<>: 'Text "'"
            ':$$: 'Text "Language Grammar:"
            ':$$: 'Text "DP       -> Source CHANS :=> Generator CHANS :=> Sink"
            ':$$: 'Text "DP       -> Source CHANS :=> Generator CHANS :=> FEEDBACK :=> Sink"
            ':$$: 'Text "CHANS    -> Channel CH"
            ':$$: 'Text "FEEDBACK -> FeedbackChannel CH"
            ':$$: 'Text "CH       -> Type :<+> CH | Eof"
            ':$$: 'Text "Example: 'Source (Channel (Int :<+> Int)) :=> Generator (Channel (Int :<+> Int)) :=> Sink'"
          )
```

Source Code 4.7: An example of the Associated Type Family `WithSource` that allows to implement *Type-level Defunctionalization* technique that will be the Type-level verification of the term `withSource`

In Source Code 4.7, in the highlighted lines, it can be seen how *Type-level Defunctionalization* is being expanded in a signature function definition with the form `WriteChannel a -> ReadChannel b -> ... -> monadicAction ()` depending on DPP language definition.

```haskell
data Stage a where
  Stage :: Proxy a -> a -> Stage a

mkStage' :: forall a. a -> Stage a
mkStage' = Stage (Proxy @a)
```

Source Code 4.8: `Stage` data type for implementing *Term-level Defunctionalization* providing evidence to the Type-Level Associated types

In Source Code 4.8, `Stage` data type uses a `Proxy` type. This `Proxy` type allows `Stage` to index the type definition generated by `a`. For example, in Source Code 4.6, when `withSource` interpreter is applied to `WithSource dpDefinition`, the compiler is provided with `dpDefinition` DSL type, expanding the function signature belonging to that DPP definition inside the `Stage`.

Generator and Filter. According to DPP definition in subsubsection 3.1.3, G has a F template in order to know how to dynamically interpose a new F during the runtime execution of the program. Let's first study F Data Type in the context of the framework.

```haskell
newtype Actor dpDefinition filterState filterParam monadicAction =
    Actor {  unActor :: MonadState filterState monadicAction => Stage (WithFilter
    ↪  dpDefinition filterParam monadicAction) }

newtype Filter dpDefinition filterState filterParam st =
    Filter { unFilter :: NonEmpty (Actor dpDefinition filterState filterParam
    ↪  (StateT filterState (DP st))) }
    deriving Generic
```

Source Code 4.9: This code shows the definition of the `Filter` data type which contains a non-empty set of `Actor`. The `Actor` data type is an `Stage` in the Context of the `MonadState` to allow keeping a local memory in the execution context of the filter.

In Source Code 4.9 the definition of the `Filter` data type contains a non-empty set of `Actor`. An `Actor` is a `Stage`, because an actor is the minimal unit of execution of a filter. A `Filter` has a `NonEmpty Actor` – Non-empty List – because a filter is built by a sequence of actors calls. Moreover, `Actor` Stage is defunctionalized with `WithFilter` *Associated Type Family*. `Filter` runs in an explicit `StateT` monadic context. This is because the F instance should have an state, according to DPP definition in subsubsection 3.1.3. For example, in the case of $\mathsf{DP}_{\mathsf{WCC}}$, as we have seen in subsection 3.2, F keeps an updated list of connected components that updates as long as it receives more edges that are connected with the current list of vertices. `Actor` data type – see Source Code 4.9 –, is constrained by `MonadState` which is in the same execution context of the whole `NonEmpty Actor` list of the `Filter`. This means the `StateT` is executed for each `Actor` of that filter, sharing the same state between them.

```haskell
mkFilter :: forall dpDefinition filterState filterParam st. WithFilter dpDefinition filterParam (StateT filterState (DP st))
         -> Filter dpDefinition filterState filterParam st
mkFilter = Filter . single

single :: forall dpDefinition filterState filterParam st. WithFilter dpDefinition filterParam (StateT filterState (DP st))
       -> NonEmpty (Actor dpDefinition filterState filterParam (StateT filterState (DP st)))
single = one . actor

actor :: forall dpDefinition filterState filterParam st. WithFilter dpDefinition filterParam (StateT filterState (DP st))
      -> Actor dpDefinition filterState filterParam (StateT filterState (DP st))
actor = Actor . mkStage' @(WithFilter dpDefinition filterParam (StateT filterState (DP st)))

(|>>>) :: forall dpDefinition filterState filterParam st. Actor dpDefinition filterState filterParam (StateT filterState (DP st))
       -> Filter dpDefinition filterState filterParam st
       -> Filter dpDefinition filterState filterParam st
(|>>>) a f = f & _Wrapped' %~ (a <|)
infixr 5 |>>>

(|>>) :: forall dpDefinition filterState filterParam st. Actor dpDefinition filterState filterParam (StateT filterState (DP st))
      -> Actor dpDefinition filterState filterParam (StateT filterState (DP st))
      -> Filter dpDefinition filterState filterParam st
(|>>) a1 a2 = Filter (a1 <|one a2)
infixr 5 |>>
```

Source Code 4.10: Combinators and small constructor to enable building actors and filter.

Finally, in Source Code 4.10, some combinators and smart constructors are provided in the framework to enable the construction of `Filter` and `Actor`. `mkFilter` is a smart constructor for `Filter` Data Constructor. `single` wraps one actor inside a `Filter`. `actor` is a smart constructor for `Actor` Data Constructor. `(|>>>)` is an appending combinator of an `Actor` to a `Filter`. `(|>>>)` also ensures actor execution order, i.e. the latest actor added is the latest to be executed.

```
data GeneratorStage dpDefinition filterState filterParam st = GeneratorStage
{ _gsGenerator      :: Stage (WithGenerator dpDefinition (Filter dpDefinition
↪  filterState filterParam st) (DP st))
, _gsFilterTemplate :: Filter dpDefinition filterState filterParam st
}
```

Source Code 4.11: `Generator` Data type which contains the `Stage` code of the generator itself, and the `Filter` template that it can be spawned by the `Generator`.

In Source Code 4.11, G contains a F template and its own stage behavior. `Generator` data type has a field with the `Filter` template that could be spawned by the algorithms defined by users according to the data received from its input channels. `Generator` has also another field with the behavior of the G – a `Stage` –.

4.2.3. Runtime System (RS)

The RS can be divided into two parts: the mechanism to generate stages dynamically in runtime, and the execution entry point of the DPP. Regarding execution entry point, all the stages that we have seen in previous sections are the pieces needed to build an executable `DP st a` monad. This executable monad has an existential type similar to `ST` monad to not escape out from the context on different stages. Once the dynamic pipeline starts to execute, the core of the framework dynamically generates stages between G and previous stages, according to users definitions, i.e. an *anamorphism* [20] that creates F instances until some condition is met.

```haskell
1  unfoldF :: forall dpDefinition readElem st filterState filterParam l.
   ↪  SpawnFilterConstraint dpDefinition readElem st filterState filterParam l
2          => UnFoldFilter dpDefinition readElem st filterState filterParam l
3          -> DP st (HList l)
4  unfoldF = loopSpawn
5
6  where
7    loopSpawn uf@UnFoldFilter{..} =
8      maybe (pure _ufRsChannels) (loopSpawn <=< doOnElem uf) =<< DP (pull
       ↪  _ufReadChannel)
9
10   doOnElem uf@UnFoldFilter{..} elem' = do
11     _ufOnElem elem'
12     if _ufSpawnIf elem'
13      then do
14        (reads', writes' :: HList l3) <- getFilterChannels <$> DP (makeChansF
          ↪  @(ChansFilter dpDefinition))
15        let hlist = elem' .*. _ufReadChannel .*. (_ufRsChannels `hAppendList`
          ↪  writes')
16        void $ runFilter _ufFilter (_ufInitState elem') hlist (_ufReadChannel .*.
          ↪  (_ufRsChannels `hAppendList` writes'))
17        return $ uf { _ufReadChannel = hHead reads', _ufRsChannels = hTail reads' }
18      else return uf
```

Source Code 4.12: `unfolF` is the *anamorphism* combinator to spawn new `Filter` types between the `Generator` and previous stages.

In Source Code 4.12, it is presented how is the *anamorphism* mechanims that generates dynamic stages between `G` and the previous stages. That *anamorphism* is implemented with the function `unfoldF`. That function receives an `UnFoldFilter` Data type, which contains the recipe for controlling that unfold recursive call. In line 12, `_ufSpawnIf` field of `UnFoldFilter`, indicates when to stop the recursion. Inside the conditional, in line 14, new channels are created for the new filter to be spawned. New channels connect the new filter with the previous stages and with `Generator`. After that, in line 16 `runFilter` starts the monadic computation, spawning the filter stage with its actors. Finally, the new list of channels are returned for the next recursive step to allow further channel connections.

```haskell
mkUnfoldFilter :: (readElem -> Bool)
    -> (readElem -> DP st ())
    -> Filter dpDefinition filterState filterParam st
    -> (readElem -> filterState)
    -> ReadChannel readElem
    -> HList l
    -> UnFoldFilter dpDefinition readElem st filterState filterParam l


mkUnfoldFilterForAll' :: (readElem -> DP st ())
                      -> Filter dpDefinition filterState filterParam st
                      -> (readElem -> filterState)
                      -> ReadChannel readElem
                      -> HList l
                      -> UnFoldFilter dpDefinition readElem st filterState
                      ↪  filterParam l

mkUnfoldFilterForAll :: Filter dpDefinition filterState filterParam st
                      -> (readElem -> filterState)
                      -> ReadChannel readElem
                      -> HList l
                      -> UnFoldFilter dpDefinition readElem st filterState
                      ↪  filterParam l
```

Source Code 4.13: Combinators for building `UnfoldFilter` types indicating the type of the `unfold` that users want to achieve.

Several smart constructors are also provided for building `UnfoldFilter` Data Type. In Source Code 4.13 the first combinator is the default smart constructor. i) First field `(readElem -> Bool)` indicate if the a new filter should be spawn or not. ii) Second field `(readElem -> DP st ())` is a monadic optional computation to do when received a new element, for example logging. iii) Third field `Filter` data type to be spawned. iv) Fourth field `(readElem -> filterState)` is initialization of the `Filter` State. v) Fifth field `(ReadChannel readElem)` that feeds the filter instance. vi) Last field is the *Heterogeneous List* with the rest of the channels to connect with other stages. . The combinator `mkUnfoldFilterForAll` is an smart constructor of `UnfoldFilter` that allows to spawn a new filter for each element received in the `G`.

4.3. Libraries and Tools

One of the most important task of the implementation is the selection of concurrency libraries to support an intensive parallelization workload. Parallelization techniques and tools have been intensively studied and implemented in Haskell [19]. Indeed, it is well known that green threads and sparks allow spawning thousands to millions of parallel computations. These parallel computations do not penalize performance when compare with Operative System (OS) level threading [16]. A straightforward assumption to achieve here, is to use `monad-par` library[14]. Nevertheless, in this work, we have discarded the use of sparks [18, 17] because we can achieve the level of required parallelism spawning green threads only. The next obvious choice is to use `forkIO :: IO () -> IO ThreadId` from `base` library[15]. However, that would imply handling all the threads lifecycles and errors programmatically without any abstraction to facilitate that complex task. Therefore, we choose `async` library[16] which enables to spawn asynchronous computations [16] on Haskell using green threads, and at the same time, it provides useful combinators to managing thread terminations and errors.

Regarding channels, there are several techniques to communicate threads or sparks in Haskell like `MVar` or concurrent safe mechanisms like Software Transactional Memory (STM) [7]. At the same time, in Haskell library ecosystem, there are `Channels` abstractions based on previous mentioned communication techniques. In that sense, for conducting the communication between dynamic stages and data flowing in a dynamic pipeline, we have selected `unagi-chan` library [17] which provides the following advantages to our solution: Firstly, `MVar` channel without using STM reducing overhead. STM is not required in a dynamic pipeline because each specific stage running in a separated thread, can only access to its `I/O` channels for reading/writing accordingly, and these operations are not concurrently shared by other threads (stages) for the same channels. Second, non-blocking channels. `unagi-chan` library contains blocking and non-blocking channels for reading. This aspect is key to gain speed up on the implementation. Third, the library is optimized for $x86$ architectures with use of low-level `fetch-and-add` instructions. Finally, `unagi-chan` is $100x$ faster[18] on Benchmarking compare with STM and default base `Chan` implementations.

5. Enumerating Weakly Connected Component on the DPF

In this section we present the most relevant details of the implementation of the $\mathsf{DP}_{\mathsf{WCC}}$ using DPF-Haskell. In addition, we describe the experiments and discuss the results of the empirical evaluation that we have conducted to analyze the performance of this implementation.

[14]`https://hackage.haskell.org/package/monad-par`
[15]`https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Concurrent.html`
[16]`https://hackage.haskell.org/package/async`
[17]`https://hackage.haskell.org/package/unagi-chan`
[18]`https://github.com/jberryman/unagi-chan`

5.1. $\mathsf{DP}_{\mathsf{WCC}}$ *Implementation*

As we said before, the $\mathsf{DP}_{\mathsf{WCC}}$ implementation has been made as a proof of concept to understand and explore the limitations and challenges that we could find in the development of a future DPF in Haskell. In Section 4 we emphasize that the focus of DPF in Haskell is on the IDL component. Hence, the development of the $\mathsf{DP}_{\mathsf{WCC}}$ is as general as possible using most of the constructs and abstractions required by the IDL. Lets introduce the minimal code needed for encoding any DPP using DPF-Haskell. [19]

```haskell
type DPConnComp = Source (Channel (Edge :<+> ConnectedComponents :<+> Eof))
            :=> Generator (Channel (Edge :<+> ConnectedComponents :<+> Eof))
            :=> Sink

program :: FilePath -> IO ()
program file = runDP $ mkDP @DPConnComp (source' file) generator' sink'
```

Source Code 5.1: In this code we can appreciate the main construct of our $\mathsf{DP}_{\mathsf{WCC}}$ which is a combination of $\mathsf{Sr}_{\mathsf{WCC}}$, $\mathsf{G}_{\mathsf{WCC}}$ and $\mathsf{Sk}_{\mathsf{WCC}}$

In Source Code 5.1 there are two important declarations. First, the *Type Level* declaration of the $\mathsf{DP}_{\mathsf{WCC}}$ to indicate DPF-Haskell how our stages are going be connected, and using that *Type Level* construct, we use the IDL to allow the framework interpret the type representation of our DPP and ensuring at compilation time that we provide the correct stages, *Source* ($\mathsf{Sr}_{\mathsf{WCC}}$), *Generator* ($\mathsf{G}_{\mathsf{WCC}}$) and *Sink* ($\mathsf{Sk}_{\mathsf{WCC}}$), that matches those declaration. According to this declaration what we need to provide is the correct implementation of `source'`, `generator'` and `sink'` which *Type checked* the DPP type definition[20].

[19]All the code that we expose here can be accessed publicly in `https://github.com/jproyo/dynamic-pipeline/tree/main/examples/Graph`

[20]The names of the functions are completely choosen by the user of the framework and it should not be confused with the internal framework combinators.

```haskell
    source' :: FilePath
            -> Stage
              (WriteChannel Edge -> WriteChannel ConnectedComponents -> DP st ())
    source' filePath = withSource @DPConnComp
      $ \edgeOut _ -> unfoldFile filePath edgeOut (toEdge . decodeUtf8)

    sink' :: Stage (ReadChannel Edge -> ReadChannel ConnectedComponents -> DP st
    ↪  ())
    sink' = withSink @DPConnComp $ \_ cc -> withDP $ foldM_ cc print

    generator' :: GeneratorStage DPConnComp ConnectedComponents Edge st
    generator' =
      let gen = withGenerator @DPConnComp genAction
      in  mkGenerator gen filterTemplate
```

Source Code 5.2: In this code we can appreciate the $\mathsf{Sr}_{\mathsf{WCC}}$, $\mathsf{G}_{\mathsf{WCC}}$ and $\mathsf{Sk}_{\mathsf{WCC}}$ functions that matches the type level definition of the DP. $\mathsf{Sr}_{\mathsf{WCC}}$ and $\mathsf{Sk}_{\mathsf{WCC}}$ are completely trivial but $\mathsf{G}_{\mathsf{WCC}}$ will be analyzed later due to its internal complexity.

As we appreciate in Source Code 5.2, $\mathsf{Sr}_{\mathsf{WCC}}$ and $\mathsf{Sk}_{\mathsf{WCC}}$ are trivial. In the case of `source'` the only work it needs to do is to read the input data edge by edge and downstream to the next stages. That process is achieved with a DPF-Haskell combinator called `unfoldFile` which is a *catamorphism* of the input data to the stream. `sink'` delivers to the output of the program the upstream connected components received from previous stages. $\mathsf{Sk}_{\mathsf{WCC}}$ implementation is done using an *anamorphism* combinator provided by the framework as well, which is `foldM_`. The $\mathsf{G}_{\mathsf{WCC}}$ Stage is a little more complex because it contains the core of the algorithm explained in subsection 3.2. According to what we described in subsubsection 3.1.3, *Generator* stage spawns a *Filter* on each received edge in our case of $\mathsf{DP}_{\mathsf{WCC}}$. Therefore, it needs to contain that recipe on how to generate a new *Filter* instance – in our case of Haskell it is a defunctionalized Data Type or Function –. Then, there are two important functions to describe: `genAction` which tells how to spawn a new *Filter* and under what circumstances, and `filterTemplate` which carries the function to be spawn.

```haskell
1  genAction :: Filter DPConnComp ConnectedComponents Edge st
2            -> ReadChannel Edge
3            -> ReadChannel ConnectedComponents
4            -> WriteChannel Edge
5            -> WriteChannel ConnectedComponents
6            -> DP st ()
7  genAction filter' readEdge readCC _ writeCC = do
8    let unfoldFilter = mkUnfoldFilterForAll filter' toConnectedComp readEdge
   ↪  (readCC .*. HNil)
9    results <- unfoldF unfoldFilter
10   foldM_ (hHead results) (`push` writeCC)
```

Source Code 5.3: In this code we can appreciate the Generator Action code which will expand all the filters in runtime in front of it and downstream all the connected components calculated for those, to the Sink

DPF-Haskell provides several combinators to help the user with the *Generator* code, in particular with the spawning process as it has been describe in section 4. `genAction` for DP$_{WCC}$ will use the combinator `mkUnfoldFilterForAll` which will spawn one *Filter* per received edge in the channel, expanding dynamically the stages on runtime. In line 10, we can appreciate how after expanding the filters, the generator will downstream to the *Sink*, the received Connected Components calculated from previous filters.

```haskell
filterTemplate :: Filter DPConnComp ConnectedComponents Edge st
filterTemplate = actor actor1 |>> actor actor2

actor1 :: Edge
       -> ReadChannel Edge
       -> ReadChannel ConnectedComponents
       -> WriteChannel Edge
       -> WriteChannel ConnectedComponents
       -> StateT ConnectedComponents (DP st) ()
actor1 _ readEdge _ writeEdge _ =
  foldM_ readEdge $ \e -> get >>= doActor e
 where
  doActor v conn
    | toConnectedComp v `intersect` conn = modify' (toConnectedComp v <>)
    | otherwise = push v writeEdge

actor2 :: Edge
       -> ReadChannel Edge
       -> ReadChannel ConnectedComponents
       -> WriteChannel Edge
       -> WriteChannel ConnectedComponents
       -> StateT ConnectedComponents (DP st) ()
actor2 _ _ readCC _ writeCC = do
  foldWithM_ readCC pushMemory $ \e -> get >>= doActor e

 where
   pushMemory = get >>= flip push writeCC

   doActor cc conn
    | cc `intersect` conn = modify' (cc <>)
    | otherwise = push cc writeCC
```

Source Code 5.4: Filter template code composed by 2 Sequential Actors that will calculate the Connected Components and downstream them.

Finally, in Source Code 5.4 the *Filter* template code is defined. As we have seen in subsection 3.2, DP_{WCC} *Filter* is composed of 2 Actors. The first actor collect all the possible vertices that are incidence to some vertices edge that was instantiated with. Once it does not receive any more edges, it starts downstream it set of vertices to the following filters in order to build a maximal connected component, this is actor2. At the end of processing, actor2 will downstream its connected component to the following stages. As we show, with the help of the Haskell Dynamic Pipeline Framework, building a DPP algorithm like WCC enumeration consist in few lines of codes with the *Type Safety* that Haskell provides.

6. Empirical Evaluation

The empirical study aims at evaluating $\mathsf{DP}_{\mathsf{WCC}}$ implemented in the baseline Haskell and DPF-Haskell. Our goal is to answer the following research questions:

RQ1) Does $\mathsf{DPFH}_{\mathsf{WCC}}$ exhibit similar execution time performance compared with $\mathsf{DP}_{\mathsf{WCC}}$? **RQ2)** Does $\mathsf{DPFH}_{\mathsf{WCC}}$ enhance the continuous behavior with respect to other approaches? **RQ3)** Do the proposed approaches to $\mathsf{DP}_{\mathsf{WCC}}$ handle memory efficiently?

We have implemented three solutions to the problem of weakly connected components:

- $\mathsf{BLH}_{\mathsf{WCC}}$: implemented in `containers` Haskell library [21] using `Data.Graph`.
- $\mathsf{DP}_{\mathsf{WCC}}$: baseline implementation of the weakly connected components as a dynamic pipeline (subsection 3.2).
- $\mathsf{DPFH}_{\mathsf{WCC}}$: implementation of the weakly connected components on top of the DPF-Haskell framework (section 5).

Running Architecture. All the experiments have been executed in a $x86$ 64 bits architecture with a 6-*Core Intel Core i7* processor of 2, 2 GHz, which can emulate up to 12 virtual cores. This processor has *hyper-threading* enable. Regarding memory, the machine has 32*GB DDR4* of RAM, 256 *KB* of L2 cache memory, and 9 *MB* of L3 cache.

Haskell Setup. We have used the following Haskell setup for $\mathsf{DP}_{\mathsf{WCC}}$ implementation: GHC version 8.10.4, `bytestring 0.10.12.0`[22], `containers 0.6.2.1`[23], `relude 1.0.0.1`[24] and `unagi-chan 0.4.1.3`[25]. The use of `relude` library is because we disabled `Prelude` from the project with the language extension (language options) `NoImplicitPrelude`[26]. Regarding compilation flags (GHC options) we have compiled our program with `-threaded`, `-O3`, `-rtsopts`, `-with-rtsopts=-N`. Since we have used `stack` version 2.5.1 [27] as a building tool on top of GHC the compilation command is `stack build`[28]. Additionally, the setup for $\mathsf{DPFH}_{\mathsf{WCC}}$ is the same as the exposed before but with the difference that this setup use `dynamic-pipeline 0.3.2.0`[29] library written in the context of this work.

DataSets. For all the experiments, we have used the following networks taken from SNAP [28].

[21]`https://hackage.haskell.org/package/containers`
[22]`https://hackage.haskell.org/package/bytestring`
[23]`https://hackage.haskell.org/package/containers`
[24]`https://hackage.haskell.org/package/containers`
[25]`https://github.com/jberryman/unagi-chan`
[26]`https://downloads.haskell.org/ghc/8.8.4/docs/html/users_guide/`
[27]`https://docs.haskellstack.org/en/stable/README/`
[28]For more information about package.yaml or cabal file please check `https://github.com/jproyo/upc-miri-tfm/tree/main/connected-comp`
[29]`https://hackage.haskell.org/package/dynamic-pipeline`

Network	**Nodes**	**Edges**	**Diameter**	**#WCC**	**#Nodes Largest WCC**
Enron Emails	36,692	183,831	11	1,065	33,696 (0.918)
Astro Physics Collaboration Net	18,772	198,110	14	290	17,903 (0.954)
Google Web Graph	875,713	5,105,039	21	2,746	855,802 (0.977)

Table 1: DataSet of Graphs Selected

The criteria for selecting the networks have been followed the idea or testing the solution in more complex graphs, in which all of them are undirected graphs but with different sizes concerning its number of nodes as we can see in Table 1.

6.1. Experiments Definition

E1: Implementation Analysis. In this experiment, we measure GHC statistics running time enabling `+RTS -s` flags. The metrics that we measure are *MUT Time* which is the amount of time in seconds GHC is running computations and *GC Time* which is the number of seconds that GHC is running garbage collector. *Total execution time* is the sum of both in seconds. At the same time, we are going to check the correctness of the output counting the number of WCC generated by the algorithm against the already known topology of it in section 6. Similarly, in this experiment, we measure GHC statistics running time enabling `+RTS -s` flags. The metrics that we measure are *MUT Time* which is the amount of time in seconds GHC is running computations and *GC Time* which is the number of seconds that GHC is running garbage collector. *Total execution time* is the sum of both in seconds. The experiment provides evidence to answer the research question [RQ2].

E2: Benchmark Analysis. This experiment measures *Average Running Time.* The *Average Running Time* is the average running time of 1000 resamples using `criterion` tool. In each sample, the running time is measure from the beginning of the execution of the program until when the last answer is produced. This experiment will help to answer research question [RQ1].

E3: Continuous Behavior - Diefficiency Metrics. In this experiment, we conduct two benchmark analysis over execution time comparing $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$. In the first benchmark analysis, we use `criterion` tool in Haskell which runs over four iterations of each of the algorithms to get a mean execution time in seconds and compare the results in a plot. In the second benchmark, we use Diefficency Metrics (Dm) Tool `diefpy` in order to measure with the ability of DPP model to generate results incrementally [1]. This is one of the strongest feature of DPP Paradigm since it allows process and generate results without no need of waiting for processing until the last element of the data source. This kind of aspect is essential not only for big data inputs where perhaps the requirements allow for processing until some point of the time having partial results but at the same time is important to process unbounded streams. This experiment measures Diefficency Metrics described in subsubsection 3.1.2, and will help to answer research question [RQ2].

E4: Performance Analysis. In this experiment, we measure internal parallelism in GHC and memory usage during the execution of one of the example networks. The motivation of this is to verify empirically how $\mathsf{DP}_{\mathsf{WCC}}$ is handling parallelization and memory usage. This experiment is conducted using two tools, *ThreadScope* Tool[30] for conducting multithreading analysis and *eventlog2html* Tool[31]

[30] `https://wiki.haskell.org/ThreadScope`
[31] `https://mpickering.github.io/eventlog2html/`

to conduct memory usage analysis. Regarding multithreading analysis the metrics that we measure are the distribution of threads among processors over execution time which is how many processors are executing running threads over the whole execution; and the mean number of running threads per time slot which is calculated by zooming in 8 time slots and taking the mean number of threads per processor to see if it is equally distributed among them. In regards to memory management, the metric that we measure is the amount of memory in MB consumed per data type during the whole execution time. The experiment helps to answer the research questions [RQ1,RQ3].

6.2. Discussion of Observed Results

Experiment: E1. The following represents the execution for running these graphs on our DPP implementation.

Network	**Exec Param**	**MUT Time**	**GC Time**	**Total Time**
Enron Emails	`+RTS -N4 -s`	2.797s	0.942s	3.746s
Astro Physics Coll Net	`+RTS -N4 -s`	2.607s	1.392s	4.014s
Google Web Graph	`+RTS -N8 -s`	137.127s	218.913s	**356.058s**

Table 2: Total Execution times of each of the networks implemented with $\mathsf{DP}_{\mathsf{WCC}}$. *MUT Time* is the time of running or executing code and *GC Time* is the time that the program spent doing Garbage collection. *Total Execution time* is the sum of both times

It is important to point out that since the first two networks are smaller in the number of edges compared with *web-Google*, executing those with 8 cores as the `-N` parameters indicates, does not affect the final speed-up since GHC is not distributing threads on extra cores because it handles the load with 4 cores only.

As we can see in Table 2, we are obtaining remarkable execution times for the first two graphs and it seems not to be the case for *web-Google*. Doing a deeper analysis on the topology of this last graph, we can see according to Table 1 that the number of *Nodes in the largest WCC* is the highest one. This means that there is a WCC which contains 97.7% of the nodes. Moreover, we can confirm that if we analyze even deeper how is the structure of that WCC with the output of the algorithm, we can notice that the largest WCC is the last one on being processed. Having that into consideration we can state that due to the nature of our algorithm which needs to wait for collecting all the vertices in the `actor2` filter stage it penalizes our execution time for that particular case. A more elaborated technique for implementing the actors is required to speed up execution.

Regarding the correctness of the output, we have verified with the outputs that the number of connected components is the same as the metrics already gathered in Table 1.

The following represents the *Total execution time* for each of the networks implemented with $\mathsf{DPFH}_{\mathsf{WCC}}$.

Network	**Exec Param**	**MUT Time**	**GC Time**	**Total Time**
Enron Emails	`+RTS -N4 -s`	1.795s	0.505s	2.314s
Astro Physics Coll Net	`+RTS -N4 -s`	2.294s	1.003s	3.311s
Google Web Graph	`+RTS -N8 -s`	169.381s	270.784s	440.176s

Table 3: Total Execution times of each of the networks implemented with $\mathsf{DPFH}_{\mathsf{WCC}}$. *MUT Time* is the time of running or executing code and *GC Time* is the time that the program spent doing Garbage collection. *Total Execution time* is the sum of both times

In Table 3, we are obtaining similar execution times compared with Table 2 in subsection 3.2. In fact, for *Enron Emails* and *Astro Physics Coll* networks, all times are better than the implementation of the *Proof of Concept*. Regarding *Google Web* network, the time is slightly worse but there is the fact that DPF-Haskell is adding some overhead over plain code execution. According to these results we can partially answer [RQ1], because the implementation of $\mathsf{DPFH}_{\mathsf{WCC}}$ has similar performance compared with *Proof of Concept* implementation.

Experiment: E2.

Criterion Benchmark. In Figure 11, orange bars report the time taken by `Data.Graph` in DP_{WCC} in Haskell `containers` library[32]. Blue light bars represent the time taken by DP_{WCC} in Haskell.

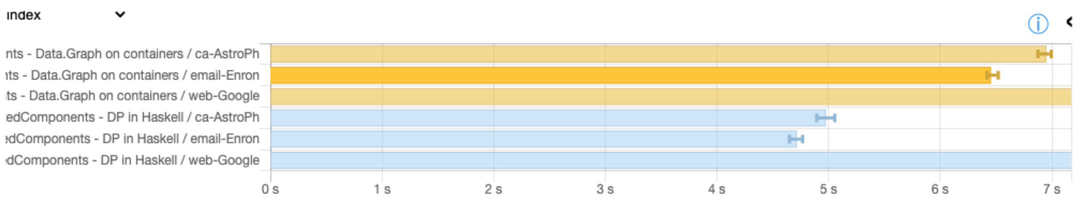


Figure 11: Benchmark of average execution time of $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$. Average Execution time of running 1000 samples over each of the networks.

Figure 11 shows that $\mathsf{DP}_{\mathsf{WCC}}$ solution is 1.3 faster compare with $\mathsf{BLH}_{\mathsf{WCC}}$. Despite this, if we zoom in Figure 11, it can be observed that $\mathsf{DP}_{\mathsf{WCC}}$ solution is slower compared with $\mathsf{BLH}_{\mathsf{WCC}}$. Regarding mean execution times for each implementation on each case measure by `criterion` library, we can display the following results:

Network	$\mathsf{DP}_{\mathsf{WCC}}$	$\mathsf{BLH}_{\mathsf{WCC}}$	**Speed-up**
Enron Emails	4.68s	6.46s	1.38
Astro Physics Coll Net	4.98s	6.95s	1.39
Google Web Graph	386s	106s	-3.64

Table 4: Comparison of Mean Execution times between $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$ for each network

These results allow for answering Question [Q2]. We already had a partial answer with the previous experiment E1 about [Q2] (section 6) where we have seen that the graph topology is affecting the performance and the parallelization, penalizing $\mathsf{DP}_{\mathsf{WCC}}$ for this particular case. In this benchmark, the solution against $\mathsf{BLH}_{\mathsf{WCC}}$ confirms the hypothesis.

Moreover, Figure 12, we can appreciate the *Average Execution Time* for each of the networks after running `criterion` tool for the $\mathsf{DPFH}_{\mathsf{WCC}}$ implementation confirming that after changing the DPP implementation using DPF-Haskell the performance remains stable.

[32]`https://hackage.haskell.org/package/containers`

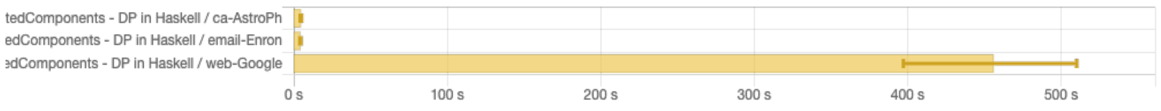


Figure 12: Average Execution time of running 1000 samples over each of the networks with $\mathsf{DPFH_{WCC}}$ implementation.

If we compare those *Average Execution Times* with the previous obtained in subsection 3.2, we have the following comparison table.

Network	$\mathsf{DPFH_{WCC}}$	$\mathsf{DP_{WCC}}$	**Speed-up**
Enron Emails	4.30s	4.68s	0.91
Astro Physics Coll Net	4.76s	4.98s	0.95
Google Web Graph	456s	386s	-1.18

Table 5: Comparison of Average Execution Times between $\mathsf{DPFH_{WCC}}$ and the implementation of baseline implementation ($\mathsf{DP_{WCC}}$)

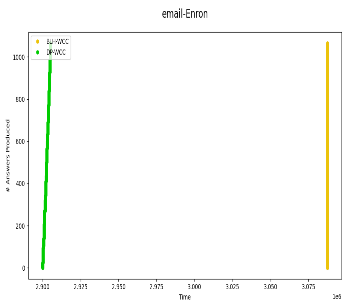
As we can see in Table 5, all the *Average Execution times* are better for the new implementation with $\mathsf{DPFH_{WCC}}$ compared with the $\mathsf{DP_{WCC}}$. As it is consistent with the *Total Execution time* of the previous experiment in section 6.2, the only network that perform slightly worse is *Google Web*, but the difference is not significant enough taking into consideration the overhead introduced by DPF-Haskell. These results allow for answering the research question [RQ1] confirming that the performance in terms of execution is better for smaller networks and almost similar for bigger networks.

Experiment: E3. Some considerations are needed before starting to analyze the data gathered with Dm tool. Firstly, the tool is plotting the results according to the traces generated by the implementation, both $\mathsf{DP_{WCC}}$ and $\mathsf{BLH_{WCC}}$. By the nature of DPP model, we can gather or register that timestamps as long as the model is generating results. In the case of $\mathsf{BLH_{WCC}}$, this is not possible since it calculates WCC at once. This is not an issue and we still can check at what point in time all WCC in $\mathsf{BLH_{WCC}}$ are generated. In those cases, we are going to see a straight vertical line.

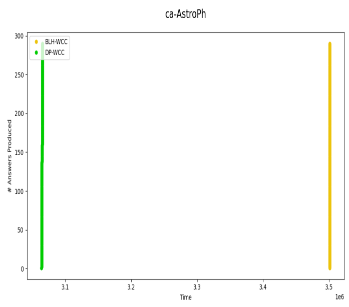
Network	**Implementation**	**dief@t Metric**	**dief@k Metric**
ca-AstroPh	$\mathsf{DP_{WCC}}$	1.27×10^8	1.99×10^5
	$\mathsf{BLH_{WCC}}$	0	0
email-Enron	$\mathsf{DP_{WCC}}$	1.97×10^8	2.51×10^6
	$\mathsf{BLH_{WCC}}$	0	0
web-Google	$\mathsf{DP_{WCC}}$	1.10×10^7	1.10×10^7
	$\mathsf{BLH_{WCC}}$	8.75×10^{11}	0

Table 6: This tables shows the `dief@t` and `dief@k` values gather for $\mathsf{DP_{WCC}}$ in DPF-Haskell. We can appreciate that in all cases $\mathsf{DP_{WCC}}$ has a higher value of `dief@t` and a lower value of `dief@k` showing continuos behavior

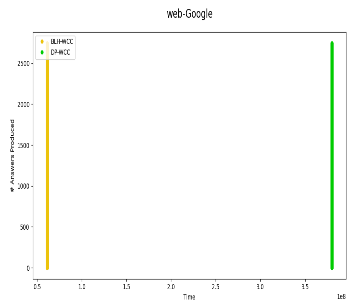
Having said that, we can see the results of Dm which are presented in two types of plots. The first one is regular line graphs in where the x axis shows the time escalated when the result was generated and the y axis is showing the component number that was generated at that time. The second type of plot is a radar plot in which shows how the solution is behaving on the dimensions of Time for the first tuple (TFFT), Execution Time (ET), Throughput (T), Completeness (Comp) and Diefficiency Metric `dief@t` (`dief@t`) and how are the tension between them; all these metrics are higher is better. All the details about these metrics are explained here [1].



(a) email-Enron Diefficency Metrics $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$ Line Plot



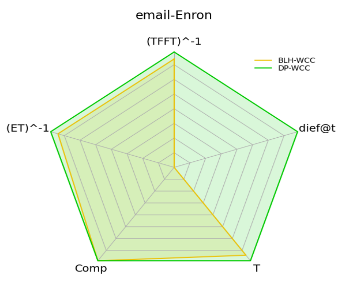
(b) ca-AstroPh Diefficency Metrics $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$ Line Plot



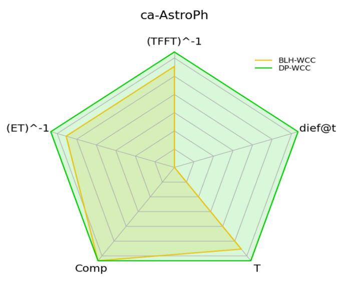
(c) web-Google Diefficency Metrics $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$ Line Plot

Figure 13: These figures show `dief@t` observed results after running all the scenarios for each network with $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$. y axis represents the number of Answers produced and x axis is the t time of the `dief@t` metric describe in subsubsection 3.1.2. The more data points distributed throughout the x axis, the higher, the continuous behavior. The scale in axis x which represents *Time t*, where t is the microsecond difference between the start of the program and the result delivered by it.

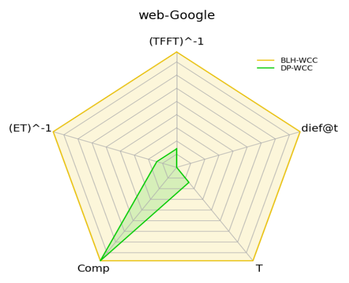
Based on the results shown in all the figures above, all the solutions in $\mathsf{DP}_{\mathsf{WCC}}$ are being generated incrementally, but there is some difference that we would like to remark. In the case of *email-Enron* and *ca-AstroPh* graphs as we can see in Figure 13a and Figure 13b, there seems to be a more incremental generation of results. This is behavior is measured with the values of Diefficiency Metric `dief@t` (`dief@t`). *ca-AstroPh* as it can be seen in Figure 13b, is even more incremental showing a clear separation between some results and others. The *web-Google* network which is shown in Figure 13c, is a little more linear and that is because all the results are being generated with very little difference in time between them. Having the biggest WCC at the end of *web-Google*, DPP algorithm it is retaining results until the biggest WCC can be solved, which takes longer.



(a) email-Enron Diefficency Metrics $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$ Radar Plot



(b) ca-AstroPh Diefficency Metrics $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$ Radar Plot



(c) web-Google Diefficency Metrics $\mathsf{DP}_{\mathsf{WCC}}$ vs. $\mathsf{BLH}_{\mathsf{WCC}}$ Radar Plot

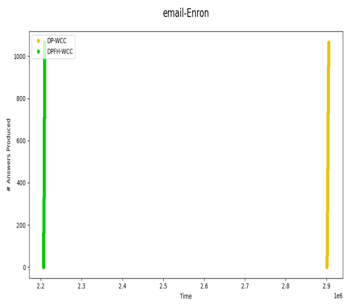
Figure 14: Radial plots show how the different dimensions values provided by `diefpy` tool such as T, TFFT, `dief@t`, ET and Comp are related each other for each experimental case. These figures show radial plot observed results after running for each network. `dief@t` is described in subsubsection 3.1.2.

As we can appreciate in the above radar plots our previous analysis can be confirmed. We can see for example that the Throughput of *web-Google* in Figure 14c, in the case of $\mathsf{DP}_{\mathsf{WCC}}$ is worse than $\mathsf{BLH_{WCC}}$, which is not happening for the others. We can say that regarding [Q2] (section 6) although $\mathsf{DP_{WCC}}$ is faster than the traditional approach, the speed-up dimension execution factor is not always the most interest analysis that we can have, because as we have seen even when in the case of *web-Google* Graph $\mathsf{DP_{WCC}}$ is slower at execution, it is at least generating incremental results without the need to wait for the rest of the computations.

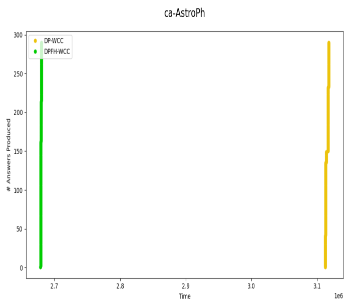
Moreover, we have run a comparison analysis between $\mathsf{DP_{WCC}}$ and $\mathsf{DPFH_{WCC}}$, using the same Diefficency Metrics, to confirm the new implementation $\mathsf{DPFH_{WCC}}$ is still presenting continuous behavior delivering incremental results.

Network	Implementation	dief@t Metric	dief@k Metric
ca-AstroPh	$\mathsf{DPFH_{WCC}}$	1.27×10^8	1.80×10^5
	$\mathsf{DP_{WCC}}$	8.77×10^5	8.77×10^5
email-Enron	$\mathsf{DPFH_{WCC}}$	7.42×10^8	1.38×10^6
	$\mathsf{DP_{WCC}}$	1.98×10^6	1.98×10^6
web-Google	$\mathsf{DPFH_{WCC}}$	1.29×10^7	1.29×10^7
	$\mathsf{DP_{WCC}}$	2.35×10^{10}	1.17×10^7

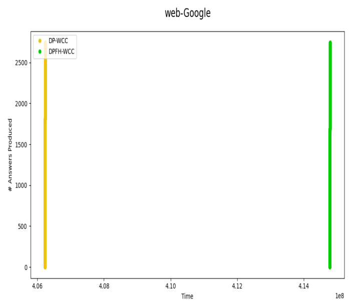
Table 7: This tables shows the `dief@t` and `dief@k` values gather for $\mathsf{DP_{WCC}}$ in DPF-Haskell. We can appreciate that in all cases $\mathsf{DP_{WCC}}$ has a higher value of `dief@t` and a lower value of `dief@k` showing continuos behavior



(a) email-Enron Diefficency Metrics $\mathsf{DPFH_{WCC}}$ vs. $\mathsf{DP_{WCC}}$ Line Plot



(b) ca-AstroPh Diefficency Metrics $\mathsf{DPFH_{WCC}}$ vs. $\mathsf{DP_{WCC}}$ Line Plot

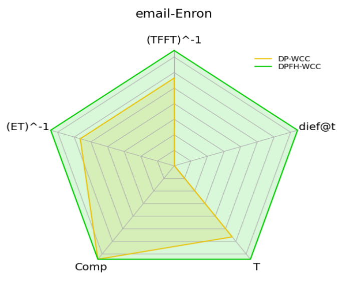


(c) web-Google Diefficency Metrics $\mathsf{DPFH_{WCC}}$ vs. $\mathsf{DP_{WCC}}$ Line Plot

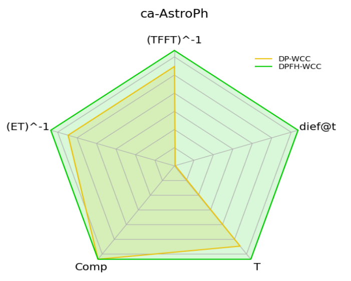
Figure 15: These figures show `dief@t` observed results after running all the scenarios for each network with $\mathsf{DPFH_{WCC}}$ vs. $\mathsf{DP_{WCC}}$. y axis represents the number of Answers produced and x axis is the t time of the `dief@t` metric describe in subsubsection 3.1.2. The more data points distributed throughout the x axis, the higher, the continuous behavior. The scale in axis x which represents *Time* t, where t is the microsecond difference between the start of the program and the result delivered by it.

Based on the results shown in Figure 15 and with the support of the metrics values in Table 7, it can be appreciated that both solutions, $\mathsf{DP_{WCC}}$ and $\mathsf{DPFH_{WCC}}$ show continuous behavior. Moreover, in Table 7, $\mathsf{DPFH_{WCC}}$ has a higher value of `dief@t` and a lower value of `dief@k` for *ca-AstroPh* and *email-Enron* confirming the continuous behavior. In the case of *web-Google* which is the biggest network, $\mathsf{DP_{WCC}}$ seems to be more continuous than $\mathsf{DPFH_{WCC}}$ according to its `dief@t` and `dief@k` values, but that does not mean that $\mathsf{DPFH_{WCC}}$ is not continuous. If that was the case its values should be 0. As we can appreciate in Figure 16 radar plots also confirm our previous analysis on continuity.

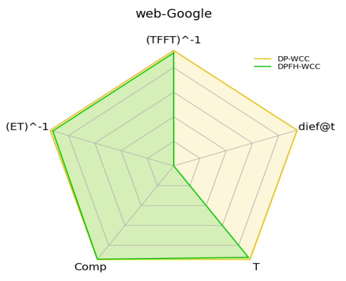
In conclusion, we can say that regarding [RQ2] (section 6) although $\mathsf{DP_{WCC}}$ is faster and more continuous in *web-Google* Graph, $\mathsf{DPFH_{WCC}}$ is much more faster and more continuous in the other



(a) email-Enron Diefficency Metrics $\mathsf{DPFH_{WCC}}$ vs. $\mathsf{DP_{WCC}}$ Radar Plot



(b) ca-AstroPh Diefficency Metrics $\mathsf{DPFH_{WCC}}$ vs. $\mathsf{DP_{WCC}}$ Radar Plot



(c) web-Google Diefficency Metrics $\mathsf{DPFH_{WCC}}$ vs. $\mathsf{DP_{WCC}}$ Radar Plot

Figure 16: Radial plots show how the different dimensions values provided by `diefpy` tool such as T, TFFT, `dief@t`, ET and Comp are related each other for each experimental case. These figures show radial plot observed results after running for each network. `dief@t` is described in subsubsection 3.1.2.

networks, showing that it enhance the continuity approach.

Experiment: E3. For this type of analysis, our experiment focuses on *email-Enron* network only because profiling data generated by GHC is big enough to conduct the analysis and on the other, and enabling profiling penalize execution time. Moreover, it is important to remark that the analysis was conducted on $\mathsf{DP_{WCC}}$ and $\mathsf{DPFH_{WCC}}$ but since the results are similar in terms of plots and behavior, we show only the results obtained for $\mathsf{DP_{WCC}}$ measurements.

Multithreading. For analyzing parallelization and multithreading we have used *ThreadScope* Tool which allows us to see how the parallelization is taking place on GHC at a fine grained level and how the threads are distributed throughout the different cores requested with the `-N` execution `ghc-option` flag.

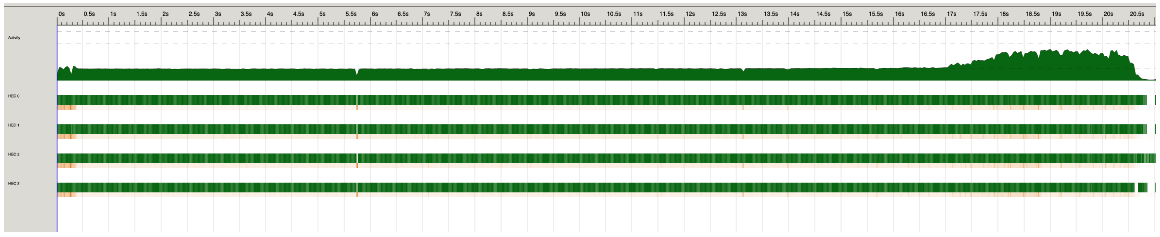


Figure 17: Threadscope Image of General Execution

In Figure 17, we can see that the parallelization is being distributed evenly among the 4 Cores that we have set for this execution. The distribution of the load is more intensive at the end of the execution, where `actor2` filter stage of the algorithm is taking place and different filters are reaching execution of that second actor.

Another important aspect shown in Figure 17, is that this work is not so significant for GHC and the threads and distribution of the work keeps between 1 or 2 cores during the execution time of the `actor1`. However, the usages increase on

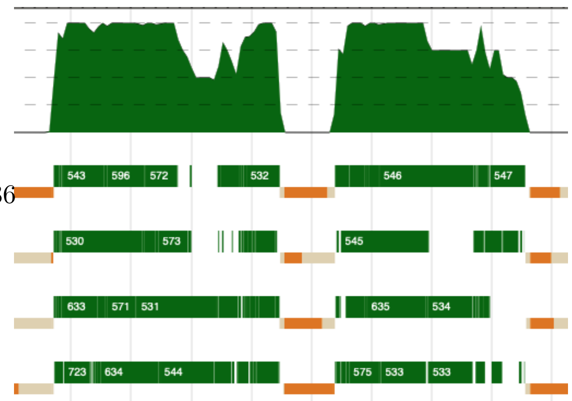


Figure 18: Threadscope Image of Zoomed Fraction

the second actor as pointed out before. In this regard, we can answer research questions [Q1] and [Q3] (section 6), verifying that Haskell not only supports the required parallelization level but is evenly distributed across the program execution too.

Finally, it can also be appreciated that there is no sequential execution on any part of the program because the 4 cores have *CPU* activity during the whole execution time. This is because as long the program start, and because of the nature of the DPP model, it is spawning the *Source* stage in a separated thread. This is a clear advantage for the model and the processing of the data since the program does not need to wait to do some sequential processing like reading a file, before start computing the rest of the stages.

Figure 18 zooms in on *ThreadScope* output in a particular moment, approximately in the middle of the execution. We can appreciate how many threads are being spawned and by the tool and if they are evenly distributed among cores. The numbers inside green bars represent the number of threads that are being executed on that particular core (horizontal line) at that execution slot. Thus, the number of threads varies among slot execution times because, as it is already known, GHC implements *Preemptive Scheduling* [15].

Having said that, it can be appreciated in Figure 18 our first assumption that the load is evenly distributed because the mean number of executing threads per core is 571.

Memory allocation. Another important aspect in our case is how the memory is being managed to avoid memory leaks or other non-desired behavior that increases memory allocation during the execution time. This is even more important in the particular implementation of WCC using DPP model because it requires to maintain the set of connected components in memory throughout the execution of the program or at least until we can output the calculated WCC if we reach to the last *Filter* and we know that this WCC cannot be enlarged anymore.

In order to verify this, we measure memory allocation with *eventlog2html* which converts generated profiling memory eventlog files into graphical HTML representation.

As we can see in Figure 19, $\mathsf{DP}_{\mathsf{WCC}}$ does an efficient work on allocating memory since we are not using more than 57 MB of memory during the whole execution of the program.

On the other hand, if we analyze how the memory is allocated during the execution of the program, it can also be appreciated that most of the memory is allocated at the beginning of the program and steadily decrease over time with a small peak at the end that does not overpass even half of the initial peak of 57 MB. The explanation for this behavior is quite straightforward because at the beginning we

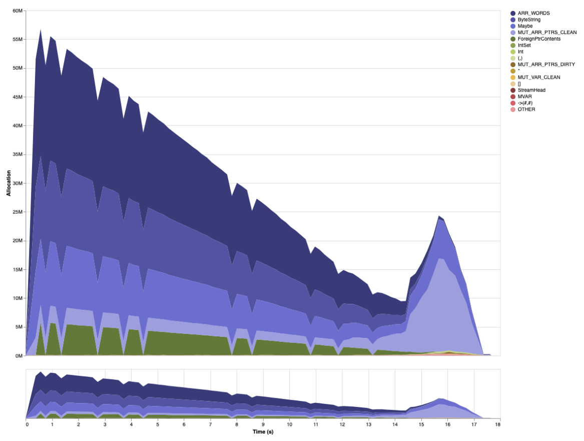


Figure 19: Memory Allocation

are reading from the file and transforming a `ByteString` buffer to `(Int, Int)` edges. This is seen in the image in which the dark blue that is on top of the area is `ByteString` allocation. Light blue is allocation of `Maybe a` type which is the type that is returned by the *Channels* because it can contain a value or not. Data value `Nothing` is indicating end of the *Channel*.

Another important aspect is the green area which represents `IntSet` allocation, which in the case of our program is the data structure that we use to gather the set of vertices that represents a WCC. This means that the amount of memory used for gathering the WCC itself is minimum, and it is decreasing over time, which is another empirical indication that we are incrementally releasing results to the user. It can be seen as well that as long the green area reduces the lighter blue (`MUT_ARR_PTRS_CLEAN`[33]) increases at the same time indicating that the computations for the output (releasing results) is taking place.

Finally, according to what we have stated above, we can answer the question [Q3] (section 6) showing that not only memory management was efficient, but at the same time, the memory was not leaking or increasing across the running execution program.

The empirical evaluation of $\mathsf{DP}_{\mathsf{WCC}}$ evidences suitability, and robustness of this language to of support the development of a Dynamic Pipeline Framework. Measuring using diefficienc metrics reveals some advantageous capability of $\mathsf{DP}_{\mathsf{WCC}}$ implementation to deliver incremental results compared with default containers library implementation. Regarding the main aspects where DPP is strong, i.e., pipeline parallelism and time processing, the $\mathsf{DP}_{\mathsf{WCC}}$ performance shows that Haskell can deal with the requirements for the WCC problem without penalizing neither execution time nor memory allocation. In particular, the $\mathsf{DP}_{\mathsf{WCC}}$ implementation outperforms in those cases where the topology of the graph is sparse and where the number of vertices in the largest WCC is not big enough.

To conclude, $\mathsf{DP}_{\mathsf{WCC}}$ has gathered enough evidence to show that the implementation of a Dynamic Pipeline solutions in Haskell Programming Language is feasible. This fact opens a wide range of algorithms to be explored using the Dynamic Pipeline Paradigm, supported by purely functional programming language. This results give us insights about how to proceed for implementing a first version of a DPF using (parallel) Haskell.

7. Conclusions and Ongoing Work

The empirical evaluation of the $\mathsf{DP}_{\mathsf{WCC}}$ implementation to compute weakly connected components of a graph, evidence suitability, and robustness to provide a Dynamic Pipeline Framework in that language. Measuring using `dief@t` metrics in **??** reveals some advantageous capability of $\mathsf{DP}_{\mathsf{WCC}}$ implementation to deliver incremental results compared with default containers library implementation. Regarding the main aspects where DPP is strong, i.e. pipeline parallelism and time processing, the $\mathsf{DP}_{\mathsf{WCC}}$ performance shows that Haskell can deal with the requirements for the WCC problem without penalizing neither execution time nor memory allocation. In particular, the $\mathsf{DP}_{\mathsf{WCC}}$ implementation outperforms in those cases where the topology of the graph is more sparse and where the number of vertices in the largest WCC is not big enough. We think this work has gathered enough evidence to show that the implementation of Dynamic Pipeline in Haskell

[33]GHC-Exts-Heap-ClosureTypes.html

Programming Language is feasible. This fact opens a wide range of algorithms to be explored using the Dynamic Pipeline Paradigm, supported by purely functional programming language. As we mentioned in section **??**, we are addressing the design and the definition of the DSL in Haskell, taking into account the knowledge obtained in this work. The complete DPF's implementation will contain the *Type-Level* DSL allowing the user to define algorithms in terms of DP and the Interpreter of DSL (IDL) that will be mainly based on what has been presented here.

Bibliography

[1] M. Acosta, M.-E. Vidal, and Y. Sure-Vetter. Diefficiency metrics: measuring the continuous efficiency of query processing approaches. In *International Semantic Web Conference*, pages 3–19. Springer, 2017.

[2] M. M. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. *ACM SIGPLAN Notices*, 40(9):241–253, 2005.

[3] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174, 2001.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] M. Fowler. *Domain-specific languages.* Pearson Education, 2010.

[6] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review*, 40(5):151–162, 2006.

[7] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

[8] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, 2004.

[9] W. A. Howard. The formulae-as-types notion of construction. In H. Curry, H. B., S. J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[10] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196–es, 1996.

[11] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 96–107, 2004.

[12] O. Kiselyov, S. P. Jones, and C.-c. Shan. Fun with type functions. In *Reflections on the Work of CAR Hoare*, pages 301–331. Springer, 2010.

[13] T. Kosar, S. Bohra, and M. Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

[14] I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, Z. Zhang, and J. Sukha. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing (TOPC)*, 2(3):1–42, 2015.

[15] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach. Lightweight concurrency primitives for ghc. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 107–118, 2007.

[16] S. Marlow. Parallel and concurrent programming in Haskell. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *CEFP 2011*, volume 7241 of *LNCS*, pages 339–401. O'Reilly Media, Inc., 2012.

[17] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 65–78, 2009.

[18] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: better strategies for parallel haskell. *ACM Sigplan Notices*, 45(11):91–102, 2010.

[19] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. *ACM SIGPLAN Notices*, 46(12):71–82, 2011.

[20] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 124–144, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-47599-6.

[21] E. Pasarella, M.-E. Vidal, and C. Zoltan. Comparing mapreduce and pipeline implementations for counting triangles. *Electronic proceedings in theoretical computer science*, 237:20–33, 2017.

[22] S. Peyton-Jones and E. Meijer. Henk: a typed intermediate language. *TIC*, 97:10, 1997.

[23] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)*, 52:1 – 37, 2019.

[24] J. P. R. Sales, E. Pasarella, C. Zoltan, and M.-E. Vidal. Towards a dynamic pipeline framework implemented in (parallel) haskell. In *PROLE2021*. SISTEDES, 2021. URL `http://hdl.handle.net/11705/PROLE/2021/017`.

[25] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66, 2012.

[26] C. Zoltan, E. Pasarella, J. Araoz, and M.-E. Vidal. The Dynamic Pipeline Paradigm. In *PROLE2019*. SISTEDES, 2019. URL `http://hdl.handle.net/11705/PROLE/2019/017`.

[27] M. Acosta. Python diefficiency metric tool. `https://github.com/SDM-TIB/diefpy/`. Accessed: 2021-05-03.

[28] Stanford University. Snap datasets: Stanford large network dataset collection. `https://snap.stanford.edu/data/index.html`. Accessed: 2021-04-15.

Appendix A. Summary of URL references

bytestring 0.10.12.0	https://hackage.haskell.org/package/bytestring
Data.Graph-containers 0.6.2.1	https://hackage.haskell.org/package/containers
relude 1.0.0.1	https://hackage.haskell.org/package/containers
unagi-chan 0.4.1.3	https://github.com/jberryman/unagi-chan
NoImplicitPrelude	https://downloads.haskell.org/ghc/8.8.4/docs/html/users_guide/
stack version 2.5.1	https://docs.haskellstack.org/en/stable/README/
stack build	https://github.com/jproyo/upc-miri-tfm/tree/main/connected-comp
dynamic-pipeline 0.3.2.0	https://hackage.haskell.org/package/dynamic-pipeline