GRAFS: Declarative Graph Analytics*

FARZIN HOUSHMAND, University of California, Riverside, USA
MOHSEN LESANI, University of California, Riverside, USA
KEVAL VORA, Simon Fraser University, Canada

Graph analytics elicits insights from large graphs to inform critical decisions for business, safety and security. Several large-scale graph processing frameworks feature efficient runtime systems; however, they often provide programming models that are low-level and subtly different from each other. Therefore, end users can find implementation and specially optimization of graph analytics error-prone and time-consuming. This paper regards the abstract interface of the graph processing frameworks as the instruction set for graph analytics, and presents GRAFS, a high-level declarative specification language for graph analytics and a synthesizer that automatically generates efficient code for five high-performance graph processing frameworks. It features novel semantics-preserving fusion transformations that optimize the specifications and reduce them to three primitives: reduction over paths, mapping over vertices and reduction over vertices. Reductions over paths are commonly calculated based on push or pull models that iteratively apply kernel functions at the vertices. This paper presents conditions, parametric in terms of the kernel functions, for the correctness and termination of the iterative models, and uses these conditions as specifications to automatically synthesize the kernel functions. Experimental results show that the generated code matches or outperforms handwritten code, and that fusion accelerates execution.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Formal language definitions**; **Source code generation**; **Correctness**.

Additional Key Words and Phrases: Program Synthesis, Fusion

ACM Reference Format:
Farzin Houshmand, Mohsen Lesani, and Keval Vora. 2021. GRAFS: Declarative Graph Analytics. *Proc. ACM Program. Lang.* 5, ICFP, Article 83 (August 2021), 32 pages. https://doi.org/10.1145/3473588

1 INTRODUCTION

Large-scale *graph analytics* has recently gained popularity due to its growing applicability across various important domains including social networks, market influencer analysis, bioinformatics, criminology, and machine learning and data mining. Several large-scale graph processing systems [Gonzalez et al. 2012; Malewicz et al. 2010; Mariappan et al. 2021; Mariappan and Vora 2019; Roy et al. 2013; Shun and Blelloch 2013; Vora 2019; Zhang et al. 2018; Zhu et al. 2016, 2015] have been developed to enable efficient graph analysis across shared memory and distributed platforms. Their programming models often require graph analysis problems to be expressed in terms of low-level kernel functions over vertices and edges. However, analyses over graphs are best expressed using *higher-level abstractions* such as reduction over paths in the graph. For instance, shortest path, reachability and connected component problems are fundamentally formulated in terms of

*This work is supported by National Science Foundation grants 1942711, 1718997 and 1910878.

Authors' addresses: Farzin Houshmand, University of California, Riverside, USA, fhous001@cs.ucr.edu; Mohsen Lesani, University of California, Riverside, USA, lesani@cs.ucr.edu; Keval Vora, Simon Fraser University, Canada, keval@sfu.ca.

2475-1421/2021/8-ART83
https://doi.org/10.1145/3473588

paths. Further, elaborate graph analysis problems that involve multiple reductions over paths or vertices are difficult to correctly implement using the offered low-level programming models. More importantly, manual *optimizations* such as merging multiple iterations can be time-consuming and error-prone. In particular, showing *correctness and termination* properties requires reasoning about the flow of values between vertices across multiple iterations that emulate values for paths.

This project regards the interface of the graph processing frameworks as the instruction set for graph analytics, and introduces Grafs, a graph analytics language and synthesizer. The Grafs language is a *high-level declarative specification language* that provides features for common graph processing idioms such as reduction over paths. We show that the declarative language can easily and concisely capture the common graph analysis problems. Given a specification, the Grafs synthesizer *automatically synthesizes code* for five graph processing frameworks: Ligra [Shun and Blelloch 2013], GridGraph [Zhu et al. 2015], PowerGraph [Gonzalez et al. 2012], Gemini [Zhu et al. 2016], and GraphIt [Zhang et al. 2018].

To synthesize efficient implementations, Grafs optimizes specifications by syntactic *fusion transformations* that fuse similar operations to be executed together. We formalize the syntax and the semantics of the Grafs language and the fusion rules, and prove that the fusion transformations are semantics-preserving. Effectively, fusion reduces specifications to the sequence of three primitives: reduction over paths, mapping over vertices and reduction over vertices.

Graph analytics frameworks offer *iterative programming models* to calculate reduction over paths. The values for vertices or edges are calculated iteratively based on the values of neighbors. Influenced by their runtime systems, these frameworks differ on how values are propagated between iterations. Some allow computations to both pull and push values to neighbors [Gonzalez et al. 2012] whereas others only allow push [Zhu et al. 2015] and others support a hybrid [Shun and Blelloch 2013; Zhang et al. 2018; Zhu et al. 2016]. Not only the propagation methods, but also *system-specific* nuances of the frameworks make their implementation of the same analysis problem subtly different. For example, they follow different protocols for atomicity of updates.

We formalize a comprehensive set of *iterative models* that given certain kernel functions, calculate path-based reductions. For each model, we present *correctness and termination conditions* on candidate kernel functions. Given a path-based reduction, the Grafs synthesizer enumerates candidate kernel functions and uses the correctness conditions as specifications to automatically *synthesize the kernel functions*. After fusion reduces specifications to the three primitives, reduction over paths, mapping over vertices and reduction over vertices, the synthesizer reduces reductions over paths to iterative calculations. Thus, graph analysis is reduced to *iteration-map-reduce primitives*. Grafs translates each of these primitives to implementations in each of the five target frameworks.

We apply Grafs to common graph analysis use-cases and generate code for each of the five frameworks. We note that graph processing frameworks often offer a more flexible and expressive API. However, we show that Grafs can express a large collection of common use-cases. The experimental results show that Grafs concisely captures use-cases, efficiently analyzes and synthesizes code, and its fusion brings up to 4× and in average 2.4× speedup.

In summary, this paper makes the following contributions. It provides the high-level declarative language Grafs and its semantics for large-scale graph analytics. It captures the iteration-map-reduce graph processing primitives that implement graph computations as structured let terms. Grafs presents semantics-preserving fusion transformations that are aware of these primitives: they fuse computation into and maintain this structure. This paper formally models and proves the formal correctness and termination conditions for a comprehensive set of iterative models. Further, it combines type-directed enumerative synthesis and constrained-based synthesis to automatically synthesize the iterative kernel functions. The resulting tool can target five different graph processing frameworks and is evaluated on multiple standard benchmarks. The experiments show that fusion

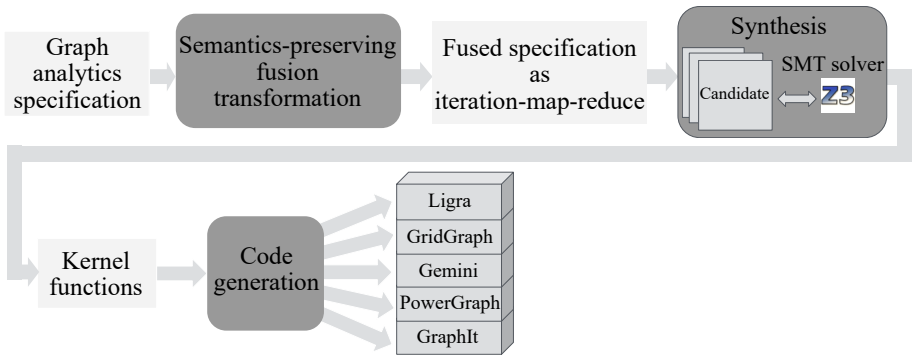


Fig. 1. Workflow of GRAFS (Graph Analytics Fusion and Synthesis)

can accelerate execution. GRAFS showcases that declarative languages and fusion transformations are effective for large-scale analytics.

In the following sections will present (1) Graph analytics specification language GRAFS and its semantics (§ 3 and § 5.1), (2) Semantics-preserving and platform-independent fusion transformations (§ 5.2), (3) The formalization of iterative graph computation models (§ 4), their correctness and termination conditions (§ 6.1), and synthesis of their kernel functions (§ 6.2), and (4) The synthesis tool that generates code for five graph processing frameworks and its experimental results (§ 7).

2 OVERVIEW

Fig. 1 shows the overview of GRAFS. The user writes her graph analytics as a declarative specification. Then, the fusion transformations optimize the input specification and translates it to iteration-map-reduce primitives. Subsequently, the synthesis process generates iterative kernel functions for the optimized specification. Finally, the code generation backend translates the kernels to five target graph processing frameworks. In this section, we present an example use-case in the GRAFS specification language, and then show how that specification can be fused into an equivalent more efficient and canonical specification. Then, we illustrate iterative reduction models (i.e., algorithms). Next, we consider the iteration-map-reduce primitives, and see both fused and unfused implementations of our use-case based on them. Finally, we see a glimpse of the correctness conditions of iterative reductions and how a synthesizer can use the conditions to generate the iterative kernel functions.

Specification. The GRAFS language allows declarative and concise specification of graph analysis computations. For example, Fig. 2, Eq. 1 represents the specification of the RADIUS use-case. The radius of a graph is the minimum eccentricity over its vertices. The eccentricity of a source vertex s is the longest of the shortest paths from s to any other vertex. The inner-most reduction of the RADIUS use-case is a *path-based reduction* that calculates the shortest path from a source vertex s to a destination vertex v. It applies the minimum reduction function min to the result of applying the weight function weight to all paths p in $\mathsf{Paths}(s, v)$, that is the set of paths from s to v. Then, it specifies the eccentricity of s as a nesting *vertex-based reduction* with the reduction function max to find the longest of the shortest paths over all destination vertices v. Finally, it specifies the radius as the minimum of the eccentricity of the sample sources s_1 and s_2.

Fusion. A naive execution of specifications may execute path-based and vertex-based reductions multiple times. We show that multiple such reductions can be fused into a single reduction, and

specifications can be represented as a common *triple-let form* with separate terms for path-based reduction, mapping over vertices and vertex-based reduction.

For example, the Radius use-case includes multiple path-based reductions one per source that can be fused together. Further, the path-based reductions are enclosed by vertex-based reductions that can be fused together as well. We illustrate this fusion in Fig. 2. We consider the fusion steps in turn. The specification of Radius is represented in Eq. 1. In Eq. 2, the outer min function over the two sources is unrolled. In Eq. 3, we restate each of the two reductions in a triple-let form. Grafs features a triple-let term that separates path-based reductions, mapping over vertices and vertex-based reductions, and thus, facilitates fusion. The term $\max_{v \in V} \min_{p \in \mathsf{Paths}(s_1, v)} \mathsf{length}(p)$ is rewritten as the following three lets. The first let, ilet $x := \min_{s_1}$ length, calculates a path-based reduction. For each vertex, it calculates the shortest length over the paths from the source s_1, and binds the result to x. The second let applies a map function in each vertex on the results of the path-based reduction. In this case, there is only one path-based reduction; therefore, the map function in the second let, mlet $x' := x$, is simply the identity function, and the result is bound to x'. (In use-cases with an expression on multiple path-based reductions, the map in the second let captures the expression.) The third let calculates a reduction over all vertices. In this example, the third let, rlet $x'' := \max\ x'$, calculates the maximum value over all vertices and binds the result to x''. In Eq. 3, a similar transformation is applied for the other source s_2 as well.

Next, in Eq. 4, the two triple-let terms are fused into one by pairing the operations of the corresponding lets, and the outer min is applied to the two final results x'' and y''. In the next two steps, the paired path-based and vertex-based reductions are fused. In Eq. 5, the two path-based reductions of the first let, ilet, are fused into one. The fused reduction calculates the pair of the two values simultaneously. (The two sources s_1 and s_2 are used to initialize the first and second elements of the pairs respectively.) The fused path function $\mathcal{F}$ returns the pair of the results of the two path functions. Similarly, the fused reduction function $\mathcal{R}$ applies the two reduction functions to the first and second elements of the input pairs respectively. Finally, in Eq. 6, the pair of vertex-based reductions of the third let, rlet, are fused into one. The fused reduction function $\mathcal{R}'$ applies the two reduction functions to the first and second elements of the input pairs respectively. The original Radius specification executes two rounds of path-based and vertex-based reductions; however, the fused version computes one path-based and one vertex-based reduction on tuples of two elements at the same time. We will see the formal fusion rules including rules for more elaborate terms such as nested path-based reductions (§ 5.2 and § 5.3).

The final term represents the specification of Radius as an equivalent sequence of one path-based reduction, one map in each vertex, and one reduction over all vertices. We will next see that path-based reductions are calculated iteratively. Thus, fusion reduces Grafs specifications to three primitives: *Iteration-Map-Reduce*: iteration for iterative path-based reduction, map for mapping over vertices and reduce for reduction over vertices. Map and reduce over vertices can be directly implemented; next, we consider iterative path-based reductions.

Iterative Path-based Reduction. Calculating path-based reductions by explicit enumeration of paths is prohibitively inefficient. Instead, path-based reductions are calculated iteratively by local updates on the value of vertices based on the values of their neighbors. As an example, we consider the pull iterative model for idempotent reduction functions. Let us consider the shortest path use-case $\mathrm{SSSP}(s)(v) = \min_{p \in \mathsf{Paths}(s,v)} \mathsf{weight}(p)$. It specifies a path-based reduction from the source s where the reduction function $\mathcal{R}$ is min and the path function $\mathcal{F}$ is weight. (We saw a similar reduction, the shortest length, as the innermost reduction of Radius.)

$$\textsc{Radius} = \min_{s\in\{s_1,s_2\}} \max_{v\in V} \min_{p\in \text{Paths}(s,v)} \text{length}(p) \tag{1}$$

$$= \min\left(\max_{v\in V} \min_{p\in \text{Paths}(s_1,v)} \text{length}(p),\ \max_{v\in V} \min_{p\in \text{Paths}(s_2,v)} \text{length}(p)\right) \tag{2}$$

$$= \min\left(\left(\begin{array}{l}\text{ilet } x := \min\limits_{s_1} \text{ length in}\\ \text{mlet } x' := x \text{ in}\\ \text{rlet } x'' := \max\ x' \text{ in}\\ x''\end{array}\right), \left(\begin{array}{l}\text{ilet } y := \min\limits_{s_2} \text{ length in}\\ \text{mlet } y' := y \text{ in}\\ \text{rlet } y'' := \max\ y' \text{ in}\\ y''\end{array}\right)\right) \tag{3}$$

$$= \left(\begin{array}{l}\text{ilet } \langle x, y\rangle := \langle \min\limits_{s_1} \text{length}, \min\limits_{s_2} \text{length}\rangle \text{ in}\\ \text{mlet } \langle x', y'\rangle := \langle x, y\rangle \text{ in}\\ \text{rlet } \langle x'', y''\rangle := \langle \max\ x', \max\ y'\rangle \text{ in}\\ \min(x'', y'')\end{array}\right) \tag{4}$$

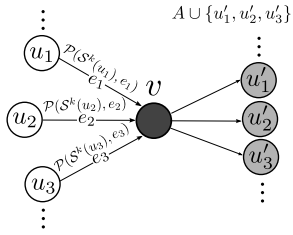
$$= \left(\begin{array}{l}\text{ilet } \langle x, y\rangle := \mathop{\mathcal{R}}\limits_{\langle s_1,s_2\rangle} \mathcal{F} \text{ in}\\ \text{mlet } \langle x', y'\rangle := \langle x, y\rangle \text{ in}\\ \text{rlet } \langle x'', y''\rangle := \langle \max\ x', \max\ y'\rangle \text{ in}\\ \min(x'', y'')\end{array}\right) \text{ where } \begin{array}{l}\mathcal{F} := \lambda p.\ \langle \text{length}(p), \text{length}(p)\rangle\\ \mathcal{R}\ (\langle a, b\rangle, \langle a', b'\rangle) :=\\ \quad \langle \min(a, a'), \min(b, b')\rangle\end{array} \tag{5}$$

$$= \left(\begin{array}{l}\text{ilet } \langle x, y\rangle := \mathop{\mathcal{R}}\limits_{\langle s_1,s_2\rangle} \mathcal{F} \text{ in}\\ \text{mlet } \langle x', y'\rangle := \langle x, y\rangle \text{ in}\\ \text{rlet } \langle x'', y''\rangle := \mathcal{R}'\ \langle x', y'\rangle \text{ in}\\ \min(x'', y'')\end{array}\right) \text{ where } \begin{array}{l}\mathcal{R}'\ (\langle a, b\rangle, \langle a', b'\rangle) :=\\ \quad \langle \max(a, a'), \max(b, b')\rangle\end{array} \tag{6}$$

Fig. 2. Fusion of the RADIUS Use-case.

```
def Iteration-Pull (I, P, R)
  foreach (v ∈ V)
    S(v) ← I(v)
    if (S(v) ≠ ⊥) A ← A ∪ {v}
  while (A ≠ ∅)
    foreach (v ∈ A)
      gv ← ⊥
      foreach (⟨u, v⟩ ∈ in-edges(v))
        gv ← R(gv, P(S(u), ⟨u, v⟩))
      nv ← R(S(v), gv)
      if (nv ≠ S(v))
        S(v) ← nv
        foreach (⟨v, u'⟩ ∈ out-edges(v))
          A' ← A' ∪ {u'}
    A, A' ← A', ∅
```

(a)



(b)

```
def Map (f)
  foreach (v ∈ V)
    S(v) ← f(S(v)))
```

```
def Reduce (R)
  val ← ⊥
  foreach (v ∈ V)
    val ← R(val, S(v))
  return val
```

(c)

Fig. 3. Pull Iterative Reduction.

The pull-based iterative reduction is presented in Fig. 3a and illustrated in Fig. 3b. Each vertex stores a value $\mathcal{S}(v)$; we denote the value of a vertex v in the iteration k as $\mathcal{S}^k(v)$. The iterative calculation is based on three kernel function: the initialization function $\mathcal{I}$, the propagation function $\mathcal{P}$ and the reduction function $\mathcal{R}$. The function $\mathcal{I}$ is a function from vertices to their initial value. For the SSSP use-case, $\mathcal{I}$ is $\lambda v.$ if $(v = s)$ 0 else $\bot$ that initializes the value of the source s to zero

```
RADIUS (Unfused):
    Iteration(λv. if (v = s1) 0 else ⊥,
              λn, e. n + 1,
              min)
    ecc1 ← Reduce(max)
    Iteration(λv. if (v = s2) 0 else ⊥,
              λn, e. n + 1,
              min)
    ecc2 ← Reduce(max)
    radius ← min(ecc1, ecc2)
```

(a)

```
RADIUS (Fused):
    Iteration(λv. ⟨if (v = s1) 0 else ⊥,
                  if (v = s2) 0 else ⊥⟩,
              λn, e. ⟨n + 1, n + 1⟩,
              λ⟨a, b⟩, ⟨a′, b′⟩.
                  ⟨min(a, a), min(b, b′)⟩)
    ⟨ecc1, ecc2⟩ ← Reduce(λ⟨a, b⟩, ⟨a′, b′⟩.
                          ⟨max(a, a′), max(b, b′)⟩)
    radius ← min(ecc1, ecc2)
```

(b)

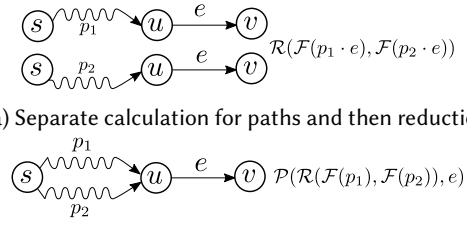
Fig. 4. Unfused and fused implementations of RADIUS as iteration-map-reduce rounds.

(the some value of zero to be more precise) and the other vertices to none $\perp$. In each iteration, if the value of a vertex changes, its successors are added to the active set A for the next iteration. Fig. 3b shows the calculations for the active vertex v that is shown in black. In an iteration $k+1$, an active vertex v pulls the value $\mathcal{S}^k(u)$ of each of its predecessors u. For each predecessor u, it applies the propagation function $\mathcal{P}$ to the value $\mathcal{S}^k(u)$ and the edge $\langle u, v\rangle$. For the SSSP use-case, the function $\mathcal{P}$ is $\lambda n, e.\ n + \text{weight}(e)$ that adds the value of the predecessor to the weight of the edge from it. It then applies $\mathcal{R}$ (that is min in SSSP) to reduce the propagated values together and with the current value $\mathcal{S}^k(v)$ of v. The result is the new value $\mathcal{S}^{k+1}(v)$ of v. If the value of v changes, the successors of v that are marked as gray are active in the next iteration. The calculation stops when the values of vertices stay unchanged in two consecutive iterations. We will formalize a comprehensive set of iterative models (§ 4).

Iteration-Map-Reduce. After the fusion transformation, the specification reduces to *iteration-map-reduce primitives*: reduction over paths, mapping over vertices and reduction over vertices. We saw in Fig. 3a and b that reductions over paths can be computed using the iterative models. A sketch of both Map and Reduce operations is shown in Fig. 3c. The Map operation simply goes over all the vertices in the graph and applies the input function f to the value of each vertex. Similarly, the Reduce operation reduces vertex values with the input reduction function $\mathcal{R}$. Fig. 4 shows sketches for both unfused and fused implementation of the RADIUS use-case. Fig. 4a shows the translation of the original unfused specification of the RADIUS use-case that we saw in Fig. 2, Eq. 1. In contrast, Fig. 4b shows the translation of the final fused specification of the RADIUS use-case that we saw in Fig. 2, Eq. 6. (We note that since the map is the identity function for the RADIUS use-case, it is elided in these implementations.) The unfused version in Fig. 4a executes two rounds of iteration-map-reduce. The first and the second rounds perform calculations for the first source s_1 and the second source s_2 respectively. Each round first performs an iteration to calculate the shortest path from the source to each vertex. Then, it calculates the eccentricity of the source by applying the max reduction function on the values of all vertices. Finally, the radius of the graph is minimum of the two eccentricity values. The fused version in Fig. 4b performs one round of iteration-map-reduce. The fused iteration stores and performs operations on a pair of values: shortest paths to each of the two sources. The initialization function initializes the values of the two sources, and the propagation and reduce functions propagate and reduce the shortest path values to them at the same time. Similarly, the subsequent reduction over all vertices calculates the eccentricity of the two sources at the same time. As we will see in the experiments (§ 7), this reduces the computation load by a factor of two.

Correctness and Synthesis. We formalize *correctness and termination conditions* for calculation of path-based reductions based on the iterative models. The conditions are parametric in terms of the kernel initialization and propagation functions and are used to automatically synthesize the kernel functions (§ 6.1 and § 6.2).

We present and prove sufficient conditions for a comprehensive set of iterative models (§ 6.1). As an example, we consider the pull model and illustrate one of the correctness conditions on the propagation function $\mathcal{P}$ in Fig. 5a and Fig. 5b. Consider a vertex v and a predecessor u of v. Consider calculating the reduction over all the paths to v that go through u. Fig. 5a shows the direct calculation where the value of the path function for each path to v is separately calculated, and then the results are reduced. On the other hand, Fig. 5b shows a calculation using the propagation function $\mathcal{P}$ where first, the values of the path function for the paths to the predecessor u are calculated and reduced, and then, the result is propagated by $\mathcal{P}$ to v. In order to correctly calculate path-based reductions by local updates, the result of the above two calculations should be the same. Intuitively, local propagations from predecessors should be equivalent to global reductions over paths. Further, to reason about termination, we formalize the termination conditions for iterative models. Iterations incrementally consider longer paths. Cycles of a graph generate an infinite number of paths and can cause divergence. However, under certain conditions on the reduction and path functions $\mathcal{R}$ and $\mathcal{F}$, adding longer paths has no effect on the vertex values. For example, for the shortest path use-case SSSP (with non-negative edges), after a certain number of iterations, all the simple paths of the graph are already considered, and longer cyclic paths cannot improve the shortest path. Therefore, the calculation eventually terminates. We will see a formal definition of this condition and prove that it is sufficient for termination.



(a) Separate calculation for paths and then reduction

(b) Reduction at predecessor and then propagation

Fig. 5. The Correctness of the Pull Model. The path $p \cdot e$ denotes the extension of path p with edge e.

We use the correctness conditions to *synthesize correct kernel functions* (§ 6.2). In particular, we apply type-guided enumerative synthesis to find candidates, and automatic solvers to check the validity of the correctness conditions for each candidate. The result is correct-by-construction kernel functions that can iteratively calculate path-based reductions. We translate the synthesized functions to code in five high-performance graph processing frameworks (§ 7).

3 DECLARATIVE GRAPH ANALYTICS

The GRAFS language declaratively and concisely captures *mathematical specifications of graph analysis computations*. The language design is guided by common idioms in graph processing use-cases. It supports reduction over values of paths to a vertex, and mapping and reduction over those values. Fig. 6 presents example use-cases. More use-cases are available in the appendix § 1.

Path-based Reductions. The use-case SSSP specifies the weight of the shortest path from the source vertex s to each vertex v. The set of paths from a source vertex s to a destination vertex v is denoted by $\mathsf{Paths}(s, v)$. The specification applies the minimum reduction function min to the result of applying the weight function weight to all paths p in $\mathsf{Paths}(s, v)$. The specification of connected component (for undirected graphs) CC takes the smallest identifier of the vertices in a component as the representative identifier of that component. The set of all paths (from any source vertex) to a destination vertex v is denoted by $\mathsf{Paths}(v)$. The specification CC defines the connected component of each vertex v as the minimum identifier of the head vertices of the paths $\mathsf{Paths}(v)$. The above two specifications apply a reduction function $\mathcal{R}$ to the result of a path function $\mathcal{F}$ for a set of paths.

We call these reductions *path-based reductions*. Similarly, the breadth-first-search use-case BFS calculates the parent for each vertex in the BFS tree rooted at a source vertex s. For each vertex v, it specifies a path-based reduction to find the shortest-length path from s to v, and returns the penultimate of that path. The penultimate of a path is the vertex before the last in the path. The specification uses the reduction function arg min to get the path with the minimum length rather than the minimum length itself, and then applies the penultimate function to the path. (A simpler specification can simply apply min instead of arg min and return the minimum path length, i.e., the depth of the vertex in the breadth-first-search tree.)

Nested Path-based Reductions. Path-based reductions can be nested. The use-case WSP specifies the widest shortest path from a source s to each vertex v. We use the let syntactic sugar to enhance readability. WSP has a nested reduction (with the reduction function args min) to find the shortest paths, and then a nesting reduction to find the widest capacity in those paths. WSP is used as a metric of the trust of a user to other users in social networks where the capacity of each edge is the local trust rating of the source user to the sink user [Golbeck 2005]. Intuitively, users with wider (stronger trust ratings) and shorter (closer) paths are more trustworthy sources of information. Similarly, the use-case NSP specifies the number of shortest paths from a source s to each vertex v. It uses a nested reduction to find the shortest paths and then applies the cardinality operator to the resulting set. (We will see in § 5.3 that cardinality is a syntactic sugar for a path-based reduction with the sum $\sum$ function.)

$$
\begin{aligned}
\text{SSSP}(s)(v) &= \min_{p \in \text{Paths}(s,v)} \text{weight}(p) \\
\text{CC}(v) &= \min_{p \in \text{Paths}(v)} \text{head}(p) \\
\text{BFS}(s)(v) &= \text{penultimate}(\ \arg\min_{p \in \text{Paths}(s,v)} \text{length}(p)) \\
\text{WSP}(s)(v) &= \text{let } P := \arg\min_{p \in \text{Paths}(s,v)} \text{length}(p) \text{ in} \\
&\quad \max_{p \in P} \text{capacity}(p) \\
\text{NSP}(s)(v) &= \left| \arg\min_{p \in \text{Paths}(s,v)} \text{weight}(p) \right| \\
\text{NWR}(s)(v) &= \frac{\min_{p \in \text{Paths}(s,v)} \text{capacity}(p)}{\max_{p \in \text{Paths}(s,v)} \text{capacity}(p)} \\
\textsc{Trust}(v) &= \max_{s \in \overline{s}} \left(\frac{\max_{p \in \text{Paths}(s,v)} \text{capacity}(p)}{\min_{p \in \text{Paths}(s,v)} \text{length}(p)} \right) \\
\textsc{Radius} &= \min_{s \in \{\overline{s}\}} \max_{v \in V} \min_{p \in \text{Paths}(s,v)} \text{length}(p) \\
\text{DRR} &= \frac{\max_{s \in \{\overline{v}\}} \max_{v \in V} \min_{p \in \text{Paths}(s,v)} \text{length}(p)}{\min_{s \in \{\overline{v}\}} \max_{v \in V} \min_{p \in \text{Paths}(s,v)} \text{length}(p)} \\
\text{DS}(s) &= \bigcup_{v \in V \wedge \left(\min_{p \in \text{Paths}(s,v)} \text{weight}(p) \right) > 7} \{v\} \\
\textsc{LTrust}(s) &= \text{let SSSP} := \lambda s, v. \min_{p \in \text{Paths}(s,v)} \text{weight}(p) \text{ in} \\
&\quad \text{let WP} := \lambda s, v. \max_{p \in \text{Paths}(s,v)} \text{capacity}(p) \text{ in} \\
&\quad \min_{v \in V \wedge \text{ SSSP}(s,v) < \textsc{Radius}} \text{WP}(s, v)
\end{aligned}
$$

Fig. 6. A Subset of Use-cases in Grafs. SSSP: Single Source Shortest Path, CC: Connected Components, BFS: Breadth-First Search, WSP: Widest Shortest Path, NSP: Number of Shortest Paths, NWR: Narrowest to Widest Ratio, Trust: Trust from users $\{\overline{s}\}$, Radius: Radius sampled on vertices $\{\overline{s}\}$, DRR: Diameter to Radius Ratio, DS: Vertices with the distance of at least 7, LTrust: Least trust in the radius.

Mapping over Vertices. Mathematical operators can be applied to path-based reductions. The use-case NWR specifies the narrowest to widest path ratio from a source to each vertex. At each vertex, it divides the result of two path-based reductions. Similarly, the use-case Trust is the result of division and maximum operations between path-based reductions. It specifies the trust from a set of users $\{\overline{s}\}$ to each other user v. As before, wider and shorter paths are favored.

Vertex-based Reductions. The values of vertices calculated by a path-based reduction can be subsequently reduced by a *vertex-based reduction*. For example, the eccentricity of a source vertex s is the longest of the shortest paths

from s to any other vertex. The radius of a graph is the minimum eccentricity over its vertices. The Radius use-case specifies eccentricity as a vertex-based reduction with the reduction function max to find the longest of the shortest paths over all vertices. It then specifies the radius as the minimum of the eccentricity of a set of sample sources $\{\bar{s}\}$. Similar to path-based reductions, mathematical operators can be applied to vertex-based reductions. As the set of sampled sources $\{\bar{s}\}$ is finite, the outer min function can be unrolled to an infix operator between vertex-based reductions. Similarly, the use-case DRR, that is the ratio of the diameter over the radius of the graph, is specified as maximum and minimum operations between vertex-based reductions, and a subsequent division.

The use-case DS specifies the set of vertices with the distance of at least 7 from the source s. The union $\cup$ vertex-based reduction is used to calculate the set. The set of vertices that it is applied to are constrained by a nested path-based reduction to specify the distance. (In § 5.3, we show that *constrained vertex-based reductions* can be desugared to standard vertex-based reductions that are applied to path-based reductions on pairs of values.) The next use-case, LTrust, represents a measure of the least amount of trust from a user to her neighbourhood in a social network. Similar to DS, the use-case LTrust is specified as a constrained vertex-based reduction. Given a source s, it calculates the widest path to each vertex within the radius of s (i.e., k-hop neighbourhood of s where k is the radius of the graph), and then returns the narrowest of those.

4 ITERATIVE MODELS

We formalize four canonical models for iterative graph computations: the pull and push models with idempotent and non-idempotent reduction. Graph computation frameworks [Gonzalez et al. 2012; Malewicz et al. 2010; Roy et al. 2013; Shun and Blelloch 2013; Zhu et al. 2016, 2015] implement variants of these models. Later in § 6, we use these models to implement path-based reductions and present their correctness conditions.

In these models, each vertex is first initialized. Then, the value of each vertex is iteratively updated based on the values of its predecessors. In each iteration, the vertex pulls the values of its predecessors or each predecessor pushes its value to the vertex. Then, the values of the predecessors and the current value of the vertex are reduced to calculate the new value of the vertex. Before assigning the reduced value to the vertex, a final function may be applied to it. The iteration stops when the value of no vertex changes. The models are parametrized by four kernel functions: $\mathcal{I}$, $\mathcal{P}$, $\mathcal{R}$ and $\mathcal{E}$. The *initialization* function $\mathcal{I}$ defines the initial value for each vertex. The *propagation* function $\mathcal{P}$, given a value n and an edge $\langle u, v\rangle$ where n is the value of u, defines the value that is propagated to v. The commutative and associative *reduction* function $\mathcal{R}$ defines how the propagated values are aggregated. The *epilogue* function $\mathcal{E}$ defines the final update.

We present a high-level language to specify the kernel functions. We compile kernels specified in this language to executable programs in five graph processing frameworks. The grammar for the bodies of the kernel functions is presented in Fig. 7a. (Later in § 6.2, the same grammar is used by the synthesis process; given higher-level specifications, it automatically generates the kernel functions in this language.) Fig. 7b shows the iterative kernel functions for two example use-cases: the shortest path SSSP and the page-rank PageRank (PR). For the shortest path SSSP use-case, the initialization function $\mathcal{I}$ initializes the source vertex s to (some value of) 0 and the other vertices to none $\perp$. The propagation function $\mathcal{P}$ adds the value v of the predecessor to the weight of the edge e. The reduction function $\mathcal{R}$ is the minimum (that is idempotent) and the epilogue function $\mathcal{E}$ is the identity function. For the page-rank use-case PR, $\mathcal{I}$ divides the value 1 between the number of vertices $|V|$. The function $\mathcal{P}$ divides the value v of the predecessor between its successors. The function $\mathcal{R}$ is sum (that is non-idempotent). The function $\mathcal{E}$ multiplies the sum with the damping factor γ and adds a constant.

e	::=	$n \mid v$	Body Exp	
	|	$e+e \mid e-e \mid -e \mid \langle e,e\rangle$		
	|	$e\times e \mid e/e \mid e=e \mid e<e$		
	|	$\min(e,e) \mid \max(e,e)$		
	|	if (e) then e else e		
	|	$\text{weight}(e) \mid \text{capacity}(e)$		
	|	$\text{indeg}(e) \mid \text{outdeg}(e)$		
	|	$\text{src}(e) \mid \text{dst}(e)$		
	|	$\lvert V\rvert$	Graph Order	
n	::=	$0 \mid 1 \mid .. \mid \text{True} \mid \text{False}$	Literal	
v			Variable	

(a) Grammar

SSSP

$\mathcal{I} := \lambda v.\ \text{if } (v = s)\ 0 \text{ else } \bot$

$\mathcal{P} := \lambda v, e.\ v + \text{weight}(e)$

$\mathcal{R} := \lambda v, v'.\ \min(v, v')$

$\mathcal{E} := \lambda v.\ v$

PageRank (PR)

$\mathcal{I} := \lambda v.\ 1 / |V|$

$\mathcal{P} := \lambda v, e.\ v / \text{outdeg}(\text{src}(e))$

$\mathcal{R} := \lambda v, v'.\ v + v'$

$\mathcal{E} := \lambda v.\ \gamma \times v\ +\ (1-\gamma) / |V|$

(b) Examples

Fig. 7. (a) Grammar for Kernel Functions (b) Example Kernel Functions. (min and + filter none values $\bot$.)

Pull Model. The characteristic of the pull model is that vertices pull the values of their predecessors to calculate their new values. We consider the pull model for idempotent and non-idempotent reduction functions in turn.

*Pull model with idempotent reduction (*pull+*).* The pull model for idempotent reduction is represented in Fig. 8, Def. 1. The value of the vertex v in the iteration k is represented as $\mathcal{S}^k_{\text{pull+}}(v)$. In the beginning when $k = 0$, vertices have no value $\bot$. In the first iteration $k = 1$, they are initialized by the initialization function $\mathcal{I}$. In subsequent iterations $k+1,\ k \geq 1$, each vertex v pulls values of its predecessors. For each predecessor u, the propagation function $\mathcal{P}$ is applied to the value $\mathcal{S}^k_{\text{pull+}}(u)$ of u (from the previous iteration k) and the connecting edge $\langle u, v\rangle$. Then, as illustrated in Fig. 3b, all the propagated values are reduced by $\mathcal{R}$ with each other and then with the previous value $\mathcal{S}^k_{\text{pull+}}(v)$ of v. Finally, applying the epilogue function to the reduced value results in the new value $\mathcal{S}^{k+1}_{\text{pull+}}(v)$ of v. As an optimization, the above update is performed only if the set of predecessors of v whose value have changed in the previous iteration $\text{CPreds}^k(v)$ is non-empty.

*Pull model with non-idempotent reduction (*pull−*).* The pull model for non-idempotent reduction is represented in Fig. 8, Def. 2. The value of the vertex v in the iteration k is represented as $\mathcal{S}^k_{\text{pull−}}(v)$. Similar to the previous model, the values from predecessors are propagated and reduced. The difference is that after reducing the propagated values, the result is not reduced with the previous value of the vertex. The reason is to avoid duplicate reduction with the non-idempotent reduction function. Consider a vertex v and a predecessor u of v. Assume that the value of u represents the reduction of a set S of values. After the value of u is propagated to v, the value of v includes the reduced and propagated values of S. Assume that the value of u is updated again to represent the reduction of more values. If the new value of u is propagated to v and reduced with the current value of v, then the set S is included in the value of v twice.

Push Model. In the pull model above, each vertex itself pulls values from its predecessors. In contrast, in the push model, the predecessors push values to the vertex when they are updated. We consider the push model for idempotent and non-idempotent reduction functions in turn.

*Push model with idempotent reduction (*push+*).* The push model for idempotent reduction is represented in Fig. 8, Def. 3. The value of the vertex v in the iteration k is represented as $\mathcal{S}^k_{\text{push+}}(v)$. The iterations 0 and 1 are similar to the previous models. In subsequent iterations $k+1,\ k \geq 1$, for each vertex v, the predecessors $\{u_0, .., u_{n-1}\}$ that have been changed in the previous iteration independently propagate their values and reduce it with the current value of v. Since the reduction function is commutative and associative, the predecessors can apply their updates in any order.

$$\text{CPreds}^k(v) = \left\{u \mid u \in \text{preds}(v) \wedge \mathcal{S}^k(u) \neq \mathcal{S}^{k-1}(u)\right\}$$

DEFINITION 1 (PULL (IDEMPOTENT REDUCTION)).

$$\mathcal{S}^0_{\text{pull+}}(v) \coloneqq \bot$$
$$\mathcal{S}^1_{\text{pull+}}(v) \coloneqq \mathcal{I}(v)$$
$$\mathcal{S}^{k+1}_{\text{pull+}}(v) \coloneqq \begin{cases} \mathcal{S}^k_{\text{pull+}}(v) & \text{if } \text{CPreds}^k(v) = \emptyset \\ \mathcal{E}\left[\mathcal{R}\left(\mathcal{S}^k_{\text{pull+}}(v),\ \mathcal{R}_{u \in \text{preds}(v)}\, \mathcal{P}\left(\mathcal{S}^k_{\text{pull+}}(u), \langle u, v\rangle\right)\right)\right] & \text{else} \end{cases} \qquad k \geq 1$$

DEFINITION 2 (PULL (NON-IDEMPOTENT REDUCTION)).

$$\mathcal{S}^0_{\text{pull}-}(v) \coloneqq \bot$$
$$\mathcal{S}^1_{\text{pull}-}(v) \coloneqq \mathcal{I}(v)$$
$$\mathcal{S}^{k+1}_{\text{pull}-}(v) \coloneqq \begin{cases} \mathcal{S}^k_{\text{pull}-}(v) & \text{if } \text{CPreds}^k(v) = \emptyset \\ \mathcal{E}\left[\mathcal{R}_{u \in \text{preds}(v)}\, \mathcal{P}\left(\mathcal{S}^k_{\text{pull}-}(u), \langle u, v\rangle\right)\right] & \text{else} \end{cases} \qquad k \geq 1$$

DEFINITION 3 (PUSH (IDEMPOTENT REDUCTION)).

$$\mathcal{S}^0_{\text{push+}}(v) \coloneqq \bot$$
$$\mathcal{S}^1_{\text{push+}}(v) \coloneqq \mathcal{I}(v)$$
$$\mathcal{S}^{k+1}_{\text{push+}}(v) \coloneqq \mathcal{E}(S_n), \quad k \geq 1 \quad \textit{where}$$
$$\text{let } \{u_0, .., u_{n-1}\} \coloneqq \text{CPreds}^k(v) \text{ in}$$
$$S_0 \coloneqq \mathcal{S}^k_{\text{push+}}(v)$$
$$S_{i+1} \coloneqq \mathcal{R}\left(S_i,\ \mathcal{P}\left(\mathcal{S}^k_{\text{push+}}(u_i), \langle u_i, v\rangle\right)\right)$$

DEFINITION 4 (PUSH (NON-IDEMPOTENT REDUCTION)).

$$\mathcal{S}^0_{\text{push}-}(v) \coloneqq \bot$$
$$\mathcal{S}^1_{\text{push}-}(v) \coloneqq \mathcal{I}(v)$$
$$\mathcal{S}^{k+1}_{\text{push}-}(v) \coloneqq \mathcal{E}(S_n), \quad k \geq 1 \quad \textit{where}$$
$$\text{let } \{u_0, .., u_{n-1}\} \coloneqq \text{preds}(v) \text{ in}$$
$$S_0 \coloneqq \bot$$
$$S_{i+1} \coloneqq \mathcal{R}\left(S_i,\ \mathcal{P}\left(\mathcal{S}^k_{\text{push}-}(u_i), \langle u_i, v\rangle\right)\right)$$

Fig. 8. Four Iterative Reduction Methods. (GRAFS also incorporates another variant of Push, Non-idempotent Reduction (appendix § 3.1.2)). $\text{CPreds}^k(v)$: The predecessors of the vertex v that changed in the iteration k

In each iteration, the initial value S_0 of v is its value in the previous iteration k. For each changed predecessor u_i, the propagation function $\mathcal{P}$ is applied to the value $\mathcal{S}^k_{\text{push+}}(u_i)$ of u_i (from the previous iteration k) and the connecting edge $\langle u_i, v\rangle$. The result is then reduced with the current value S_i of v to calculate its new value S_{i+1}. Propagation and reduction by the last changed predecessor u_{n-1} results in the value S_n. The final value of v is the result of applying the epilogue $\mathcal{E}$ to S_n.

*Push model with non-idempotent reduction (*push−*).* This model works for non-idempotent (in addition to idempotent) reduction functions. The iterative model is represented in Fig. 8, Def. 4. Let the value of the vertex v in the iteration k be represented as $\mathcal{S}^k_{\text{push}-}(v)$. Since the reduction function may not be idempotent, in contrast to the previous model, vertices start from the none value $S_0 = \bot$, and all the predecessors u_i propagate their values in each iteration. For each predecessor u_i, the propagate function $\mathcal{P}$ is applied to the latest value $\mathcal{S}^k_{\text{push}-}(u_i)$ of u_i and the connecting edge $\langle u_i, v\rangle$. The resulting value is reduced with the current value S_i of v. We note that this variant makes all vertices active during an iteration; GRAFS also incorporates another variant (appendix § 3.1.2) where only the vertices whose values change are active and propagate their values. In this variant, an active predecessor u_i first rollbacks its previous update before applying its new update.

The iterative models that we saw here are *synchronous*. In the synchronous model, vertices store their previous in addition to their new value to propagate their previous value. In the *asynchronous* model, however, each vertex stores one value, and vertices can propagate intermediate values. The four asynchronous models and their correctness is available in the appendix § 3.1.3. Further, we present streaming iterative models and their correctness in the appendix § 3.1.4.

5 SPECIFICATION AND FUSION

In this section, we define the core specification language, its denotational semantics, and the semantics-preserving fusion transformations.

5.1 Core Specification Language

To present the crux of the fusion transformations, we define a core specification language in Fig. 9. It features both reduction over paths and reduction over vertices. A computation can be specified as a reduction r over the values of vertices. The value of vertices, in turn, can be specified as a nested reduction m over the paths to each vertex. More elaborate computations can be specified by nested path-based computations, and applying operations between multiple path-based and vertex-based computations. We will visit each term type in turn.

Vertex-based and path-based reductions. A vertex-based reduction $\underset{\mathsf{V}}{\mathcal{R}}\ m$ applies a reduction function $\mathcal{R}$ to the result of path-based reductions m over all vertices V. The function $\mathcal{R}$ is a commutative and associative function such as min, max, $\vee$, $\wedge$ and $\sum$. Larger vertex-based reductions $r \oplus r'$ can be constructed using the operators $\oplus$. A path-based reduction $\underset{p \in P}{\mathcal{R}}\ \mathcal{F}(p)$ applies a reduction function $\mathcal{R}$ to the results of applying the function $\mathcal{F}$ to each path p in set of paths P. Similar to vertex-based reductions, larger path-based reductions $m \oplus m'$ can be constructed using the operators $\oplus$. The path function $\mathcal{F}$ is the length, weight, or capacity of the path. The set of paths P can be either Paths that denotes all the paths to each vertex, or the restricted paths $\underset{p \in P}{\text{args}\,\mathcal{R}}\ \mathcal{F}(p)$ where $\mathcal{R} \in \{\min, \max\}$ that denotes the paths in P whose $\mathcal{F}$ value is the extremum. (The r and m terms can be also variables x that can be substituted with a value n or a map value d from vertices to values.)

r	$:=$	$\underset{\mathsf{V}}{\mathcal{R}}\ m \mid r \oplus r \mid x \mid$ ilet $X := M$ in mlet $X := E$ in rlet $X := R$ in e	Vertex-based Red.
m	$:=$	$\underset{p \in P}{\mathcal{R}}\ \mathcal{F}(p) \mid m \oplus m \mid x \mid$ ilet $X := M$ in e	Path-based Red.
P	$:=$	Paths $\mid \underset{p \in P}{\text{args}\,\mathcal{R}}\ \mathcal{F}(p)$	Paths
$\mathcal{R}$	$:=$	min $\mid$ max $\mid \vee \mid \wedge \mid \sum$	Reduction Fun.
$\mathcal{F}$	$:=$	length $\mid$ weight $\mid$ capacity	Path Fun.
$\oplus$	$:=$	min $\mid$ max $\mid \wedge \mid \vee \mid +$ $\mid -$ $\mid \times \mid / \mid = \mid < \mid >$	Operation
p			Path Variable
x			Variable
e	$:=$	$e \oplus e \mid x$	
X	$:=$	$\langle X, X \rangle \mid x$	
M	$:=$	$\langle M, M \rangle \mid \mathcal{R}\,\mathcal{F}$	
R	$:=$	$\langle R, R \rangle \mid \mathcal{R}\,\langle \overline{x} \rangle \mid \mathcal{R}\,\langle \overline{d} \rangle$	
E	$:=$	$\langle E, E \rangle \mid e$	
$\mathbb{R}$	$:=$	$[\,] \mid \mathbb{R} \oplus r \mid r \oplus \mathbb{R}$	Context for r
$\mathbb{M}$	$:=$	$[\,] \mid \underset{\mathsf{V}}{\mathcal{R}}\ \mathbb{M} \mid \mathbb{M} \oplus m \mid m \oplus \mathbb{M}$	Context for m
$\mathbb{M}\mathsf{s}$	$:=$	$[\,] \mid \langle \mathbb{M}\mathsf{s}, M \rangle \mid \langle M, \mathbb{M}\mathsf{s} \rangle \mid$ ilet $X := \mathbb{M}\mathsf{s}$ in $e \mid$ ilet $X := \mathbb{M}\mathsf{s}$ in mlet $X := E$ in rlet $X := R$ in e	Context for M
$\mathbb{R}\mathsf{s}$	$:=$	$[\,] \mid \langle \mathbb{R}\mathsf{s}, R \rangle \mid \langle R, \mathbb{R}\mathsf{s} \rangle \mid$ ilet $X := M$ in mlet $X := E$ in rlet $X := \mathbb{R}\mathsf{s}$ in e	Context for R
n			Value
v			Vertex Value
$d : \mathcal{D}_m$	$:=$	$(\mathsf{V}(g) \mapsto \mathbb{N}) \cup \{\bot\}$	Sem. Dom. of m
$\mathsf{n}_\bot : \mathcal{D}_r$	$:=$	$\mathbb{N} \cup \{\bot\}$	Sem. Dom. of r

Fig. 9. Core Specification Language

Let forms. Let terms factor different reductions. Factored reductions are conducive to fusion. As shown in Fig. 9, the terms m and r both have let forms. The m term constructor ilet $X := M$ in e binds variables X to factored path-based reductions M for the expression e. The expression e can apply operators $\oplus$ to the variables X. Both the variables X and reductions M can be inductively constructed as pairs. A single path-based reduction M is simply represented as $\mathcal{R}\,\mathcal{F}$ where $\mathcal{R}$ is the reduction function and $\mathcal{F}$ is the path function. Similarly, the triple-let r constructor ilet $X := M$ in

$$\textsc{SPRed}\quad \frac{[\![\underset{p\in P}{\mathcal{R}}\ \mathcal{F}(p)]\!]\,(g) \;=}{[\mathsf{v} \mapsto \mathcal{R}\ \{\mathcal{F}(p) \mid p \in [\![P]\!]\,(g)(\mathsf{v})\}]_{\mathsf{v}\in\mathsf{V}(g)}}$$

$$\textsc{SMBin}\quad [\![m \oplus m']\!]\,(g) \;=\; [\![m]\!]\,(g) \;\oplus\; [\![m']\!]\,(g)$$

$$\textsc{SVRed}\quad [\![\underset{\mathsf{V}}{\mathcal{R}}\ m]\!]\,(g) \;=\; \mathcal{R}\ \left\{\overline{[\![m]\!](g)(\mathsf{v})}_{\ \mathsf{v}\in\mathsf{V}(g)}\right\}$$

$$\textsc{SRLet}\quad [\![\begin{array}{l}\text{ilet } X := M \text{ in}\\ \text{mlet } X' := E \text{ in}\\ \text{rlet } X'' := R \text{ in}\\ e\end{array}]\!]\,(g) \;= \;[\![\, e\,[X'' := [\![R\,[X' := [\![E\,[X := [\![M]\!]\,(g)]]\!]\,]\!]\,(g)]\,]\!]$$

$$\textsc{SMPair}\quad [\![\langle M,\ M' \rangle]\!]\,(g) = \langle [\![M]\!](g),\ [\![M']\!](g) \rangle$$

$$\textsc{SRPair}\quad [\![\langle R,\ R' \rangle]\!]\,(g) = \langle [\![R]\!](g),\ [\![R']\!](g) \rangle$$

$$\textsc{SMM}\quad [\![\mathcal{R}\ \mathcal{F}]\!] = \left[\!\!\left[\underset{p\in\mathsf{Paths}}{\mathcal{R}}\ \mathcal{F}(p) \right]\!\!\right]$$

$$\textsc{SRR}\quad \left[\!\!\left[\mathcal{R}\ \left\langle \overline{[\mathsf{v} \mapsto n_{\mathsf{v}}]}_{\mathsf{v}\in\mathsf{V}(g)}, .., \overline{[\mathsf{v} \mapsto n'_{\mathsf{v}}]}_{\mathsf{v}\in\mathsf{V}(g)} \right\rangle \right]\!\!\right] = \left[\!\!\left[\underset{\mathsf{V}}{\mathcal{R}}\ \left(\overline{[\mathsf{v} \mapsto \langle n_{\mathsf{v}},\ ..,\ n'_{\mathsf{v}} \rangle]}_{\mathsf{v}\in\mathsf{V}(g)} \right) \right]\!\!\right]$$

Fig. 10. Denotational Semantics of the Specification Language

mlet $X' := E$ in rlet $X'' := R$ in e binds variables X to factored path-based reductions M, binds variables X' to expressions E (on X), and binds variables X'' to factored vertex-based reductions R (on X'). A triple-let term represents an r term as a sequence: path-based reductions, mappings on the results, and finally vertex-based reductions on the results. We will see that this form enables fusion (§ 5.2) and can be directly implemented (§ 7). Similar to M, the factored vertex-based reductions R can be inductively constructed as pairs. A single vertex-based reduction R is $\mathcal{R}\ \langle\overline{x}\rangle$ that is a reduction over tuples of variables $\langle\overline{x}\rangle$ (or is $\mathcal{R}\ \langle\overline{d}\rangle$ after the variables are substituted with map values d from vertices to values). To concisely represent the fusion rules, we define the context $\mathbb{R}$ to abstract the surrounding term where a term r appears. Similarly, we define the contexts $\mathbb{M}$, $\mathbb{M}$s, and $\mathbb{R}$s for the terms m, M and R.

Semantics and Compositionality. We define the denotational semantics of the specification language (presented in Fig. 9). For brevity, we showcase the semantics $[\![\ ]\!]$ of a subset of the term constructors in Fig. 10. (The full semantics is available in the appendix § 2.1.) The semantics of an undefined or stuck computation is represented as $\perp$. In each rule, it is assumed that the semantics of subterms are not undefined; otherwise, the semantics of the whole term is undefined as well. The domain $\mathcal{D}_m$ of a path-based computation m on a graph g is a finite map from each vertex of g to natural numbers $\mathsf{V}(g) \mapsto \mathbb{N}$, and $\perp$ (for undefined computation). The domain $\mathcal{D}_r$ of a vertex-based computation r is the natural numbers $\mathbb{N}$ and $\perp$. We use the notation $\overline{[k_i \mapsto v_i]}_i$ for a finite map that maps each key k_i to value v_i over the range i.

The rule SPRed defines the semantics of the path-based reduction $\underset{p\in P}{\mathcal{R}}\ \mathcal{F}(p)$. It uses the (elided) semantics of paths P that is a map from each vertex v to the set of paths to v. For each vertex v, the rule SPRed applies the function $\mathcal{F}$ to each path to v, and then applies the reduction function $\mathcal{R}$ to the resulting values. (Since the reduction functions $\mathcal{R}$ are commutative and associative, they can be applied to the values in any order.) The rule SMBin defines the semantics of $m \oplus m'$ as the result of the operator $\oplus$ on the semantics of m and m'. The operators $\mathcal{R}$ and $\oplus$ are in the syntactic and semantic domains when they are on the left- and right-hand side of the rules respectively. The operator $\oplus$ is simply lifted to maps by pointwise application to the values of each key.

The rule SVRed defines the semantics of the vertex-based reduction $\underset{\mathsf{V}}{\mathcal{R}}\ m$ using the map resulted from the semantics of m; it reduces the values of the map for all vertices. The rule SRLet defines

the semantics of triple-let terms by three subsequent substitutions: the substitution of the variables X with the semantics of M in E, the substitution of X' with the semantics of E in R, and finally the substitution of X'' with the semantics of R in e. (The semantics of e and E are elided.)

The rules SMPair and SRPair define the semantics of pairs of factored reductions M and R inductively. The two rules SMM and SRR reduce the semantics of single factored reductions to expanded reductions. The rule SMM defines the semantics of the factored path-based reduction $\mathcal{R}\ \mathcal{F}$ as a path-based reduction on the paths Paths. The rule SRR defines the semantics of a factored vertex-based reduction. It merges the tuple of the factored maps $\overline{[\mathsf{v} \mapsto n_\mathsf{v}]}_{\mathsf{v} \in V(g)}$, .., $\overline{[\mathsf{v} \mapsto n'_\mathsf{v}]}_{\mathsf{v} \in V(g)}$ into a map from vertices v to tuples $\langle n_\mathsf{v}, .., n'_\mathsf{v} \rangle$, and then applies the vertex-based reduction.

We prove that the semantics is compositional. If two terms are semantically equivalent, replacing one with the other in any context is semantics-preserving. Compositionality of the semantics is used to prove that the fusion transformations are semantic-preserving. The following theorem states the compositionality for r. (Similar lemmas for m, M and R and their proofs are available in the appendix § 4.2.)

Lemma 1 (Compositionality). *For all r, r' and $\mathbb{R}$, if $[\![r]\!] = [\![r']\!]$ then $[\![\mathbb{R}[r]]\!] = [\![\mathbb{R}[r']]\!]$.*

5.2 Fusion

We now present the fusion transformations. Fusion reduces computation time by combining separate reductions into a single reduction. The transformations have three main forms: fusion of nested path-based reductions, fusion of pairs of path-based reductions, and fusion of pairs of vertex-based reductions. The result of fusion is an equivalent specification in the triple-let form with separate terms for path-based reduction, mapping over vertices and vertex-based reduction.

The fusion rules are presented in Fig. 11. The top-level fusion relation $\Rightarrow_r$ is called r-fusion and transforms an r term to another. The other fusion relations $\Rightarrow_m$, $\Rightarrow_M$, and $\Rightarrow_R$ which transform m, M and R terms are called m-fusion, M-fusion and R-fusion. We consider m-fusions first. The rule FMInR states that m-fusions can be applied to m terms that appear in the context of r terms. (Both $\mathbb{M}[m_1]$ and $\mathbb{M}[m_2]$ in this rule are r terms.) The rule FMInM states that m-fusions can be applied to m terms in the context of other m terms.

Fusing nested path-based reductions. The rule FPNest m-fuses nested path-based reductions to flat reductions. Consider the nested path-based reduction $\underset{p' \in P'}{\mathcal{R}}\ \mathcal{F}(p)$ where the set of paths P' is another path-based reduction $\underset{p \in P}{\mathrm{args}\mathcal{R}'}\ \mathcal{F}'(p)$ where $\mathcal{R}'$ is min or max. Let us assume that $\mathcal{R}'$ is min. A straightforward calculation computes $\mathcal{F}'$ on the paths P and finds the subset of paths P' with the minimum value, and then computes $\mathcal{F}$ on the paths P' and reduces them by $\mathcal{R}$. An optimized calculation can compute both $\mathcal{F}'$ and $\mathcal{F}$ on the paths P simultaneously and only consider the pairs with the minimum first element to calculate the reduction $\mathcal{R}$ over the second elements. To calculate the values of the path functions $\mathcal{F}$ and $\mathcal{F}'$, this approach enumerates paths only once instead of twice. Therefore, the two reductions can be fused into one reduction as ilet $\langle x, x' \rangle := \underset{p' \in P}{\mathcal{R}''}\ \mathcal{F}''(p')$ in x'. The new path function $\mathcal{F}''$ returns the pair of values $\mathcal{F}'(p')$ and $\mathcal{F}(p')$ for an input path p'. The new reduction function $\mathcal{R}''$ considers the first element of the two input pairs and if the first element of one input is (strictly) smaller than the other, that input is returned. That input takes over because the set of paths for the reduction $\mathcal{R}$ are only those with the minimum value for $\mathcal{F}'$. On the other hand, if the first elements of the inputs are equal, their second elements are reduced by $\mathcal{R}$ to make the second element of the output pair. The rule FPNest can be repeatedly applied to a deeply nested path-based reduction to flatten it to a reduction over the basic paths term Paths.

FMINR
$$\frac{m_1 \Rightarrow_m m_2}{\mathbb{M}[m_1] \Rightarrow_r \mathbb{M}[m_2]}$$

FMINM
$$\frac{m_1 \Rightarrow_m m_2}{\mathbb{M}[m_1] \Rightarrow_m \mathbb{M}[m_2]}$$

FPNEST
$$\underset{p' \in \text{args}\, \underset{p \in P}{\mathcal{R}'}\, \mathcal{F}'(p)}{\mathcal{R}}\ \mathcal{F}(p') \Rightarrow_m \text{ilet } \langle x, x' \rangle := \underset{p \in P}{\mathcal{R}''}\ \mathcal{F}''(p) \text{ in } x'$$
where $\mathcal{R}' \in \{\min, \max\}$
$\mathcal{F}'' := \lambda p.\ \langle \mathcal{F}'(p), \mathcal{F}(p) \rangle$
$\mathcal{R}''(\langle a, b \rangle, \langle a', b' \rangle) :=$
if $(a = a')$ then $\langle a, \mathcal{R}(b, b') \rangle$
else if $(\mathcal{R}'(a, a') = a)$ then $\langle a, b \rangle$ else $\langle a', b' \rangle$

FPRED
$$\underset{p \in \text{Paths}}{\mathcal{R}}\ \mathcal{F}(p) \Rightarrow_m \text{ilet } x := \mathcal{R}\ \mathcal{F} \text{ in } x$$

FILETBIN
$$(\text{ilet } X_1 := M_1 \text{ in } e_1) \oplus (\text{ilet } X_2 := M_2 \text{ in } e_2) \Rightarrow_m \text{ilet } \langle X_1, X_2 \rangle := \langle M_1, M_2 \rangle \text{ in } e_1 \oplus e_2 \quad \text{if } \begin{array}{l} \text{free}(e_1) \cap X_2 = \emptyset \\ \text{free}(e_2) \cap X_1 = \emptyset \end{array}$$

FMINILET
$$\frac{M_1 \Rightarrow_M M_2}{\text{ilet } X := \mathbb{M}\text{s}[M_1] \text{ in } e \Rightarrow_m \text{ilet } X := \mathbb{M}\text{s}[M_2] \text{ in } e}$$

FMPAIR
$$\langle \mathcal{R}\ \mathcal{F}, \mathcal{R}'\ \mathcal{F}' \rangle \Rightarrow_M \mathcal{R}''\ \mathcal{F}''$$
where $\mathcal{F}'' := \lambda p.\ \langle \mathcal{F}(p), \mathcal{F}'(p) \rangle$
$\mathcal{R}''(\langle a, b \rangle, \langle a', b' \rangle) :=$
$\langle \mathcal{R}(a, a'), \mathcal{R}'(b, b') \rangle$

FVRED
$$\underset{\vee}{\mathcal{R}}\ (\text{ilet } X := \mathcal{R}'\ \mathcal{F} \text{ in } e) \Rightarrow_r \text{ilet } X := \mathcal{R}'\ \mathcal{F} \text{ in mlet } x := e \text{ in rlet } x' := \mathcal{R}\ x \text{ in } x'$$

FLETSBIN
$$\left(\begin{array}{l} \text{ilet } X_1 := M_1 \text{ in} \\ \text{mlet } X_1' := E_1 \text{ in} \\ \text{rlet } X_1'' := R_1 \text{ in} \\ e_1 \end{array}\right) \oplus \left(\begin{array}{l} \text{ilet } X_2 := M_2 \text{ in} \\ \text{mlet } X_2' := E_2 \text{ in} \\ \text{rlet } X_2'' := R_2 \text{ in} \\ e_2 \end{array}\right) \Rightarrow_r \left(\begin{array}{l} \text{ilet } \langle X_1, X_2 \rangle := \langle M_1, M_2 \rangle \text{ in} \\ \text{mlet } \langle X_1', X_2' \rangle := \langle E_1, E_2 \rangle \text{ in} \\ \text{rlet } \langle X_1'', X_2'' \rangle := \langle R_1, R_2 \rangle \text{ in} \\ e_1 \oplus e_2 \end{array}\right) \text{ if } \begin{array}{l} \text{free}(E_1) \cap X_2 = \\ \text{free}(E_2) \cap X_1 = \\ \text{free}(R_1) \cap X_2' = \\ \text{free}(R_2) \cap X_1' = \\ \text{free}(e_1) \cap X_2'' = \\ \text{free}(e_2) \cap X_1'' = \emptyset \end{array}$$

FMINLETS
$$\frac{M_1 \Rightarrow_M M_2}{\left(\begin{array}{l} \text{ilet } X := \mathbb{M}\text{s}[M_1] \text{ in} \\ \text{mlet } X' := E \text{ in} \\ \text{rlet } X'' := R \text{ in } e \end{array}\right) \Rightarrow_r \left(\begin{array}{l} \text{ilet } X := \mathbb{M}\text{s}[M_2] \text{ in} \\ \text{mlet } X' := E \text{ in} \\ \text{rlet } X'' := R \text{ in } e \end{array}\right)}$$

FRINLETS
$$\frac{R_1 \Rightarrow_R R_2}{\left(\begin{array}{l} \text{ilet } X := M \text{ in} \\ \text{mlet } X' := E \text{ in} \\ \text{rlet } X'' := \mathbb{R}\text{s}[R_1] \text{ in } e \end{array}\right) \Rightarrow_r \left(\begin{array}{l} \text{ilet } X := M \text{ in} \\ \text{mlet } X' := E \text{ in} \\ \text{rlet } X'' := \mathbb{R}\text{s}[R_2] \text{ in } e \end{array}\right)}$$

FRPAIR
$$\langle \mathcal{R}_1\ x_1, \mathcal{R}_2\ x_2 \rangle \Rightarrow_R \mathcal{R}_3\ \langle x_1, x_2 \rangle$$
where $\mathcal{R}_3(\langle a, b \rangle, \langle a', b' \rangle) :=$
$\langle \mathcal{R}_1(a, a'), \mathcal{R}_2(b, b') \rangle$

Fig. 11. Fusion Rules

Factoring, pairing and fusing path-based reductions. The rule FPRED factors out a flat reduction to an equivalent let form. The rule FILETBIN fuses an operation between two let terms to a single let term. It pairs the factored reductions M_1 and M_2 of the two let terms. The condition of the rule prevents the free variables of the expression of one term from clashing with the bound variables of another. The rule FMINILET allows the factored reductions M in the context of a let term to be fused. The rule FMPAIR M-fuses a pair of factored reductions $\langle \mathcal{R}\ \mathcal{F}, \mathcal{R}'\ \mathcal{F}' \rangle$ to a single reduction $\mathcal{R}''\ \mathcal{F}''$ that calculates the two reductions simultaneously. The path function $\mathcal{F}''$ returns the pair of the results of $\mathcal{F}$ and $\mathcal{F}'$. Similarly, the reduction function $\mathcal{R}''$ returns a pair: the reduction of the first elements by $\mathcal{R}$ and the second elements by $\mathcal{R}'$.

Factoring into, pairing and fusing triple-let terms. The rules above can factor all path-based reductions m to the let form, and fuse factored reductions to a single one. The next rule FVRed transforms vertex-based reductions that are applied to these path-based reductions to an equivalent triple-let form. The triple-let form factors path-based and vertex-based reductions in separate let parts. The rule FLetsBin fuses an operation between two triple-let terms to a single triple-let term. It pairs the factored path-based reductions M, expression E, and vertex-based reduction R of the two terms. The rules FMInLets and FRinLets allow the factored reductions M and R in the context of a triple-let term to be fused.

Fusing vertex-based reductions. The rule FRPair presents R-fusions. It fuses a pair of factored vertex-based reductions $\langle \mathcal{R}_1\ x_1, \mathcal{R}_2\ x_2 \rangle$ to a single reduction $\mathcal{R}_3\ \langle x_1, x_2 \rangle$. Given two pairs, $\mathcal{R}_3$ returns a pair: the reduction of the first elements by $\mathcal{R}_1$ and the second elements by $\mathcal{R}_2$.

We saw an example fusion in Fig. 2. (More examples are available in the appendix § 2.3.) The fusion transformation presented above is semantic-preserving: terms are only fused into other terms with the same semantics. The following theorem states the semantics-preservation property of fusion. (The proofs are available in the appendix § 4.3.)

Theorem 1 (Semantics-preserving Fusion). *For all r_1 and r_2, if $r_1 \Rightarrow_r r_2$ then $[\![r_1]\!] = [\![r_2]\!]$.*

5.3 Extensions

We now consider extensions to the core syntax and the fusion rules.

Common Operation Elimination. Fusion factors path-based reductions, vertex-based map operations and vertex-based reductions into the triple-let form. This form facilitates common operation elimination. For example, if a path-based reduction appears twice in the first let and assigned to two sets of variables, one can be eliminated and the result of the other can be assigned to both sets of variables. (The elimination rules are available in the appendix § 2.2.1.)

Domain. The scalar semantic domain of the core language was confined to the natural numbers. The domain are simply extended to booleans, vertex identifiers and also sets of values. Thus, the reduction operations are extended with union $\cup$ and intersection $\cap$, and the path functions are extended with head and penultimate. The function head returns the identifier of the head vertex of the path, and the function penultimate returns the identifier of the penultimate (that is the vertex before the last) of the path.

Unary operations and Literals. The path-based reductions m and vertex-based reductions r and their fusion rules can be simply extended with unary operations and literals. (the appendix § 2.2.3)

Vertex Variables. We extend the core syntax with path terms $\mathsf{Paths}(v, v')$ and $\mathsf{Paths}(v)$ that can specify vertex variables as source and destination. The term $\mathsf{Paths}(v, v')$ specifies the set of paths from the source v to the destination v', and the term $\mathsf{Paths}(v)$ specifies the set of paths from any source to the destination v. Thus, the source s of a path-based reduction can be either a vertex v or none $\perp$. A factored path-based reduction $\underset{c}{\mathcal{R}}\ \mathcal{F}$ carries its configuration c, that is either a source, or a pair of other configurations. We also extend the syntax with vertex-based reductions $\underset{v \in V}{\mathcal{R}}\ m$ that can bind the vertex variable v. (These syntactic extensions and their fusion rules are available in the appendix § 2.2.4.)

Syntactic Sugar. Syntactic sugar enables concise specifications. For example, the term $\mathcal{F}(\arg \underset{p \in P}{\mathcal{R}}\ \mathcal{F}'(p))$ where $\mathcal{R}$ is either min or max first finds a path p in P with the minimum or maximum value for the function $\mathcal{F}'$, and then returns the result of applying $\mathcal{F}$ to p. It is used to specify the BFS use-case. The following rule expands this term to a path-based reduction in the let

form. The path function $\mathcal{F}''$ returns the pair of the results of $\mathcal{F}'$ and $\mathcal{F}$. The reduction function $\mathcal{R}'$ returns the input pair with the minimum or maximum first element.

$$\begin{array}{l}\text{FMRED}\\ \mathcal{F}\ (\underset{p\in P}{\arg\mathcal{R}}\ \mathcal{F}'(p)) \quad := \quad \text{ilet } \langle x, x'\rangle := \underset{p\in P}{\mathcal{R}'}\ \mathcal{F}''(p) \text{ in } x' \quad \text{where} \quad \mathcal{R} \in \{\min, \max\}\\ \mathcal{F}'' \ := \ \lambda p.\ \langle \mathcal{F}'(p), \mathcal{F}(p)\rangle \qquad \mathcal{R}'(\langle a, b\rangle, \langle a', b'\rangle) \ := \ \text{if } (\mathcal{R}(a, a') = a) \text{ then } \langle a, b\rangle \text{ else } \langle a', b'\rangle\end{array} \tag{7}$$

In the appendix § 2.2.5, we present the syntactic sugar (1) cardinality $|P|$, (2) vertex-based reduction over a given subset of vertices $\underset{v\in\{v_1,..,v_n\}}{\mathcal{R}}\ m$, and (3) vertex-based reduction constrained by a path-based reduction $\underset{v\in V\wedge m'}{\mathcal{R}}\ m$. The latter was used in the use-cases NSP, RADIUS, and DS in Fig. 6.

As an example DS is fused as follows:

$$\begin{aligned}
\text{DS}(s) &= \bigcup_{v\in V\wedge \left(\min_{p\in\text{Paths}(s,v)} \text{weight}(p)\right) > 7} \{v\}\\
&= \underset{v\in V}{\mathcal{R}} \left\langle \left(\min_{p\in\text{Paths}(s,v)} \text{weight}(p)\right) > 7, \{v\}\right\rangle \quad \text{where} \quad \begin{array}{l}\mathcal{R}(\langle a, b\rangle, \langle a', b'\rangle) :=\\ \quad \text{if } (a \wedge a') \text{ then } \langle a, b \cup b'\rangle\\ \quad \text{else}(a) \text{ then } \langle a, b\rangle\\ \quad \text{else } \langle a', b'\rangle\end{array}\\
&= \underset{v\in V}{\mathcal{R}} \left\langle \left(\text{ilet } x := \min_{s} \text{weight in } x\right) > \text{ilet } x' := \bot \text{ in } 7,\ \ \text{ilet } x'' := \bot \text{ in } \{v\}\right\rangle\\
&= \underset{v\in V}{\mathcal{R}} \left(\text{ilet } \langle\langle x, x'\rangle, x''\rangle := \langle\langle \min_{s} \text{weight}, \bot\rangle, \bot\rangle \text{ in } \langle x > 7, \{v\}\rangle\right)\\
&= \underset{v\in V}{\mathcal{R}} \left(\text{ilet } x := \min_{s} \text{weight in } \langle x > 7, \{v\}\rangle\right)\\
&= \left(\begin{array}{l}\text{ilet } x := \min_{s} \text{weight in}\\ \text{mlet } x' := \langle x > 7, \{v\}\rangle \text{ in}\\ \text{rlet } x'' := \mathcal{R}\ x' \text{ in}\\ x''\end{array}\right)
\end{aligned}$$

Nested Triple-lets. The core syntax supports expressions that can be fused to a single iteration-map-reduce triple-let term. We extend the core syntax to support nested vertex-based reductions, and extend the fusion rules to fuse them. For example, the use-case LTRUST that we saw in Fig. 6 uses the vertex-based reduction RADIUS as a nested term. Nested triple-let terms can be translated to a sequence of iteration-map-reduce rounds on the graph. (In the appendix § 2.2.6, we define the extension and show that LTRUST is fused to two rounds of iteration-map-reduce.)

6 MAPPING SPECIFICATION TO ITERATION-MAP-REDUCE

As we saw in the final term of Fig. 2, fusion results in the triple-let form shown in Fig. 12. The three let parts can be directly mapped to three computation primitives: iteration, map and reduce. Each vertex stores the variables X and X'. The first let is mapped to an iterative calculation for the path-based reduction $\underset{c}{\mathcal{R}}\ \mathcal{F}$ that results in values for the variables X in each vertex. The second let is mapped to a map operation over vertices: given the values of the variables X in each vertex, the map operation calculates the values of the expressions E, and stores the results in the variables X' for the vertex. The third let is mapped to a reduction operation over vertices: given the values of the variables $\overline{x'}$ in X' in each vertex, the

$$\begin{array}{l}\text{ilet } X := \underset{c}{\mathcal{R}}\ \mathcal{F} \text{ in}\\ \text{mlet } X' := E \text{ in}\\ \text{rlet } X'' := \mathcal{R}'\ \langle\overline{x'}\rangle \text{ in } e\end{array}$$

Fig. 12. Triple-let Form

reduction operation $\mathcal{R}'\ \langle\overline{x'}\rangle$ reduces the values of $\langle\overline{x'}\rangle$ for all vertices, and stores the results in the global variables X''. Finally, the single expression e is calculated based on the values of X''.

The two latter primitives, vertex-based mapping and reduction, can be implemented by a traversal over vertices. Since the mapping and the reduction both traverse the vertices, a simple optimization is to perform them in the same pass. Now, we consider how path-based reductions can be implemented. We saw the iterative computation models in § 4. In the next subsections, we present how they can be instantiated to implement path-based reductions. We first present the correctness conditions of the iterative models to calculate path-based reductions (§ 6.1), and then present the synthesis of iteration kernel functions based on the correctness conditions (§ 6.2).

6.1 The Correctness of Iterative Path-Based Reduction

This subsection presents the iterative calculation of path-based reductions. We consider both the pull and push models with both idempotent and non-idempotent reduction. For each model, we present correctness and termination conditions.

Specification. Factored path-based reductions in the triple-let specifications have the form $\underset{c}{\mathcal{R}}\ \mathcal{F}$. Considering a general single reduction, c is either none $\perp$ or a source vertex s. The factored reduction for the former (with no source) is simply unrolled to $\mathcal{R}_{p \in \mathsf{Paths}(v)}\ \mathcal{F}(p)$ and the latter (with the source s) is unrolled to $\mathcal{R}_{p \in \{p \mid p \in \mathsf{Paths}(v)\ \wedge\ \mathsf{head}(p)=s\}}\ \mathcal{F}(p)$. Both of these reductions can be captured as the following general specification where the condition $C(p)$ is True for the former and is $\mathsf{head}(p) = s$ for the latter.

Definition 5 (Specification). $\mathcal{S}pec(v) = \mathcal{R}_{p \in \{p \mid p \in \mathsf{Paths}(v)\ \wedge\ C(p)\}}\ \mathcal{F}(p)$

The reduction function $\mathcal{R}$ is associative and commutative. It returns $\perp$ on an empty set, and returns the single element on a singleton set.

Model Instantiation. The iterative models (we saw in § 4) are parametric in terms of the kernel functions $\mathcal{I}$, $\mathcal{P}$, $\mathcal{R}$, and $\mathcal{E}$. We consider the correctness conditions on the kernel functions such that the iterative models calculate the specified path-based reduction. We will see in § 6.2 that these conditions will guide automatic synthesis of the kernel functions $\mathcal{I}$, $\mathcal{P}$, and $\mathcal{R}$ for a given path-based reduction.

Correctness. The iterative models calculate the value $\mathcal{S}^k(v)$ of each vertex v in iterations k by propagating the values of its neighbor vertices. The iteration stops when the value of no vertex changes. The values of the vertices $\mathcal{S}^k(v)$ are expected to converge to the specification $\mathcal{S}pec(v)$. We show the correctness in two steps. (1) First, we show that under certain conditions, at the end of each iteration k, the value $\mathcal{S}^k(v)$ of each vertex v is equal to the iteration specification $\mathcal{S}pec^k(v)$ for the iteration k. The specification $\mathcal{S}pec^k(v)$ is defined as the result of reduction over paths of length less than k.

Definition 6. $\mathcal{S}pec^k(v) = \mathcal{R}_{p \in \{p \mid p \in \mathsf{Paths}(v)\ \wedge\ C(p)\ \wedge\ \mathsf{length}(p)<k\}}\ \mathcal{F}(p)$

(2) Second, we show that under certain conditions, there is an index k where $\mathcal{S}pec^k(v)$ and $Spec^{k+1}(v)$ are equal with each other and $\mathcal{S}pec(v)$ as well. These two steps together show that the values of vertices $\mathcal{S}^k(v)$ eventually converge to $\mathcal{S}pec(v)$. We now consider the four variants of the iterative models. (The proofs of the theorems are available in the appendix § 3.1 and 4.4 .)

Pull Model. We consider the correctness of the pull model to calculate path-based reductions. We look at idempotent and non-idempotent reduction functions in turn.

The correctness of the pull models is dependent on the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ presented in Fig. 13. (A) The conditions $\mathbb{C}_1$ and $\mathbb{C}_2$ require the correctness of initialization function $\mathcal{I}$. If the path condition C holds on the simple initial path $\langle v, v\rangle$, the value of the initialization function $\mathcal{I}$ should be $\mathcal{F}(\langle v, v\rangle)$; otherwise, it should be none $\perp$. (B) The conditions $\mathbb{C}_3$ - $\mathbb{C}_5$ state the requirements for

the propagation function $\mathcal{P}$. The condition $\mathbb{C}_3$: It simply states that if the value of the vertex is none $\perp$, its propagated value should be none $\perp$ as well. The condition $\mathbb{C}_4$: We saw an illustration for $\mathbb{C}_4$ in Fig. 5a and Fig. 5b. For a path p, we call the value of $\mathcal{F}$ on p, the path value of p. The path $p \cdot e$ denotes the extension of the path p at the end with the edge e. Consider two paths p_1 and p_2 that end in a vertex u and there is an edge $\langle u, v\rangle$ from u to another vertex v. Reducing the path values of the two extended paths $p_1 \cdot \langle u, v\rangle$ and $p_2 \cdot \langle u, v\rangle$. should be the same as reducing the path values of p_1 and p_2, and then propagating the result with $\mathcal{P}$ through $\langle u, v\rangle$. Intuitively, this condition states that the local reduction and propagation effectively calculate reduction over paths. The condition $\mathbb{C}_5$: Vertices that have only a single incoming path do not receive multiple values to be reduced. For such vertices, $\mathbb{C}_5$ states that the propagation of the path value of p over an outgoing edge e is equal to the path value of the extended path $p \cdot e$. (C) The conditions $\mathbb{C}_6$ - $\mathbb{C}_9$ state the required properties of the reduction function $\mathcal{R}$. The none value $\perp$ should be the identity element, and $\mathcal{R}$ should be commutative, associative and idempotent. (The epilogue function $\mathcal{E}$ is instantiated to identity.) For example, given the factored path-based reduction $\min_s$ length for the shortest path use-case SSSP, the kernel functions that we saw in Fig. 7 satisfy the conditions above.

Pull model with idempotent reduction. The following theorem states that if the conditions above hold, then the value $\mathcal{S}^k_{\text{pull+}}(v)$ that the pull model with idempotent reduction (Def. 1) calculates complies with the specification $\mathcal{S}pec^k(v)$.

Theorem 2 (Correctness of Pull (idempotent reduction)). *For all $\mathcal{R}, \mathcal{F}, C, \mathcal{I}, \mathcal{P}$, and $k \geq 1$, if the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ hold, then $\mathcal{S}^k_{\text{pull+}}(v) = \mathcal{S}pec^k(v)$.*

The full proof is available in the appendix § 4.4.1. The proof is by induction on the iteration k. At the iteration $k = 1$, the specification $\mathcal{S}pec^1(v)$ requires reduction on only the paths of length zero to each vertex. Therefore, by the conditions $\mathbb{C}_1$ - $\mathbb{C}_2$, the initialization function $\mathcal{I}$ properly initializes each vertex v to $\mathcal{S}pec^1(v)$. In each iteration $k+1$, if there is any predecessor of the vertex v whose value is changed in the previous iteration k, then their new values are propagated by $\mathcal{P}$ and reduced together by $\mathcal{R}$, and then reduced with the current value of v. By the conditions $\mathbb{C}_7$ and $\mathbb{C}_8$, the reduction function $\mathcal{R}$ is commutative and associative, and can be applied to the propagated values in any order. By the induction hypothesis, the value of each predecessor u is the reduction of the paths to u of length l, $0 \leq l < k$. The predecessors that have no paths and store $\perp$ are ignored by the conditions $\mathbb{C}_3$ and $\mathbb{C}_6$. By the conditions $\mathbb{C}_4$ and $\mathbb{C}_5$, the propagation of the value of a predecessor u of the vertex v is equal to the reduction over the paths to v that pass through u. Since these paths include the edge (from u to v), their length l is $0 < l < k+1$. The previous value of v itself is the reduction over paths to v of length l, $0 \leq l < k$. Since, the reduction function $\mathcal{R}$ is idempotent, reducing these two values absorbs the values of the repeated paths, and results in the reduction over all paths of length l, $0 \leq l < k+1$ that the specification requires.

Pull model with non-idempotent reduction. The pull model with non-idempotent reduction $\mathcal{S}^k_{\text{pull}-}(v)$ is defined in Def. 2. We show that it can correctly calculate path-based reductions with non-idempotent (in addition to idempotent) reduction functions. For instance, consider the factored path-based reduction $\sum_s 1$ that counts the number of paths from the source s; the reduction function sum $\sum$ is non-idempotent. The initialization function is instantiated to $\mathcal{I} = \lambda v.\ 1$ and the propagation function is instantiated to $\mathcal{P} = \lambda n, e.\ n$ that simply propagates the value of the predecessor.

The following theorem states that if the conditions above except idempotency hold and the source vertex is not on any cycle then the pull model with non-idempotent reduction $\mathcal{S}^k_{\text{pull}-}(v)$ complies with the specification $\mathcal{S}pec^k(v)$.

Theorem 3 (Correctness of Pull (non-idempotent reduction)). *For all $\mathcal{R}, \mathcal{F}, \mathcal{I}, \mathcal{P}, k \geq 1$, and s, let $C(p) = (\mathsf{head}(p) = s)$, if $\mathbb{C}_1$ - $\mathbb{C}_8$ hold, and s is not on any cycle, $\mathcal{S}^k_{\mathsf{pull}-}(v) = \mathcal{S}pec^k(v)$.*

The full proof is available in the appendix § 4.4.2. Compared to the previous model, the reduction with the current value is avoided. However, no path is missed. The difference is only the paths of length 0. The vertices other than the source s do not have a path of length 0 from s. The source s itself is also correctly initialized to the value of $\mathcal{F}$ on the zero-length path $\langle s, s\rangle$, and since s is not on any cycle, its correct value is never overwritten.

Push Model. We now consider the correctness of the push model to calculate path-based reductions.

Push model with idempotent reduction. The following theorem states that if the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ hold, the value $\mathcal{S}^k_{\mathsf{push}+}(v)$ calculated by the push model with idempotent reduction, (Def. 3) complies with the specification $\mathcal{S}pec^k(v)$.

Theorem 4 (Correctness of push (idempotent reduction)). *For all $\mathcal{R}, \mathcal{F}, C, \mathcal{I}, \mathcal{P}$, and $k \geq 1$, if the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ hold, $\mathcal{S}^k_{\mathsf{push}+}(v) = \mathcal{S}pec^k(v)$.*

Push model with non-idempotent reduction. Similarly, the following theorem states the correctness of the push model with non-idempotent reduction (Def. 4).

A. Initialization:
$\mathbb{C}_1$: $\forall v.\ C(\langle v, v\rangle) \rightarrow \mathcal{I}(v) = \mathcal{F}(\langle v, v\rangle)$
$\mathbb{C}_2$: $\forall v.\ \neg C(\langle v, v\rangle) \rightarrow \mathcal{I}(v) = \bot$

B. Propagation:
$\mathbb{C}_3$ (None Propagation):
$\forall e.\ \mathcal{P}(\bot, e) = \bot$
$\mathbb{C}_4$ (Aggregate Propagation):
$\forall p_1, p_2, v.$
$\mathsf{tail}(p_1) = \mathsf{tail}(p_2) \rightarrow$
$\mathsf{let}\ u := \mathsf{tail}(p_1)\ \mathsf{in}$
$\mathcal{P}\ [\mathcal{R}(\mathcal{F}(p_1), \mathcal{F}(p_2)),\ \langle u, v\rangle] =$
$\mathcal{R}\ [\mathcal{F}(p_1 \cdot \langle u, v\rangle),\ \mathcal{F}(p_2 \cdot \langle u, v\rangle)]$
$\mathbb{C}_5$ (Single Path):
$\forall p, e.\ \mathcal{P}(\mathcal{F}(p),\ e) = \mathcal{F}(p \cdot e)$

C. Reduction:
$\mathbb{C}_6$ (Identity):
$\forall n.\ \mathcal{R}(n, \bot) = n$
$\mathbb{C}_7$ (Commutativity):
$\forall n, n'.\ \mathcal{R}(n, n') = \mathcal{R}(n', n)$
$\mathbb{C}_8$ (Associativity):
$\forall n, n', n''.\ \mathcal{R}(\mathcal{R}(n, n'), n'') = \mathcal{R}(n, \mathcal{R}(n', n''))$
$\mathbb{C}_9$ (Idempotency):
$\forall n.\ \mathcal{R}(n, n) = n$

Termination:
$\mathbb{C}_{10}$: $\forall p.\ \mathcal{R}(\mathcal{F}(p), \mathcal{F}(\mathsf{simple}(p))) = \mathcal{F}(\mathsf{simple}(p))$

Fig. 13. Correctness and Termination Conditions

Theorem 5 (Correctness of Push (non-idempotent reduction)). *For all $\mathcal{R}, \mathcal{F}, \mathcal{I}, \mathcal{P}, k \geq 1$, and s, let $C(p) = (\mathsf{head}(p) = s)$, if $\mathbb{C}_1$ - $\mathbb{C}_8$ hold, and s is not on any cycle, $\mathcal{S}^k_{\mathsf{push}-}(v) = \mathcal{S}pec^k(v)$*

Termination. We show that under certain conditions, there exists an iteration k where $\mathcal{S}pec^k(v)$ (Def. 6) stays unchanged and converges to the original specification $\mathcal{S}pec(v)$ (Def. 5). Iterations incrementally consider longer paths; however, longer paths do not necessarily yield new information. For example, in the shortest path use-case SSSP, after considering all the simple paths, the longer paths (that are cyclic) cannot lead to shorter paths (in graphs with non-negative edges). Given a path p, we call the path that results from removing its cycles the simplification $\mathsf{simple}(p)$ of p. In the shortest path use-case SSSP, the reduction function $\mathcal{R}$ is min and the path function $\mathcal{F}$ is weight. Reducing the $\mathcal{F}$ value of $\mathsf{simple}(p)$ with the $\mathcal{F}$ value of p results in the former. Therefore, simplified paths are enough to arrive at the same result for the reduction, and longer paths do not change the result. We capture this property as the condition $\mathbb{C}_{10}$ in Fig. 13 and prove convergence.

Theorem 6 (Termination). *For all $\mathcal{R}, \mathcal{F}$, and C, if the graph is acyclic or the condition $\mathbb{C}_{10}$ holds, then there exists k such that for every $k' \geq k$, $\mathcal{S}pec^{k'}(v) = \mathcal{S}pec(v)$.*

Let l be the length of the longest simple path to the vertex v. After the iteration $k = l + 1$, the value of $\mathcal{S}pec^k(v)$ stays unchanged. This is because the reduction with the paths of length greater

```
def Synth𝒫 (ℱ, ℛ)
  Let T be the return type of ℱ.
  memoize variable v for type T and size 1
  memoize variable l for type Edge and size 1
  foreach (literal l_i with type T_i)
    memoize l_i for T_i and size 1
  size ← 1
  while (true)
    E ← Candidates (T, size)
    foreach (e ∈ E)
      if ℱ;ℛ;Γ ⊢ (ℂ4 ∧ ℂ5)[𝒫 := (λv, l. e)]
        return (λv, l. e)
    size ← size + 1
```

(a)

```
P := List[V],
eweight: ⟨V, V⟩ → ℕ
weight: P → ℕ

∀p. if (p = ⊥)
      weight(p) = 0
    else
      let v := head(p), p′ := tail(p) in
      if (p′ = ⊥)  weight(p) = 0
      else
        let v′ := head(p′) in
        weight(p) = weight(p′) + eweight(⟨v′, v⟩)
```

(b)

Fig. 14. (a) Synthesis of the Propagation Function $\mathcal{P}$. (b) Context assertions Example.

than l does not change the value of $\mathcal{S}pec^k(v)$. If a path p that is longer than l exists, then p is not simple, i.e., it includes a cycle. This is refuted if the graph is acyclic. Otherwise, the simplification of p, simple(p), is already in the set of paths of length less than $l + 1$ and by the condition $\mathbb{C}_{10}$, reducing the path value of p with the path value of simple(p) results in the path value of simple(p).

An immediate corollary of the above two theorems is that iteration eventually terminates and converges to the specification $\mathcal{S}pec(v)$ (if the corresponding conditions in Theorem 2 to Theorem 5 hold). The final iteration is simply the maximum value of k from Theorem 6 for all vertices. For example, the corollary for the pull model for idempotent reduction functions is the following.

COROLLARY 7 (TERMINATION FOR PULL MODEL WITH IDEMPOTENT REDUCTION). *For all $\mathcal{R}$, $\mathcal{F}$, $\mathcal{C}$, $\mathcal{I}$, and $\mathcal{P}$, if the conditions $\mathbb{C}_1$ - $\mathbb{C}_9$ hold, and the graph is acyclic or the condition $\mathbb{C}_{10}$ holds, then there exists an iteration k such that $\mathcal{S}^k_{\text{pull+}}(v) = \mathcal{S}pec(v)$.*

We state the correctness conditions and prove similar theorems for all the models. The informal and formal proofs are available in the appendix § 3.1 and 4.4 respectively.

6.2 Synthesis of Iterative Reduction

Given a path-based reduction $\mathcal{R} \underset{c}{\mathcal{F}}$, we now use the correctness conditions presented in the previous subsection to automatically synthesize correct-by-construction kernel functions.

For example, consider the push iterative model with idempotent reduction that we saw in Fig. 8, Def. 3. By Theorem 4, we need to find the functions $\mathcal{I}$, $\mathcal{P}'$ and $\mathcal{R}'$ such that the conditions $\mathbb{C}_1$ - $\mathbb{C}_{10}$ (presented in Fig. 13) hold. We use these conditions to synthesize the functions $\mathcal{I}$, $\mathcal{P}'$ and $\mathcal{R}'$. In particular, (1) we use the initialization conditions $\mathbb{C}_1$ - $\mathbb{C}_2$ to synthesize $\mathcal{I}$, (2) we use the propagation conditions $\mathbb{C}_4$ and $\mathbb{C}_5$ to synthesize $\mathcal{P}$ and then wrap it in the following function $\mathcal{P}'$ to handle none $\perp$ values and satisfy the condition $\mathbb{C}_3$. The some value of v is denoted as $[v]$.

$$\mathcal{P}' := \lambda n, e.\ \text{if } (n = \perp) \text{ return } \perp \text{ else return } [\mathcal{P}(n, e)]$$

and (3) we check the conditions $\mathbb{C}_7$ - $\mathbb{C}_9$ for the reduction function $\mathcal{R}$, and then wrap $\mathcal{R}$ in the following reduction function $\mathcal{R}'$ to handle none $\perp$ values so that the condition $\mathbb{C}_6$ is satisfied. If the conditions $\mathbb{C}_7$ - $\mathbb{C}_9$ hold for $\mathcal{R}$, they hold for $\mathcal{R}'$ as well.

$$\mathcal{R}' := \lambda a, b.\ \text{if } (a = \perp) \text{ return } b \text{ else if } (b = \perp) \text{ return } a \text{ else return } [\mathcal{R}(a, b)]$$

To find candidate expressions for the body of $\mathcal{I}$ and $\mathcal{P}$, we apply a type-guided enumerative search. It enumerates expressions from the grammar that we saw in Fig. 7a in the order of increasing size. To support overloaded operators, the expression constructors have union types. To synthesize an expression of the given type, the search only considers expression constructors that return that type. Given the parameter types of the constructor, it then recursively searches for the arguments, and uses memoization to avoid redundant enumeration.

The procedure Synth$\mathcal{P}$ that synthesizes $\mathcal{P}$ is shown in Fig. 14a. (The synthesis of the other kernel functions is similar.) It starts by memoizing expressions of size one, literals and variables, to make them available for the synthesis of the body of $\mathcal{P}$. Let T be the return type of $\mathcal{F}$; thus, vertices store values of type T. The propagation function $\mathcal{P}$ takes a value stored at a vertex (of type of T) and an edge (of type Edge) and returns a vertex value (of type T). Thus, the two input parameters of type T and Edge are memoized as available expressions. Then, candidate bodies for $\mathcal{P}$ (of type T) of increasing sizes are obtained.

A candidate is correct if the condition $\mathbb{C}_4$ and $\mathbb{C}_5$ are valid when $\mathcal{P}$ is replaced with the candidate. To check the validity of an assertion, we use off-the-shelf SMT solvers to check the satisfiability of its negation. The context of the validity check $\mathcal{F}; \mathcal{R}; \Gamma$ is the definition of the functions $\mathcal{F}$ and $\mathcal{R}$ from the given path-based reduction, and a set of assertions Γ that define basic graph functions and relations. We model paths P as lists of vertices V, and define graph functions and relations including the path functions length, weight, punultimate and capacity in the combination of the quantified uninterpreted functions and list theories. Fig. 14b showcases the axiomatization of the weight function in Γ. (More assertions are available in the appendix § 3.2.) The edge-weight eweight is a function on pairs of vertices $\langle \mathsf{V}, \mathsf{V} \rangle$, and the path weight weight is a function on paths P to natural numbers $\mathbb{N}$. If the list for the path is empty or has a single vertex, the weight of the path is trivially zero; otherwise, the weight of the path is recursively the sum of the weight of the path without the last edge, and the edge-weight of the last edge.

For termination, we check a stronger condition than $\mathbb{C}_{10}$. We remove an edge instead of a cycle: for every path p and edge e, if reducing the $\mathcal{F}$ value of p with the $\mathcal{F}$ value of $p \cdot e$ results in the former, then the reduction is terminating. (For the push variant that requires rollback, after the propagation function $\mathcal{P}$ is synthesized, a correctness condition on $\mathcal{P}$ and the rollback function $\mathcal{B}$ is used to synthesize $\mathcal{B}$ (appendix § 3.1.2).)

7 EXPERIMENTAL RESULTS

Implementation. We implemented the Grafs synthesis tool in three parts: fusion, synthesis and backends. The fusion phase closely follows the fusion rules (of § 5.2) using the visitor pattern. The synthesis phase uses the Z3 SMT solver to check the validity of the correctness conditions. Grafs incorporates a dedicated backend for each framework. Each backend generates a framework-specific C++ file containing the initialization $\mathcal{I}$, propagation $\mathcal{P}$, (if needed rollback $\mathcal{B}$) and reduction function $\mathcal{R}$. (The different mappings for each of the target frameworks are presented in the appendix § 5.1.) Grafs can be modularly extended with backends for new frameworks.

Platform and Benchmarks. We performed the experiments on a 4-node cluster, each with 32 cores and 64GB memory. The experiments for frameworks that are exclusively for shared memory are performed on one of these nodes. The nodes are connected via 40Gbps InfiniBand network, and they run CentOS 7.4 Linux x86_64 kernel version 3.10. All programs are compiled with gcc-5.1.0 (for Ligra, GridGraph and GraphIt) and mpich-3.2.1 and openmpi-3.0 (for PowerGraph and Gemini, respectively). We used social network graphs of various sizes: LiveJournal (LJ, 1.1GB), Twitter (TW, 23GB), TwitterMPI (TM, 28GB), and Friendster (FR, 31GB). We report the average of 5 executions.

Results Summary. To evaluate the Grafs synthesis tool, we study the effect of fusion on performance. Then, we study the effect of different fusion types separately. The experiments show

Table 1. Execution times (in seconds). H: Handwritten, S: Synthesized, R: the ratio H/S.

Prog.	Input	Ligra			GridGraph			Gemini			PowerGraph (Push)			PowerGraph (Pull)			GraphIt (Push)		
		H	S	**R**	H	S	**R**	H	S	**R**	H	S	**R**	H	S	**R**	H	S	**R**
DRR	LJ	1.03	0.33	**3.1**	15.3	3.8	**4**	1.2	0.4	**3**	20.4	6.4	**3.2**	36	10	**3.6**	0.63	0.21	**3**
	TW	-	-	-	82	23	**3.6**	7.2	3.7	**2**	120	48	**2.5**	292	81	**3.6**	16.3	5.4	**3**
	TM	-	-	-	141	44	**3.3**	8.9	3.8	**2.3**	166	50	**3.3**	462	86	**2.9**	19	6	**3.1**
	FR	-	-	-	265	73	**3.6**	13	4.9	**2.6**	247	86	**2.9**	522	154	**3.4**	30	9.3	**3.2**
Trust	LJ	1	0.36	**2.7**	14.1	4.4	**3.2**	1.18	0.62	**1.9**	20.5	7.5	**2.7**	37	10	**3.7**	0.61	0.29	**2.1**
	TW	-	-	-	85	28	**3.1**	6.2	5	**1.2**	122	50	**2.4**	293	99	**2.9**	16.1	8.3	**1.9**
	TM	-	-	-	122	40	**3**	7.5	5.6	**1.3**	157	71	**2.2**	455	129	**3.5**	17.2	6.9	**2.5**
	FR	-	-	-	218	117	**2**	10.1	6.7	**1.5**	252	98	**2.6**	526	173	**3**	28	13	**2.1**
LTrust	LJ	1.02	0.63	**1.6**	11.5	6	**1.9**	1.13	0.86	**1.3**	23	15.1	**1.5**	41	28	**1.4**	0.59	0.39	**1.5**
	TW	-	-	-	74	47	**1.6**	6.1	4.9	**1.2**	130	108	**1.2**	-	-	-	16	11.4	**1.4**
	TM	-	-	-	142	82	**1.7**	7.3	6	**1.2**	200	134	**1.5**	-	-	-	10	20.1	**2**
	FR	-	-	-	210	175	**1.2**	10.2	8.8	**1.1**	286	198	**1.5**	-	-	-	34	24.9	**1.3**

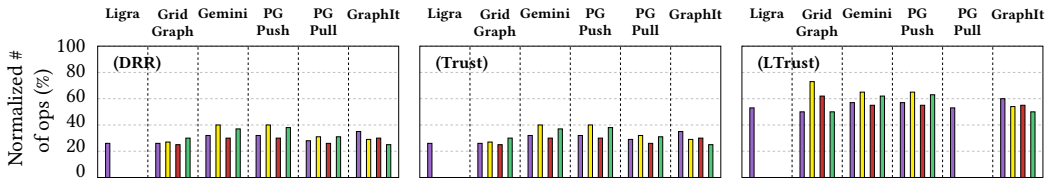


Fig. 15. Edge-work Ratio: Normalized # of edges processed by the fused over the unfused version. Missing bars correspond to programs not successfully running on input graphs.

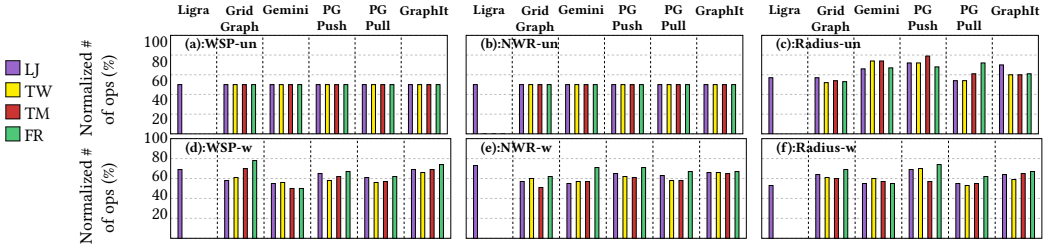


Fig. 16. Edge-work ratio: # of edges processed by the fused over the unfused version. Missing bars correspond to programs not successfully running on inputs. (Absolute execution times are available in the appendix § 6.2.)

that fusion can lead up to 4× and in average 2.4× faster execution time compared to the unfused codes. We then report the number of lines of code for the specifications and their synthesis time. Compared to existing frameworks, GRAFS allows significantly more concise specifications, and can efficiently generate code in less than two minutes. Finally, we compare the synthesized code with handwritten versions for use-cases available in the frameworks. Experimental results show that the synthesized code either matches or outperforms handwritten code. We then show that synthesized programs scale similar to handwritten programs. (More experiments including the scalability of fusion on different number of sources for RADIUS are available in the appendix § 6.)

The Effect of Fusion. We study the performance benefits of fusion on the more elaborate use-cases: TRUST, DRR, and LTRUST (presented in Fig. 6). The absolute execution times and edge-work ratio are presented in Table 1 and Fig. 15 respectively. The goal of these measurement is to

compare handwritten and synthesized programs. Therefore, for each pair, we fix the framework, its configuration and the iterative model. These measurements are not meant to compare frameworks with each other, as fine-tuning configurations is framework-specific and orthogonal to the goal of these experiments. The number of edges processed by a program indicates the number of times that propagation happens across edges; thus, it represents the amount of computation performed throughout the execution. The edge-work ratio is the number of edges processed by the synthesized (fused) programs normalized w.r.t. that by the unfused versions. The experimental results show that fusion reduces the edge-work ratio up to a quarter and leads to up to 4× speedup. These use-cases benefit from fusion rules for path-based and vertex-based reductions, common operation elimination (appendix § 2.2.1) and factoring of nested vertex-based reductions (appendix § 2.2.6).

DRR. DRR calculates the ratio of the diameter over radius sampled over two sources. In addition to the rules FMPair, FRPair and FLetsBin which fuse path-based and vertex-based reductions, common operation elimination factors redundant path-based computations in diameter and radius. Therefore, instead of 4 reductions, Grafs fuses and calculates 1 reduction. (The complete fusion steps are available in the appendix § 2.3.) In Fig. 15, we observe that the edge-work ratio is 25-40%. This translates to 2-4× speedup in Table 1.

Trust. Trust specifies the trust from a given set of sources to other vertices. It applies division and maximum operator between path-based reductions: the widest and shortest paths. The rules FILetBin and FMPair fuse the 4 path-based reductions to 1. As Fig. 15 shows, the edge-work ratio is 25-40%, and as Table 1 shows, the speedup is 1.2-3.7×. We note that the theoretical bound on the edge-work ratio for both DRR and Trust is 25%, which happens when the path-based computations for the two sources fully overlap.

LTrust. Given a source s, LTrust calculates the narrowest of the widest paths to vertices within the distance Radius from s. LTrust has a nested reduction for Radius that is factored and then fused. Moreover, the two path-based reductions, the narrowest and shortest paths, are fused by the rules FILetBin and FMPair. This results in a sequence of two iteration-map-reduce rounds. The unfused and fused programs perform four and two sequences of iteration-map-reduce rounds respectively. The theoretical bound for the edge-work ratio is 50% . Fig. 15 shows that the edge-work ratio is 57-85% which translates to 1.1-2× speedup in Table 1.

Fusion Types. In order to study the performance benefits of different types of fusion rules, we compare the unfused and the fused implementations of three representative use-cases: WSP, NWR and Radius (presented in Fig. 6). Fig. 16 shows the edge-work ratio. We visit the use-cases and the applied fusion rules in turn.

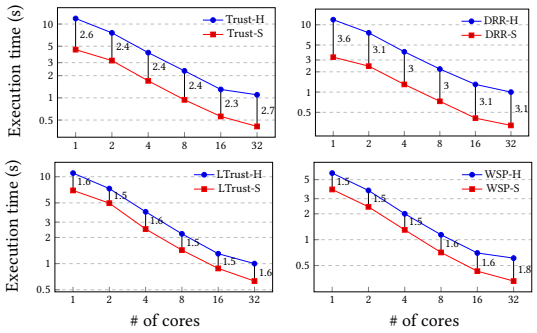
WSP. The unfused program for WSP consists of two computation phases over the edges of the input graph, one after the other. The first calculates the shortest paths from the given source to all the vertices, and the second computes the capacity of the widest path across the shortest paths. WSP is fused by the rule FPNest that fuses nested path-based reductions. The fused program executes the two computations above in one pass over a pair of values.

Assessment. Fig. 16a and Fig. 16d show the edge-work ratio of WSP for unweighted and weighted graphs respectively. In unweighted graphs, the fused program processes half the number of edges processed by the unfused program (50% ratio). The ratio is 50-70% for the weighted graphs. When graphs are unweighted, each edge represents a unit cost (either weight or capacity). In each iteration, the set of edges that contribute to the weight and capacity values of a vertex are the same. The fused program exploits this overlap by simultaneously propagating the two values across each edge. However, for weighted graphs, the two values can be propagated to the vertex in different iterations resulting in different shortest and widest paths. Hence, the fused program exploits the partial overlap between the processed edges.

Usecase	#PBR[1]	T[2]	GF[3]	L[4]	GG[5]	G[6]	PG[7]	GI[8]
BFS	1	25	1	32	100	185	280	130
CC	1	1	1	47	60	117	210	130
SSSP	1	24	1	66	73	117	280	133
WP	1	29	1	66	73	117	280	133
WSP	2	44	2	85	95	197	294	40
NWR	2	58	2	80	95	222	294	40
RADIUS	2	49	2	38	65	222	294	40
DRR	4	50	3	65	110	213	561	111
TRUST	4	105	3	105	145	300	481	96
LTRUST	4	102	4	115	148	302	481	107

[1] # of path-based reductions
[2] Synthesis time (s)
[3] GF: LoC in GRAFS
[4] L: LoC in Ligra
[5] GG: LoC in GridGraph
[6] G: LoC in Gemini
[7] PG: LoC in PowerGraph
[8] GI: LoC in GraphIt

(a)



(b)

Fig. 17. (a): Synthesis time and the number of lines of code, (b): Scalability on Ligra. X-axis: # of cores. Y-axis: time is logarithmic scale. H: Handwritten. S: Synthesized.

NWR. The unfused version of NWR calculates the narrowest and the widest paths separately. The two are fused by the rule FMPAIR that fuses multiple path-based reductions into one. It fuses the two reduction functions to one reduction function that operates on pairs. The fused propagation function passes the narrowest and widest values over an edge at the same time.

Assessment. Fig. 16b and Fig. 16e show the edge-work ratio for NWR for unweighted and weighted graphs respectively. Similar to WSP, the fused program reduces the number of processed edges to 50% for unweighted graphs and to 51-73% for weighted graphs.

RADIUS. RADIUS computes eccentricity (i.e. the maximum shortest distance) by sampling over two sources. The unfused version computes eccentricity separately for each source. However, RADIUS is fused by the rule FMPAIR (that we considered above) and the rule FRPAIR which fuses multiple vertex-based reductions into a single reduction.

Assessment. Fig. 16c and Fig. 16f show the edge-work ratio for RADIUS for unweighted and weighted graphs respectively. We observe that on unweighted graphs, the edge-work ratio is 52-78%. This ratio is 53-74% on weighted graphs. Even though fusion enables computation of multiple eccentricity values at the same time, contrary to WSP and NWR, we do not observe the 50% reduction. This is because eccentricity computations across different sources can occur via non-overlaping paths. The fused version exploits the partial overlaps.

We observe that the reduction in edge computations is different across different frameworks as well. For example, the edge-work ratio is 52-68% in GridGraph, whereas 54-78% in PowerGraph. This is because of the difference in the scheduling strategies across these different frameworks, that lead to different overlaps in edge computations. (The absolute execution times for the use-cases above, WSP, NWR and RADIUS, are available in the appendix § 6.2.)

Synthesis Time and LoC. Fig. 17a presents the synthesis time for the use-cases. Synthesis is done in less than 2 minutes and often less. It also compares the lines of code (LoC) that user should write in GRAFS and the other five frameworks. For each use-case, it reports the number of lines of code of the functions or struct definitions where a change is needed for that use-case. We observe that the GRAFS specifications are significantly smaller.

Synthesized Matching Handwritten. We compared the performance of the synthesized programs and their equivalent handwritten programs on five use-cases BFS, CC, SSSP, WP (widest path) and PR. We adopted the handwritten implementations of BFS, CC, SSSP and PR that are available in the frameworks, and developed WP based on SSSP by updating the path function. Table 2 shows the execution times of the handwritten programs (H) and synthesized programs (S),

Table 2. Execution Times (in seconds). H: Handwritten, S: Synthesized, R: the ratio H/S, ER: edge-work ratio. Missing cells are due to either missing handwritten use-cases (PR) or not successfully running on an input

Prog.	Input	Ligra				GridGraph				Gemini				PowerGraph (Push)				PowerGraph (Pull)				GraphIt (Push)			
		H	S	R	ER	H	S	R	ER	H	S	R	ER	H	S	R	ER	H	S	R	ER	H	S	R	ER
BFS	LJ	0.38	0.37	**1.02**	**1**	1.56	1.56	**1**	**1**	0.38	0.39	**0.99**	**1**	5.9	5.6	**1.04**	**1**	10.1	9.3	**1.09**	**1**	0.16	0.15	**1.06**	**1**
	TW	8.6	8.7	**0.98**	**1**	210	195	**1.07**	**1**	2.9	2.9	**1**	**1**	33.7	30.8	**1.1**	**1**	69.6	64.2	**1.08**	**1**	4.6	3.8	**1.2**	**1**
	TM	7.1	7	**1.01**	**1**	487	472	**1.03**	**1**	3.1	3.2	**0.96**	**1**	48.8	43.4	**1.12**	**1**	108.9	106.3	**1.02**	**1**	4.1	4.1	**1**	**1**
	FR	-	-	-	-	521	532	**0.97**	**1**	3.7	3.9	**0.96**	**1**	69.9	64.8	**1.07**	**1**	131	118	**1.11**	**1**	7.8	8.2	**0.95**	**1**
CC	LJ	0.36	0.38	**0.94**	**1**	2.21	2.22	**0.99**	**1**	0.77	0.77	**1**	**1**	13	10	**1.3**	**0.45**	19.6	18.9	**1.03**	**1**	0.18	0.19	**0.94**	**1**
	TW	21	20	**1.05**	**1**	230	214	**1.07**	**1**	4.8	4.9	**0.98**	**1**	84.4	69.6	**1.21**	**0.33**	122.6	119.1	**1.02**	**1**	6.3	7.5	**0.84**	**1**
	TM	13	15	**0.86**	**1**	432	423	**1.02**	**1**	7.5	7.6	**0.98**	**1**	160.3	129.7	**1.23**	**0.45**	262.7	241.4	**1.08**	**1**	6.1	6.1	**1**	**1**
	FR	-	-	-	-	606	599	**1.01**	**1**	14	14.3	**0.98**	**1**	259.1	200.7	**1.3**	**0.43**	292.3	293	**0.99**	**1**	12.2	11.7	**1.04**	**1**
SSSP	LJ	0.54	0.57	**0.94**	**1**	2.42	2.1	**1.15**	**1**	0.45	0.49	**0.9**	**1**	7.3	7.3	**1**	**1**	13.3	13.1	**1.01**	**1**	0.2	0.22	**0.9**	**1**
	TW	-	-	-	-	201	205	**0.98**	**1**	2.8	2.8	**1**	**1**	36.1	35	**1.03**	**1**	87.6	84.6	**1.03**	**1**	4.4	4.7	**0.93**	**1**
	TM	-	-	-	-	490	487	**1**	**1**	2.8	3	**0.94**	**1**	47.5	48.8	**0.97**	**1**	137.4	125.3	**1.09**	**1**	5	5.3	**0.94**	**1**
	FR	-	-	-	-	572	570	**1**	**1**	5	5.4	**0.92**	**1**	96.1	90.9	**1.05**	**1**	176.9	182.9	**0.96**	**1**	12.6	14	**0.9**	**1**
WP	LJ	0.61	0.64	**1.04**	**1**	3.46	3.2	**1.08**	**1**	0.45	0.47	**0.97**	**1**	7.8	7.9	**0.98**	**1**	15.1	14.7	**1.02**	**1**	0.25	0.2	**1.25**	**1**
	TW	-	-	-	-	245	242	**1.01**	**1**	3	3	**1**	**1**	36.4	36.4	**1**	**1**	93.2	91.68	**1.01**	**1**	5.5	5	**1.1**	**1**
	TM	-	-	-	-	479	498	**0.96**	**1**	3.2	3.2	**1**	**1**	57.6	54	**1.06**	**1**	175.7	160.5	**1.09**	**1**	8.2	7.5	**1.09**	**1**
	FR	-	-	-	-	551	545	**1.01**	**1**	5.5	5.8	**0.95**	**1**	86.3	97.2	**0.88**	**1**	198.4	225.8	**0.87**	**1**	10.9	9.6	**1.13**	**1**
PR	LJ	19.5	19	**1.01**	**1**	44	37	**1.1**	**1**	21	21	**1**	**1**	-	-	-	-	80	80	**1**	**1**	11.8	11.4	**1.03**	**1**
	TW	673	664	**1**	**1**	1000	908	**1.1**	**1**	282	400	**0.7**	**1**	-	-	-	-	1128	1041	**1.08**	**1**	319	331	**0.96**	**1**
	TM	597	646	**0.92**	**1**	1399	1441	**0.97**	**1**	880	860	**1.02**	**1**	-	-	-	-	1157	1078	**1.07**	**1**	596	613	**0.97**	**1**
	FR	-	-	-	-	1023	995	**1.02**	**1**	590	577	**1.02**	**1**	-	-	-	-	601	548	**1.09**	**1**	260	280	**0.93**	**1**

Table 3. Metrics for Comparing Handwritten and Synthesized Code. H: Handwritten, S: Synthesized. (PowerGraph does not require the user to write atomic operations.)

Prog.	Vertex Data Size (bytes) :: Edge Data Size (bytes)										# Atomics Per Edge									
	Ligra		GridGraph		Gemini		PowerGraph		GraphIt		Ligra		GridGraph		Gemini		PowerGraph		GraphIt	
	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S
BFS	8::0	8::0	8::0	8::0	8::0	8::0	12::0	12::0	4::0	8::0	1	1	1	1	1	1	0	0	1	1
CC	4::0	4::0	4::0	4::0	4::0	4::0	8::0	8::0	4::0	4::0	1	1	1	1	1	1	0	0	1	1
SSSP	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	1	1	1	1	1	1	0	0	1	1
WP	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	4::4	1	1	1	1	1	1	0	0	1	1
PR	4::0	4::0	4::0	4::0	8::0	8::0	8::0	8::0	4::0	4::0	1	1	1	1	1	1	0	0	0	0

and their relative ratio (R), i.e., former divided by the latter. It also reports the edge-work ratio (ER) that is the number of edges processed by the synthesized programs divided by that processed by the handwritten versions. (The PR use-case was run until convergence.) Although the execution time is primarily dependent on the number of processed edges, it is also dependent on the efficiency of the kernel functions, which is influenced by the number of vertex and edge variables and atomic operations. To have a more precise comparison, in Table 3, we further compare the number of atomic operations per edge computation, and the state maintained per vertex and edge, which are two key factors for efficiency of graph computations.

Assessment. We observe in Table 2 that the synthesized programs process the same number of edges compared to handwritten programs in Ligra, GridGraph and Gemini. On PowerGraph (for the use-case CC in the push model), the synthesized program process fewer edges. This is due to unnecessary processing of all the edges in the first iteration in the handwritten program which the synthesized version avoids. We also observe that the execution time is closely related to the number of processed edges. The performance of the handwritten and synthesized code is similar in most cases. The synthesized CC for PowerGraph in the push model performs 28% faster. We observe in Table 3 that the number of atomic operations per edge is exactly the same as that in the handwritten programs, and the size of the state per vertex and edge is minimal.

Scalability. Fig. 17b shows the scalability of both the handwritten and synthesized code on the Ligra framework and LJ input graph for four use-cases: Trust, DRR, LTrust, and WSP. The speedup remains steady around 2.5×, 3.1×, 1.5× and 1.5× respectively. As we saw in § 5, fusion

preserves, and furthermore increases, the parallelism of specifications. Moreover, the synthesized codes retain the edge- and vertex-level parallelism offered by the frameworks. They never rely on major synchronization bottlenecks (e.g., locking multiple edges or vertices at the same time). Thus, synthesized programs scale similar to handwritten programs.

8 RELATED WORK

Graph Processing Frameworks. Graph processing systems provide interfaces to hide the implementation details such as parallelism, synchronization and communication in scalable runtimes. At the heart of graph computations are operations over vertex and edge values and scheduling policies to determine the order in which operations are performed. Parallelism is often extracted at the vertex and edge level, and hence, most interfaces allow computations to be directly expressed as vertex-level and edge-level operations [Dathathri et al. 2018; Gonzalez et al. 2012; Grossman et al. 2018; Hoang et al. 2019; Low et al. 2012, 2014; Malewicz et al. 2010; Mariappan et al. 2021; Mariappan and Vora 2019; Nguyen et al. 2013; Roy et al. 2013; Shun and Blelloch 2013; Vora 2019; Vora et al. 2017; Zhang et al. 2018; Zhu et al. 2016, 2015]. Certain DSLs raise the abstraction level by expressing the operation in the form of sequential programs or datalog queries, in order to simplify development of graph algorithms [Aberger et al. 2017; Hong et al. 2012; Rodriguez 2015; Sevenich et al. 2016; van Rest et al. 2016; Zhang et al. 2018]. Others [Cheramangalath et al. 2017; Gill et al. 2018; Shashidhar and Nasre 2016] focus on generating implementations of graph algorithms for different architectures such as GPUs. Unlike our synthesis process that generates codes for multiple graph processing frameworks, these systems generate implementations that are tied to their runtime specifics. Moreover, GRAFS synthesizes the kernel functions.

Declarative Graph Processing DSLs. Fregel [Emoto et al. 2016] is a domain-specific language that allows graph computations to be expressed as a higher-order function that is applied at every vertex. Its latest version compiles code to the Giraph and Pregel+. Similar to GRAFS, Fregel is declarative, models termination conditions, and applies optimizing transformations (such as tupling). Following Fregel, Palgol [Zhang et al. 2017] extends Fregel's functional interface with remote data access. Similarly, s6graph [Coll Ruiz et al. 2016] is a graph processing framework with a functional interface and dedicated runtime. In addition to a vertex-centric intermediate language, GRAFS presents a higher-level language for path-based computations and its semantics, formally models a comprehensive set of the common iterative models and proves the formal correctness and termination conditions for them, captures the canonical iteration-map-reduce primitives as a let form and presents several fusion optimization types that transform specifications into these primitives, combines type-directed enumerative and constrained-based synthesis to generate the iterative kernel functions, and generates implementations in five graph processing frameworks.

Elixir [Prountzos et al. 2012, 2015] captures a graph computation as an operator on a graph neighborhood that is iteratively applied to the graph non-deterministically. It allows declarative constraints for scheduling, implementation selection, and synchronization insertion into the operators and applies automated planning to find multiple implementations. LM [Cruz et al. 2014] and CLM [Cruz et al. 2016] present a logic programming language for programming over graph structures and algorithms. Similar to Elixir, CLM supports declarative specification of scheduling and partitioning policies that allows programmers to add logical rules for optimization. In contrast, GRAFS offers a more high-level specification language for path-based computations, applies fusion optimizations, formalizes correctness and termination conditions for iterative computations, and uses them to automatically synthesize the iterative kernel functions.

To simplify constructing and reasoning about programs, declarative programming [Rocha and Launchbury 2011] is applied to many domains such as compiler optimization [Lindig and Ramsey 2004], parallel programming [Cruz et al. 2014] and configuration generation [Hewson et al. 2012].

Program Synthesis. Program synthesis has always been an area of interest for computer scientists. Previous works have employed enumeration [Itzhaky et al. 2010; Udupa et al. 2013], variants of syntax-guided synthesis [Alur et al. 2013] and type-guided synthesis [Osera and Zdancewic 2015; Polikarpova et al. 2016] to synthesize protocol snippets [Udupa et al. 2013] and Excel macros [Gulwani 2011; Gulwani et al. 2012]. Grafs's synthesis process enumerates graph processing kernel functions based on a syntax grammar for local computations.

Previous works have also used constraint solving to fill holes in program sketches [Solar-Lezama et al. 2005, 2006] including architectural kernel functions [Xu et al. 2014], and to synthesize control structures, imperative programs [Feng et al. 2017; Srivastava et al. 2010] and program templates [Barman et al. 2015], and to compose APIs [Jha et al. 2010; Shi et al. 2019]. The Grafs synthesis tool applies SMT solvers to check that the candidate kernel functions satisfy the correctness conditions of the iterative models. Built on top of Fregel, [Morihata et al. 2018] uses SMT solvers to optimize kernel functions. In contrast, Grafs automatically synthesizes the kernel functions.

Superoptimization is another thread of synthesis which applies stochastic search methods to synthesize programs [Bansal and Aiken 2006; Joshi et al. 2002, 2006; Massalin 1987; Schkufza et al. 2013]. Moreover, Souper [Sasnauskas et al. 2017] took a step further by synthesizing superoptimizers. In contrary to superoptimization which focuses on optimizing machine-level code, Grafs fusion rules optimize high-level graph processing specifications.

Distributed and concurrent program synthesis. Bigλ [Smith and Albarghouthi 2016] synthesizes map-reduce-style distributed programs and SCYTHE [Wang et al. 2017] synthesizes SQL queries based on the programming-by-example approach. Hamsaz [Houshmand and Lesani 2019] minimizes and synthesizes coordination between replicas in a distributed system. Transit [Udupa et al. 2013] describes a distributed protocol as both symbolic and concrete execution fragments called concolic snippets, and applies solvers and user feedback to interactively generate the implementation. All three works above have different synthesis domains than Grafs.

Previous works have synthesized concurrent programs either by inferring atomic sections and inserting synchronization primitives [Bloem et al. 2014; Cherem et al. 2008; Cunningham et al. 2008; Halpert et al. 2007; Vechev and Yahav 2008; Vechev et al. 2010], or by following semantic preserving rules to transform sequential to concurrent programs [Cerny et al. 2017, 2013, 2014].

Fusion. Fusion is a versatile optimization technique. Loop fusion [Bondhugula et al. 2008; Darte 1999; Kennedy and McKinley 1993; Qasem and Kennedy 2006] merges the bodies of loops on regular structures such as arrays and hence reduces the number of memory accesses and improves locality. Fusion also has been applied to tree structures [Rajbhandari et al. 2016a,b; Sakka et al. 2017, 2019] to combine multiple phases of traversal or fuse different stages of data processing pipelines [Saarikivi et al. 2017] to enhance data locality. Deforestation of functional programs [Chin 1992; Gill et al. 1993; Johann and Visser 2000; Wadler 1988] combines a sequence of function applications into a single application and eliminates intermediate values. However, deforestation is oblivious to the primitives of graph computation. Graph computations use three fundamental primitives; thus, Grafs structures these primitives as the triple-let term. The fusion rules transform computations to this structure and maintain it during fusion.

9 CONCLUSION

We saw Grafs, a declarative graph analytics language and synthesizer. It features semantics-preserving fusion optimizations. It automatically synthesizes kernel functions based on correctness conditions for iterative reductions, and generates code for high-performance graph processing frameworks. We hope that Grafs motivates fundamental research in declarative languages and automatic synthesis techniques and tools for graph analytics and broadly data mining.

REFERENCES

Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 20. https://doi.org/10.1145/3129246

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.

Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *ACM Sigplan Notices*, Vol. 41. ACM, 394–403. https://doi.org/10.1145/1168857.1168906

Shaon Barman, Rastislav Bodik, Satish Chandra, Emina Torlak, Arka Bhattacharya, and David Culler. 2015. Toward tool support for interactive synthesis. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 121–136. https://doi.org/10.1145/2814228.2814235

Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spörk. 2014. Synthesis of Synchronization Using Uninterpreted Functions. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design* (Lausanne, Switzerland) *(FMCAD '14)*. FMCAD Inc, Austin, TX, Article 11, 8 pages. http://dl.acm.org/citation.cfm?id=2682923.2682937

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

Pavol Cerny, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. 2017. From Non-preemptive to Preemptive Scheduling Using Synchronization Synthesis. *Form. Methods Syst. Des.* 50, 2-3 (June 2017), 97–139. https://doi.org/10.1007/s10703-016-0256-5

Pavol Cerny, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2013. Efficient Synthesis for Concurrency by Semantics-Preserving Transformations. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 951–967.

Pavol Cerny, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2014. Regression-Free Synthesis for Concurrency. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 568–584.

Unnikrishnan Cheramangalath, Rupesh Nasre, and Y N. Srikant. 2017. DH-Falcon: A Language for Large-Scale Graph Processing on Distributed Heterogeneous Systems. 439–450. https://doi.org/10.1109/CLUSTER.2017.72

Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. 2008. Inferring locks for atomic sections. *ACM SIGPLAN Notices* 43, 6 (2008), 304–315.

Wei-Ngan Chin. 1992. Safe fusion of functional expressions. In *ACM SIGPLAN Lisp Pointers*. ACM, 11–20.

Onofre Coll Ruiz, Kiminori Matsuzaki, and Shigeyuki Sato. 2016. s6raph: vertex-centric graph processing framework with functional interface. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. 58–64.

Flavio Cruz, Ricardo Rocha, and Seth Copen Goldstein. 2016. Declarative coordination of graph-based parallel programs. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

Flavio Cruz, Ricardo Rocha, Seth Copen Goldstein, and Frank Pfenning. 2014. A linear logic programming language for concurrent programming over graph structures. *Theory and Practice of Logic Programming* 14, 4-5 (2014), 493–507.

Dave Cunningham, Khilan Gudka, and Susan Eisenbach. 2008. Keep off the grass: Locking the right path for atomicity. In *International Conference on Compiler Construction*. Springer, 276–290.

Alain Darte. 1999. On the complexity of loop fusion. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*. IEEE, 149–157.

Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 752–768. https://doi.org/10.1145/3192366.3192404

Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. 2016. Think like a vertex, behave like a function! a functional DSL for vertex-centric big graph processing. *ACM SIGPLAN Notices* 51 (09 2016), 200–213. https://doi.org/10.1145/3022670.2951938

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017. Component-based synthesis for complex APIs. *ACM SIGPLAN Notices* 52, 1 (2017), 599–612.

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. 2018. Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, Cham, 249–264. http://doi.org/10.1007/978-3-319-96983-1_18

Jennifer Ann Golbeck. 2005. *Computing and applying trust in web-based social networks*. Ph.D. Dissertation.

Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 17–30.

Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making pull-based graph processing performant. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 246–260. https://doi.org/10.1145/3200691.3178506

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105. https://doi.org/10.1145/2240236.2240260

Richard L Halpert, Christopher JF Pickett, and Clark Verbrugge. 2007. Component-based lock allocation. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*. IEEE, 353–364.

John A Hewson, Paul Anderson, and Andrew D Gordon. 2012. A Declarative Approach to Automated Configuration.. In *LISA*, Vol. 12. 51–66.

Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. 2019. A round-efficient distributed betweenness centrality algorithm. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 272–286.

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 349–362.

Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: replication coordination analysis and synthesis. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 74.

Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. 2010. A simple inductive synthesis methodology and its applications. In *ACM Sigplan Notices*, Vol. 45. ACM, 36–46.

Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 215–224.

Patricia Johann and Eelco Visser. 2000. Warm fusion in Stratego: A case study in generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence* 29, 1 (2000), 1–34.

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. *Denali: a goal-directed superoptimizer*. Vol. 37. ACM.

Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A practical algorithm for generating optimal code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 6 (2006), 967–989.

Ken Kennedy and Kathryn S McKinley. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 301–320.

Christian Lindig and Norman Ramsey. 2004. Declarative composition of stack frames. In *International Conference on Compiler Construction*. Springer, 298–312.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '21)*. 1–16.

Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '19)*. 1–16.

Harry Massalin. 1987. Superoptimizer – a Look at the Smallest Program. *Palo Alto, California* (1987).

Akimasa Morihata, Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Hideya Iwasaki. 2018. Optimizing Declarative Parallel Distributed Graph Processing by Using Constraint Solvers. In *Functional and Logic Programming*, John P. Gallagher and Martin Sulzmann (Eds.). Springer International Publishing, Cham, 166–181.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 522–538.

Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2012. Elixir: A system for synthesizing concurrent graph programs. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 375–394.

Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2015. Synthesizing parallel graph programs via automated planning. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 533–544.

Apan Qasem and Ken Kennedy. 2006. Profitable Loop Fusion and Tiling Using Model-driven Empirical Search. In *Proceedings of the 20th Annual International Conference on Supercomputing* (Cairns, Queensland, Australia) *(ICS '06)*. ACM, New York, NY, USA, 249–258. https://doi.org/10.1145/1183401.1183437

Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, Robert J Harrison, and Ponnuswamy Sadayappan. 2016a. A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 40.

Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J Harrison, and Ponnuswamy Sadayappan. 2016b. On fusing recursive traversals of Kd trees. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 152–162.

Ricardo Rocha and John Launchbury. 2011. *Practical Aspects of Declarative Languages: 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*. Vol. 6539. Springer.

Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, 1–10.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.

Olli Saarikivi, Margus Veanes, Todd Mytkowicz, and Madan Musuvathi. 2017. Fusing Effectful Comprehensions. *SIGPLAN Not.* 52, 6 (June 2017), 17–32. https://doi.org/10.1145/3140587.3062362

Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. Treefuser: a framework for analyzing and fusing general recursive tree traversals. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 76.

Laith Sakka, Kirshanthan Sundararajah, Ryan R Newton, and Milind Kulkarni. 2019. Sound, fine-grained traversal fusion for heterogeneous trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 830–844.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422* (2017).

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 305–316.

Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. 2016. Using domain-specific languages for analytic graph databases. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1257–1268.

G Shashidhar and Rupesh Nasre. 2016. Lighthouse: An automatic code generator for graph algorithms on gpus. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 235–249.

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290386

Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. *ACM SIGPLAN Notices* 51, 6 (2016), 326–340.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 281–294.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *ACM Sigplan Notices* 41, 11 (2006), 404–415.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2010. From program verification to program synthesis. In *ACM Sigplan Notices*, Vol. 45. ACM, 313–326.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. ACM, 7.

Martin Vechev and Eran Yahav. 2008. Deriving linearizable fine-grained concurrent objects. *ACM SIGPLAN Notices* 43, 6 (2008), 125–135.

Martin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-guided synthesis of synchronization. In *ACM Sigplan Notices*, Vol. 45. ACM, 327–338.

Keval Vora. 2019. Lumos: Dependency-Driven Disk-based Graph Processing. In *USENIX Annual Technical Conference (USENIX ATC '19)*. 429–442.

Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. (2017), 237–251. https://doi.org/10.1145/3037697.3037748

Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *Proceedings of the Second European Symposium on Programming* (Nancy, France). North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 231–248. http://dl.acm.org/citation.cfm?id=80098.80104

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 452–466.

Zhilei Xu, Shoaib Kamil, and Armando Solar-Lezama. 2014. MSL: A Synthesis Enabled Language for Distributed Implementations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New Orleans, Louisana) *(SC '14)*. IEEE Press, Piscataway, NJ, USA, 311–322. https://doi.org/10.1109/SC.2014.31

Yongzhe Zhang, Hsiang-Shang Ko, and Zhenjiang Hu. 2017. Palgol: A high-level DSL for vertex-centric graph processing with remote data access. In *Asian Symposium on Programming Languages and Systems*. Springer, 301–320.

Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276491

Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 301–316.

Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 375–386.