Data Analytics and Visualization Environment for xAPI and the Total Learning Architecture

DAVE Learning Analytics Algorithms

8 November 2019

This work was supported by the U.S. Advanced Distributed Learning (ADL) Initiative (HQ0034-18-C-0040). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ADL Initiative or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes.

Distribution Statement A

Approved for public release: distribution unlimited.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
08-11-2019	Research Report	February 2018 - November 2019

4. TITLE AND SUBTITLE
Data Analytics and Visualization Environment for xAPI and the Total Learning Architecture: DAVE Learning Analytics Algorithms

5a. CONTRACT NUMBER HQ0034-18-C-0040
5b. GRANT NUMBER N/A
5c. PROGRAM ELEMENT NUMBER 0603769D8Z

6. AUTHOR(S)
Blake-Plock, Shelly.

5d. PROJECT NUMBER N/A
5e. TASK NUMBER N/A
5f. WORK UNIT NUMBER N/A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Yet Analytics, Inc.
201 E Baltimore Street
Suite. 630
Baltimore, MD 21202

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
OUSD Personnel & Readiness
Advanced Distributed Learning Initiative
13501 Ingenuity Drive, Suite 248
Orlando, Florida 32826

10. SPONSOR/MONITOR'S ACRONYM(S) OUSD/P&R/FE&T/ADLI

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution A

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report introduces a language for defining the functionality of learning analytics algorithms in terms of Operations, Primitives and Algorithms which will be used to define Algorithms corresponding to an initial set of learning analytics questions. Additional questions may be added to this set in the future. This document will be updated to include additional Operations, Primitives, and Algorithms as they are defined by the Author(s) of this report or by members of the Open Source Community. The formal definitions in this document are optimized for understandability and conceptual presentation meaning they are not presented as, or intended to be, the most computationally efficient definition possible. The formal definitions are intended to serve as referential documentation of methodologies and programmatic strategies for handling the processing of xAPI data.

15. SUBJECT TERMS
Data Analytics, Data Algorithms, Primitives, Data Visualizations, Analytics, Algorithms, Visualizations, ADL, Advanced Distributed Learning, DAVE, Data Analytics and Visualization Environment

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
U	U	U	UU		Nick Armendariz
					19b. TELEPHONE NUMBER *(Include area code)* 407-381-5550

Data Analytics and Visualization Environment for xAPI and the Total Learning Architecture: DAVE Learning Analytics Algorithms

Yet Analytics, Inc.
201 E. Baltimore St., Ste. 630, Baltimore, MD 21202 USA

November 8, 2019

Introduction

This report introduces a language for defining the functionality of learning analytics algorithms in terms of **Operations**, **Primitives** and **Algorithms** which will be used to define Algorithms corresponding to an initial set of learning analytics questions. Additional questions may be added to this set in the future. This document will be updated to include additional Operations, Primitives, and Algorithms as they are defined by the Author(s) of this report or by members of the Open Source Community. Updates may also address refinement of existing definitions, thus this document is subject to continuous change but those which are significant will be documented within the DAVE change log. Any changes made to this report or to the DAVE github repository should follow the conventions established in the Contributing Wiki Page. The formal definitions in this document are optimized for understandability and conceptual presentation meaning they are not presented as, or intended to be, the most computationally efficient definition possible. The formal definitions are intended to serve as referential documentation of methodologies and programmatic strategies for handling the processing of xAPI data.

The structure of this document is as follows:

1. An Introduction to Z notation and its usage in this document
2. A formal specification for xAPI written in Z
3. Terminology: Operations, Primitives and Algorithms
4. What is an Operation
5. What is a Primitive
6. What is an Algorithm
7. Foundational Operations
8. Common Primitives
9. Example Algorithm
   (a) Init
   (b) Relevant?
   (c) Accept?
   (d) Step
   (e) Result

1 Z Notation Introduction

The following subsections provide a high level overview of select properties of Z Notation based on "The Z Notation: A Reference Manual" by J. M. Spivey. A copy of this reference manual can be found at dave/docs/z/Z-notation reference manual.pdf. In many cases, definitions will be pulled directly from the reference manual and when this occurs, the relevant page number(s) will be included. For a proper introduction with tutorial examples, see chapter 1, "Tutorial Introduction" from pages 1 to 23. For the $LaTeX$ symbols used to write Z, see the reference document found at dave/docs/z/zed-csp-documentation.pdf.

1.1 Decorations

The following decorations are used throughout this document and are taken directly from the reference manual. For a complete summary of the Syntax of Z, see chapter 6, Syntax Summary, starting on page 142.

$'$	[indicates final state of an operation]
?	[indicates input to an operation]
!	[indicates output of an operation]
Δ	[indicates the schema results in a change to the state space]
Ξ	[indicates the schema does not result in a change to the state space]
$\gg$	[indicates output of the left schema is input to the right schema]

1.2 Types

Objects have a type which characterizes them and distinguish them from other kinds of objects.

- Basic types are sets of objects which have no internal structure of interest meaning the concrete definition of the members is not relevant, only their shared type.
- Free types are used to describe (potentially nested and/or recursive) sets of objects. In the most simple case, a free type can be an enumeration of constants.

Within the xAPI Formal Specification, both of these types are used to describe the Inverse Functional Identifier property.

- Introduction of the basic types $MBOX$, $MBOX\_SHA1SUM$, $OPENID$ and $ACCOUNT$ allows the specification to talk about these constraints within the xAPI specification without defining their exact structure
- The free type IFI is defined as one of the above basic types meaning an object of type IFI is of type $MBOX$ or $MBOX\_SHA1SUM$ or $OPENID$ or $ACCOUNT$.

Types can be composed together to form composite types and thus complex objects.

$$[MBOX, MBOX\_SHA1SUM, OPENID, ACCOUNT]$$
$$IFI ::= MBOX \mid MBOX\_SHA1SUM \mid OPENID \mid ACCOUNT$$

Within the xAPI Formal Specification, IFI is used within the definition of an *agent* as presented in the schema *Agent*.

$$\begin{array}{|l}
\textit{Agent} \\
\hline
agent : AGENT \\
objectType : OBJECTTYPE \\
name : \mathbb{F}_1 \#1 \\
ifi : IFI \\
\hline
objectType = Agent \\
agent = \{ifi\} \cup \mathbb{P}\{name, objectType\} \\
\hline
\end{array}$$

See section 2.2, pages 28 to 34, and chapter 3, pages 42 to 85, for more information about Schemas and the Z Language.

1.3 Sets

A collection of elements that all share a type. A set is characterized solely by which objects are members and which are not. Both the order and repetition of objects are ignored. Sets are written in one of two ways:

- listing their elements
- by a property which is characteristic of the elements of the set.

such that the following law from page 55 holds for some object y

$$y \in \{x_1, ..., x_n\} \iff y = x_1 \vee ... \vee y = x_n$$

1.4 Ordered Pairs

Two objects (x, y) where x is paired with y. An n-tuple is the pairing of n objects together such that equality between two n-tuple pairs is given by the law from page 55

$$(x_1, ..., x_n) = (y_1, ..., y_n) \iff x_1 = y_1 \wedge ... \wedge x_n = y_n$$

When ordered pairs are used with respect to application (as seen on page 60)

$$fx \Rightarrow f(x) \iff (x, y) \in f$$

which states that $f(x)$ is defined if and only if there is a unique value y which result from fx Additionally, application associates to the left

$$fxy \Rightarrow (fx)y \Rightarrow (f(x), y)$$

meaning $f(x)$ results in a function which is then applied to y.

1.5 Sequences

A collection of elements where their ordering matters such that

$$\langle a_1, ..., a_n \rangle \Rightarrow \{1 \mapsto a_1, ..., n \mapsto a_n\}$$

as seen on page 115. Additionally, iseq is used to describe a sequence whose members are distinct.

1.6 Bags

A collection of elements where the number of times an element appears in the collection is meaningful.

$$[\![a_1, ..., a_n]\!] \Rightarrow \{a_1 \mapsto k_1, ..., k_n \mapsto k_n\}$$

As described on page 124, each element a_i appears k_i times in the list $a_1, ..., a_n$ such that the number of occurrences of a_i within bag A is returned by

$$count\, A\, a_i \;\equiv A \,\#\, a_i$$

1.7 Maps

This document introduces a named subcategory of sets, *map* of the free type KV, which are akin to sequences and bags. To enumerate the members of a *map*, $\langle\!\langle ... \rangle\!\rangle$ is used but should not be confused with $d_i \langle\!\langle E_i[T] \rangle\!\rangle$ within a Free Type definition. The distinction between the two usages is context dependent but in general, if $\langle\!\langle ... \rangle\!\rangle$ is used outside of a constructor declaration within a Free Type definition, it should be assumed to represent a *map*.

$$KV ::= base \mid associate \langle\!\langle KV \times X \times Y \rangle\!\rangle$$

where

base	[is a constant which is the empty $KV \Rightarrow \langle\!\langle\rangle\!\rangle$]
associate	[is a constructor and is inferred to be an injection]

The full enumeration of all properties, constraints and functions specific to a *map* with type KV will be defined elsewhere but *associate* can be understood to (in the most basic case) operate as follows.

$$associate(base, x_i, y_i) = \langle\!\langle (x_i, y_i) \rangle\!\rangle \Rightarrow \langle\!\langle x_i \mapsto y_i \rangle\!\rangle$$

The enumeration of a *map* was chosen to be $\langle\!\langle ... \rangle\!\rangle$ as a *map* is a collection of injections such that if M is the result of $associate(base, x_i, y_i)$ from above then

$$atKey(M, x_i) = y_i \iff x_i \mapsto y_i \wedge (x_i, y_i) \in M$$

1.8 Select Operations and Symbols

The follow are defined in Chapter 4 (The Mathematical Tool-kit) within the reference manual and are used extensively throughout this document. In many cases, the functions listed here will serve as Operations in the context of Primitives and Algorithms.

1.8.1 Functions

$\rightarrowtail\!\!\!\!\!+$ $\nrightarrow$	[relate each x ∈ X to at most one y ∈ Y, page 105]
$\rightarrow$	[relate each x ∈ X to exactly one y ∈ Y, page 105]
$\rightarrowtail\!\!\!\!\!\nrightarrow$	[map different elements of x to different y, page 105]
$\rightarrowtail$	[$\rightarrowtail\!\!\!\!\!\nrightarrow$ that are also $\rightarrow$, page 105]
$\twoheadrightarrow\!\!\!\!\!+$	[$X \nrightarrow Y$ where whole of Y is the range, page 105]
$\twoheadrightarrow$	[$X \nrightarrow Y$ whole of X as domain and whole of Y as range, page 105]
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	[map x ∈ X one-to-one with y ∈ Y, page 105]

$$
\begin{aligned}
&X \nrightarrow Y == \{\, f : X \leftrightarrow Y \mid (\forall x : X; y1, y2 : Y \bullet \\
&\qquad\qquad (x \mapsto y_1 \in f \;\wedge\; (x \mapsto y_2) \in f \Rightarrow y_1 = y_2))\} \\
&X \rightarrow Y == \{\, f : X \nrightarrow Y \mid \operatorname{dom} f = X\} \\
&X \rightarrowtail\!\!\!\!\!\nrightarrow Y == \{\, f : X \nrightarrow Y \mid (\forall x_1, x_2 : \operatorname{dom} f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2)\} \\
&X \rightarrowtail Y == (X \rightarrowtail\!\!\!\!\!\nrightarrow Y) \cap (X \rightarrow Y) \\
&X \twoheadrightarrow\!\!\!\!\!+\, Y == \{\, f : X \rightarrowtail\!\!\!\!\!\nrightarrow Y \mid \operatorname{ran} f = Y\} \\
&X \twoheadrightarrow Y == (X \twoheadrightarrow\!\!\!\!\!+\, Y) \cap (X \rightarrow Y) \\
&X \rightarrowtail\!\!\!\!\twoheadrightarrow Y == (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)
\end{aligned}
$$

1.8.2 Ordered Pairs, Maplet and Composition of Relations

first	[returns the first element of an ordered pair, page 93]
second	[returns the second element of an ordered pair, page 93]
$\mapsto$	[maplet is a graphic way of expressing an ordered pair, page 95]
dom	[set of all x ∈ X related to at least one y ∈ Y by R, page 96]
ran	[set of all y ∈ Y related to at least one x ∈ X by R, page 96]
$\fatsemi$	[The composition of two relationships, page 97]
$\circ$	[The backward composition of two relationships, page 97]

$$
\begin{array}{|l}
[X, Y] \\
\hline
first : X \times Y \rightarrow X \\
second : X \times Y \rightarrow Y \\
\hline
\forall x : X; y : Y \bullet \\
\qquad first(x, y) = x \wedge \\
\qquad second(x, y) = y \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X,Y] \\
\_ \mapsto \_ : X \times Y \to X \times Y \\
\hline
\forall x : X; y : Y \bullet \\
x \mapsto y = (x,y) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X,Y] \\
\mathrm{dom} : (X \leftrightarrow Y) \to \mathbb{P}\,X \\
\mathrm{ran} : (X \leftrightarrow Y) \to \mathbb{P}\,Y \\
\hline
\forall R : X \leftrightarrow Y \bullet \\
\qquad \mathrm{dom}\,R = \{\, x : X;\ y : Y \mid x\underline{R}y \bullet x \} \land \\
\qquad \mathrm{ran}\,R = \{\, x : X;\ y : Y \mid x\underline{R}y \bullet y \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X,Y,Z] \\
\_ \mathbin{{}_{\circ}^{\circ}} \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \to (X \leftrightarrow Z) \\
\_ \circ \_ : (Y \leftrightarrow X) \times (X \leftrightarrow Y) \to (X \leftrightarrow X) \\
\hline
\forall\, Q : X \leftrightarrow Y; R : Y \leftrightarrow Z \bullet \\
\qquad Q \mathbin{{}_{\circ}^{\circ}} R = R \circ Q = \{\, x : X; y : Y; z : Z| \\
\qquad\qquad\qquad\qquad x\,\underline{Q}\,y \ \land\ y\,\underline{R}\,z \bullet x \mapsto z \} \\
\hline
\end{array}
$$

1.8.3 Numeric

succ	[the next natural number, page 109]
..	[set of integers within a range, page 109]
#	[number of members of a set, page 111]
min	[smallest number in a set of numbers, page 113]
max	[largest number in a set of numbers, page 113]

$$
\begin{array}{|l}
succ : \mathbb{N} \to \mathbb{N} \\
\_ \mathinner{..} \_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{P}\,\mathbb{Z} \\
\hline
\forall n : \mathbb{N} \bullet succ(n) = n + 1 \\
forall a, b : \mathbb{Z} \bullet \\
\qquad\qquad a \mathinner{..} b = \{\, k : \mathbb{Z} \mid a \leq k \leq b \}
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X] \\
\# : \mathbb{F}\,X \to \mathbb{N} \\
\hline
\forall S : \mathbb{F}\,X \bullet \\
\qquad \#S = (\mu\, n : \mathbb{N} \mid (\exists f : 1 \mathinner{..} n \rightarrowtail X \bullet \mathrm{ran}\, f = S)) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\min : \mathbb{P}_1\,\mathbb{Z} \nrightarrow \mathbb{Z} \\
\max : \mathbb{P}_1\,\mathbb{Z} \nrightarrow \mathbb{Z} \\
\hline
\min = \{\, S : \mathbb{P}_1\,\mathbb{Z}; m : \mathbb{Z} \mid \\
\qquad\qquad m \in S \wedge (\forall n : S \bullet m \leq n) \bullet S \mapsto m \} \\
\max = \{\, S : \mathbb{P}_1\,\mathbb{Z}; m : \mathbb{Z} \mid \\
\qquad\qquad m \in S \wedge (\forall n : S \bullet m \geq n) \bullet S \mapsto m \}
\end{array}
$$

1.8.4 Sequences

$\frown$	[concatenation of two sequences, page 116]
rev	[reverse a sequence, page 116]
$head$	[first element of a sequence, page 117]
$last$	[last element of a sequence, page 117]
$tail$	[all elements of a sequence except for the first, page 117]
$front$	[all elements of a sequence except for the last, page 118]
$\upharpoonleft$	[sub seq based on provided indices, order maintained, page 118]
$\upharpoonright$	[sub seq based on provided condition, order maintained, page 118]
$squash$	[compacts a fn of positive integers into a sequence, page 118]
$\frown/$	[flatten seq of seqs into single seq, page 121]
$\mathsf{disjoint}$	[pairs of sets in family have empty intersection, page 122]
$\mathsf{partition}$	[union of all pairs of sets = the family set, page 122]

$$
\begin{array}{|l}
[X] \\
\hline
\_ \frown \_ : \operatorname{seq} X \times \operatorname{seq} X \rightarrow \operatorname{seq} X \\
rev : \operatorname{seq} X \rightarrow \operatorname{seq} X \\
\hline
\forall s, t : \operatorname{seq} X \bullet \\
\qquad s \frown t = s \cup \{\, n : \operatorname{dom} t \bullet n + \# s \mapsto t(n) \} \\
\forall s : \operatorname{seq} X \bullet \\
\qquad rev s = (\lambda\, n : \operatorname{dom} s \bullet s(\# s - n + 1)) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
[X] \\
\hline
head, last : \operatorname{seq}_1 X \rightarrow X \\
tail, front : \operatorname{seq}_1 X \rightarrow \operatorname{seq} X \\
\hline
\forall s : \operatorname{seq}_1 X \bullet \\
\qquad head\, s = s(1) \wedge \\
\qquad last\, s = s(\# s) \wedge \\
\qquad tail\, s = (\lambda\, n : 1 \mathrel{..} \# s - 1 \bullet s(n+1)) \wedge \\
\qquad front\, s = (1 \mathrel{..} \# s - 1) \lhd s \\
\hline
\end{array}
$$

$$
\begin{array}{l}
[X] \\
\hline
\_ \upharpoonleft \_ : \mathbb{P}\,\mathbb{N}_1 \times \operatorname{seq} X \rightarrow \operatorname{seq} X \\
\_ \upharpoonright \_ : \operatorname{seq} X \times \mathbb{P}\,X \rightarrow \operatorname{seq} X \\
squash : (\mathbb{N}_1 \pfun X) \rightarrow \operatorname{seq} X \\
\hline
\forall U : \mathbb{P}\,\mathbb{N}_1; s : \operatorname{seq} X \bullet \\
\qquad U \upharpoonleft s = squash(U \lhd s) \\
\forall s : \operatorname{seq} X; V : \mathbb{P}\,X \bullet \\
\qquad s \upharpoonright V = squash(s \rhd V) \\
\forall f : \mathbb{N}_1 \pfun X \bullet \\
\qquad squash f = f \circ (\mu\, p : 1 \mathrel{..} \# f \rightarrowtail\!\!\!\!\rightarrow \operatorname{dom} f \mid p \circ succ \circ p^{\sim} \subseteq (\_ < \_))
\end{array}
$$

$$
\begin{array}{l}
[X] \\
\hline
{}^\frown/ : \operatorname{seq}(\operatorname{seq} X) \rightarrow \operatorname{seq} X \\
\hline
{}^\frown/\langle\rangle = \langle\rangle \\
\forall s : \operatorname{seq} X \bullet {}^\frown/\langle s \rangle = s \\
\forall q, r : \operatorname{seq}(\operatorname{seq} X) \bullet \\
\qquad {}^\frown/(q \frown r) = ({}^\frown/\,q) \frown ({}^\frown/\,r)
\end{array}
$$

$$
\begin{array}{l}
[I, X] \\
\hline
\mathsf{disjoint}\ \_ : \mathbb{P}(I \pfun \mathbb{P}\,X) \\
\_\ \mathsf{partition}\ \_ : (I \pfun \mathbb{P}\,X) \leftrightarrow \mathbb{P}\,X \\
\hline
\forall S : I \pfun \mathbb{P}\,X; T : \mathbb{P}\,X \bullet \\
\qquad (\mathsf{disjoint}\ S \iff \\
\qquad\qquad (\forall i, j : \operatorname{dom} S \mid i \neq j \bullet S(i) \cap S(j) = \emptyset)) \land \\
\qquad (S\ \mathsf{partition}\ T \iff \\
\qquad\qquad \mathsf{disjoint}\ S \land \bigcup\{\, i : \operatorname{dom} S \bullet S(i) \} = T)
\end{array}
$$

1.8.5 Bags

count, #	[the number of times something appears in a bag, page 124]
$\otimes$	[scaling across a bag, page 124]
$\uplus$	[union of two bags, sum of occurrences, page 126]
$\uminus$	[bag difference, subtract occurrences or zero if negative, page 126]
items	[conversion from seq to bag, page 127]

$[X]$

$count : \mathrm{bag}\, X \rightarrowtail\!\!\!\!\rightarrow (X \rightarrow \mathbb{N})$
$\_ \# \_ : \mathrm{bag}\, X \times X \rightarrow \mathbb{N}$
$\_ \otimes \_ : \mathbb{N} \times \mathrm{bag}\, X \rightarrow \mathrm{bag}\, X$

$\forall B : \mathrm{bag}\, X \bullet$
$\quad countB = (\lambda x : X \bullet 0) \oplus B$
$\forall x : X; B : \mathrm{bag}\, x \bullet$
$\quad B \# x = count\, B\, x$
$\forall n : \mathbb{N}; B : \mathrm{bag}\, X; x : X \bullet$
$\quad (n \otimes B) \# x = n * (B \# x)$

$[X]$

$\_ \uplus \_, \_ \mathbin{\cup\!\!\!\!-} \_ : \mathrm{bag}\, X \times \mathrm{bag}\, X \rightarrow \mathrm{bag}\, X$

$\forall B, C : \mathrm{bag}\, X; x : X \bullet$
$\quad (B \uplus C) \# x = B \# x + C \# x \wedge$
$\quad (B \mathbin{\cup\!\!\!\!-} C) \# x = \max\{B \# x - C \# x, 0\}$

$[X]$

$items : \mathrm{seq}\, X \rightarrow \mathrm{bag}\, X$

$\forall s : \mathrm{seq}\, X; x : X \bullet$
$\quad (items\, s) \# x = \#\{ i : \mathrm{dom}\, s \mid s(i) = x\}$

2 xAPI Formal Specification

The current formal specification only defines xAPI statements abstractly within the context of Z. A concrete definition for xAPI statements is outside the scope of this document.

2.1 Basic and Free Types

$[MBOX, MBOX\_SHA1SUM, OPENID, ACCOUNT]$

- Basic Types for the abstract representation of the different forms of Inverse Functional Identifiers found in xAPI

$[CHOICES, SCALE, SOURCE, TARGET, STEPS]$

- Basic Types for the abstract representation of the different forms of Interaction Components found in xAPI

$IFI ::= MBOX \mid MBOX\_SHA1SUM \mid OPENID \mid ACCOUNT$

- Free Type unique to Agents and Groups, The concrete definition of the listed Basic Types is outside the scope of this specification

$OBJECTTYPE ::= Agent \mid Group \mid SubStatement \mid StatementRef \mid Activity$

- A type which can be present in all activities as defined by the xAPI specification

$INTERACTIONTYPE ::= true{-}false \mid choice \mid fill{-}in \mid long{-}fill{-}in \mid matching \mid performance \mid sequencing \mid likert \mid numeric \mid other$

- A type which represents the possible interactionTypes as defined within the xAPI specification

$INTERACTIONCOMPONENT ::= CHOICES \mid SCALE \mid SOURCE \mid TARGET \mid STEPS$

- A type which represents the possible interaction components as defined within the xAPI specification

- the concrete definition of the listed Basic Types is outside the scope of this specification

$CONTEXTTYPES ::= parent \mid grouping \mid category \mid other$

- A type which represents the possible context types as defined within the xAPI specification

$[STATEMENT]$

- Basic type for an xAPI data point

$[AGENT, GROUP]$

- Basic types for Agents and collections of Agents

2.2 Id Schema

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Id} \\
id : \mathbb{F}_1 \#1 \\
\hline
\end{array}
$$

- the schema Id introduces the component id which is a non-empty, finite set of 1 value

2.3 Schemas for Agents, Groups and Actors

$$
\begin{array}{|l}
\multicolumn{1}{l}{Agent} \\
agent : AGENT \\
objectType : OBJECTTYPE \\
name : \mathbb{F}_1 \#1 \\
ifi : IFI \\
\hline
objectType = Agent \\
agent = \{ifi\} \cup \mathbb{P}\{name, objectType\} \\
\hline
\end{array}
$$

- The schema $Agent$ introduces the component $agent$ which is a set consisting of an ifi and optionally an $objectType$ and/or $name$

$$
\begin{array}{|l}
\multicolumn{1}{l}{Member} \\
Agent \\
member : \mathbb{F}_1 \\
\hline
member = \{a : AGENT \mid \forall a_n : a_i..a_j \bullet i \leq n \leq j \bullet a = agent\} \\
\hline
\end{array}
$$

- The schema $Member$ introduces the component $member$ which is a set of objects a, where for every a within $a_0..a_n$, a is an $agent$

$$
\begin{array}{|l}
\multicolumn{1}{l}{Group} \\
Member \\
group : GROUP \\
objectType : OBJECTTYPE \\
ifi : IFI \\
name : \mathbb{F}_1 \#1 \\
\hline
objectType = Group \\
group = \{objectType, name, member\} \vee \{objectType, member\} \vee \\
\qquad\qquad \{objectType, ifi\} \cup \mathbb{P}\{name, member\} \\
\hline
\end{array}
$$

- The schema $Group$ introduces the component $group$ which is of type $GROUP$ and is a set of either $objectType$ and $member$ with optionaly $name$ or $objectType$ and ifi with optionally $name$ and/or $member$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Actor\;}}\\
Agent\\
Group\\
actor : AGENT \lor GROUP\\
\hline
actor = agent \lor group\\
\hline
\end{array}
$$

- The schema *Actor* introduces the component *actor* which is either an *agent* or *group*

2.4 Verb Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Verb\;}}\\
Id\\
display, verb : \mathbb{F}_1\\
\hline
verb = \{id, display\} \lor \{id\}\\
\hline
\end{array}
$$

- The schema $Verb$ introduces the component *verb* which is a set that consists of either *id* and the non-empty, finite set *display* or just *id*

2.5 Object Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Extensions\;}}\\
extensions, extensionVal : \mathbb{F}_1\\
extensionId : \mathbb{F}_1 \; \#1\\
\hline
extensions = \{e : (extensionId, extensionVal) \;\; | \forall e_n : e_i..e_j \bullet i \leq n \leq j \bullet\\
\qquad\qquad (extensionId_i, extensionVal_i) \lor (extensionId_i, extensionVal_j) \land\\
\qquad\qquad (extensionId_j, extensionVal_i) \lor (extensionId_j, extensionVal_j) \land\\
\qquad\qquad extensionId_i \neq extensionId_j\}\\
\hline
\end{array}
$$

- The schema *Extensions* introduces the component *extensions* which is a non-empty, finite set that consists of ordered pairs of $extensionId$ and $extensionVal$. Different $extensionId$s can have the same $extensionVal$ but there can not be two identical $extensionId$ values
- $extensionId$ is a non-empty, finite set with one value
- $extensionVal$ is a non-empty, finite set

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;InteractionActivity\;}}\\
interactionType : INTERACTIONTYPE\\
correctResponsePattern : \mathrm{seq}_1\\
interactionComponent : INTERACTIONCOMPONENT\\
\hline
interactionActivity = \{interactionType, correctReponsePattern, interactionComponent\} \lor\\
\qquad\qquad \{interactionType, correctResponsePattern\}\\
\hline
\end{array}
$$

- The schema $InteractionActivity$ introduces the component $interactionActivity$ which is a set of either $interactionType$ and $correctResponsePattern$ or $interactionType$ and $correctResponsePattern$ and $interactionComponent$

$$
\begin{array}{|l}
\hline
\textit{Definition} \\
\hline
InteractionActivity \\
Extensions \\
definition, name, description : \mathbb{F}_1 \\
type, moreInfo : \mathbb{F}_1 \#1 \\
\hline
definition = \mathbb{P}_1\{name, description, type, moreInfo, extensions, interactionActivity\} \\
\hline
\end{array}
$$

- The schema $Definition$ introduces the component $definition$ which is the non-empty, finite power set of $name$, $description$, $type$, $moreInfo$ and $extensions$

$$
\begin{array}{|l}
\hline
\textit{Object} \\
\hline
Id \\
Definition \\
Agent \\
Group \\
Statement \\
objectTypeA, objectTypeS, objectTypeSub, objectType : OBJECTTYPE \\
substatement : STATEMENT \\
object : \mathbb{F}_1 \\
\hline
substatement = statement \\
objectTypeA = Activity \\
objectTypeS = StatementRef \\
objectTypeSub = SubStatement \\
objectType = objectTypeA \vee objectTypeS \\
object = \{id\} \vee \{id, objectType\} \vee \{id, objectTypeA, definition\} \\
\qquad \vee \{id, definition\} \vee \{agent\} \vee \{group\} \vee \{objectTypeSub, substatement\} \\
\qquad \vee \{id, objectTypeA\} \\
\hline
\end{array}
$$

- The schema $Object$ introduces the component $object$ which is a non-empty, finite set of either id, id and $objectType$, id and $objectTypeA$, id and $objectTypeA$ and $definition$, $agent$, $group$, or $substatement$

- The schema $Statement$ and the corresponding component $statement$ will be defined later on in this specification

2.6 Result Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{Score}}} \\
score : \mathbb{F}_1 \\
scaled, min, max, raw : \mathbb{Z} \\
\hline
scaled = \{n : \mathbb{Z} \mid -1.0 \leq n \leq 1.0\} \\
min = n < max \\
max = n > min \\
raw = \{n : \mathbb{Z} \mid min \leq n \leq max\} \\
score = \mathbb{P}_1\{scaled, raw, min, max\} \\
\hline
\end{array}
$$

- The schema *Score* introduces the component *score* which is the non-empty powerset of *min*, *max*, *raw* and *scaled*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{Result}}} \\
Score \\
Extensions \\
success, completion, response, duration : \mathbb{F}_1 \#1 \\
result : \mathbb{F}_1 \\
\hline
success = \{true\} \vee \{false\} \\
completion = \{true\} \vee \{false\} \\
result = \mathbb{P}_1\{score, success, completion, response, duration, extensions\} \\
\hline
\end{array}
$$

- The schema *Result* introduces the component *result* which is the non-empty power set of *score*, *success*, *completion*, *response*, *duration* and *extensions*

2.7 Context Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{Instructor}}} \\
Agent \\
Group \\
instructor : AGENT \vee GROUP \\
\hline
instructor = agent \vee group \\
\hline
\end{array}
$$

- The schema *Instructor* introduces the component *instructor* which can be either an *agent* or a *group*

$$
\begin{array}{|l}
\hline \textit{Team} \\
Group \\
team : GROUP \\
\hline
team = group \\
\hline
\end{array}
$$

- The schema $Team$ introduces the component $team$ which is a $group$

$$
\begin{array}{|l}
\hline \textit{Context} \\
Instructor \\
Team \\
Object \\
Extensions \\
registration, revision, platform, language : \mathbb{F}_1 \#1 \\
parentT, groupingT, categoryT, otherT : CONTEXTTYPES \\
contextActivities, statement : \mathbb{F}_1 \\
\hline
statement = object \setminus (id, objectType, agent, group, definition) \\
parentT = parent \\
groupingT = grouping \\
categoryT = category \\
otherT = other \\
contextActivity = \{ca : object \setminus (agent, group, objectType, objectTypeSub, substatement)\} \\
contextActivityParent = (parentT, contextActivity) \\
contextActivityCategory = (categoryT, contextActivity) \\
contextActivityGrouping = (groupingT, contextActivity) \\
contextActivityOther = (otherT, contextActivity) \\
contextActivities = \mathbb{P}_1\{contextActivityParent, contextActivityCategory, \\
\qquad contextActivityGrouping, contextActivityOther\} \\
context = \mathbb{P}_1\{registration, instructor, team, contextActivities, revision, \\
\qquad platform, language, statement, extensions\} \\
\hline
\end{array}
$$

- The schema $Context$ introduces the component $context$ which is the non-empty powerset of $registration$, $instructor$, $team$, $contextActivities$, $revision$, $platform$, $language$, $statement$ and $extensions$
- The notation $object \setminus agent$ represents the component $object$ except for its subcomponent $agent$

2.8 Timestamp and Stored Schema

$$
\begin{array}{|l}
\hline \textit{Timestamp} \\
timestamp : \mathbb{F}_1 \#1 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline \textit{Stored} \\
stored : \mathbb{F}_1 \#1 \\
\hline
\end{array}
$$

- The schema $Timestamp$ and $stored$ introduce the components $timestamp$ and $stored$ respectively. Each are non-empty, finite sets containing one value

2.9 Attachments Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Attachments\ }}\\
display, description, attachment, attachments : \mathbb{F}_1\\
usageType, sha2, fileUrl, contentType : \mathbb{F}_1\ \#1\\
length : \mathbb{N}\\
\hline
attachment = \{usageType, display, contentType, length, sha2\} \cup \mathbb{P}\{description, fileUrl\}\\
attachments = \{a : attachment\}\\
\hline
\end{array}
$$

- The schema $Attachments$ introduces the component $attachments$ which is a non-empty, finite set of the component $attachment$

- The component $attachment$ is a non-empty, finite set of the components $usageType$, $display$, $contentType$, $length$, $sha2$ with optionally $description$ and/or $fileUrl$

2.10 Statement and Statements Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Statement\ }}\\
Id\\
Actor\\
Verb\\
Object\\
Result\\
Context\\
Timestamp\\
Stored\\
Attachments\\
statement : STATEMENT\\
\hline
statement = \{actor, verb, object, stored\} \cup\\
\qquad\qquad \mathbb{P}\{\mathrm{id}, result, context, timestamp, attachments\}\\
\hline
\end{array}
$$

- The schema $Statement$ introduces the component $statement$ which consists of the components $actor$, $verb$, $object$ and $stored$ and the optional components id, $result$, $context$, $timestamp$, and/or $attachments$

- The schema $Statement$ allows for subcomponent of $statement$ to referenced via the . (selection) operator

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Statements}} \\
\hline
Statement \\
IsoToUnix \\
statements : \mathbb{F}_1 \\
\hline
statements = \{s : statement \mid \forall s_n : s_i..s_j \bullet i \leq n \leq j \\
\qquad \bullet\ convert(s_i.timestamp) \leq convert(s_j.timestamp)\} \\
\hline
\end{array}
$$

- The schema *Statements* introduces the component *statements* which is a non-empty, finite set of the component *statement* which are in chronological order.

3 Operations, Primitives and Algorithms

The following sections introduce, define and explain Operations, Primitives and Algorithms generally using the Terminology presented below. Operations are the building blocks of Primitives whereas Primitives are the building blocks of Algorithms. The definitions which follow are flexible enough to support implementation across programing languages but have been inspired by the core concepts found within Lisp and Z. The focus of these sections is to define the properties of and interactions between Operations, Primitives and Algorithms in a general way which doesn't place unnecessary bounds on their range of possible functionality with respect to processing xAPI data.

3.1 Terminology

Within this document, (s) indicates one or more.

3.1.1 Scalar

When working with xAPI data, Statements are written using JavaScript Object Notation (JSON). This data model supports a few fundamental types as described by JSON Schema. In order to speak about a singular valid JSON value (string, number, boolean, null) generically, the term Scalar is used. To talk about a scalar within a Z Schema, the following free and basic types are introduced.

$$[STRING, NULL]$$
$$Boolean ::= true \mid false$$
$$Scalar ::= Boolean \mid STRING \mid NULL \mid \mathbb{Z}$$

Arrays and Objects are also valid JSON values but will be referenced using the terms Collection and Map $\vee$ KV respectively.

3.1.2 Collection

a sequence $\langle ... \rangle$ of items c such that each $c : \mathbb{N} \times V \Rightarrow (\mathbb{N}, V) \Rightarrow \mathbb{N} \mapsto V$

$$C : Collection$$
$$C = \langle c_i..c_n..c_j \rangle \Rightarrow \{ i \mapsto c_i, n \mapsto c_n, j \mapsto c_j \} \bullet i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j$$

And the following free type is introduced for collections

$$Collection ::= emptyColl \mid append \langle\langle Collection \times Scalar \vee Collection \vee KV \times \mathbb{N} \rangle\rangle$$

$emptyColl$ [the empty Collection $\langle\rangle$]

$append$ [is a constructor and is inferred to be an injection]

KV [a free type introduced bellow]

$append(emptyColl, c?, 0) = \langle c_0 \rangle \Rightarrow \{ 0 \mapsto c? \}$ [$append$ adds $c?$ to $\langle\rangle$ at $\mathbb{N}$]

3.1.3 Key

An identifier k paired with some value v to create an ordered pair (k, v). k can take on any valid JSON value (Scalar, Collection, KV) except for the Scalar null. The following free type is introduced for keys.

$$K ::= (Scalar \setminus NULL) \,|\, Collection \,|\, KV$$

3.1.4 Value

A value v is paired with an identifier k to create an ordered pair (k, v). v can be any valid JSON value (Scalar, Collection, KV) The following free type is introduced for values.

$$V ::= Scalar \,|\, Collection \,|\, KV$$

3.1.5 Map

Within the Z Notation Introduction section, Maps are introduced using the free type KV.

$$KV ::= base \,|\, associate\langle\!\langle KV \times X \times Y \rangle\!\rangle$$

This definition is more accurately

$$KV ::= base \,|\, associate\langle\!\langle KV \times K \times V \rangle\!\rangle$$

which indicates the usage of Key k and Value v within *associate*. Using this updated definition,

$$associate(base, k, v) = \langle\!\langle (k, v) \rangle\!\rangle$$

such that a Map is a Collection of ordered pairs (k_n, v_n) and thus a Collection of mappings

$$(k_n, v_n) \Rightarrow k_n \mapsto v_n$$

but Maps are special cases of Collections as k_n is the unique identifier of v_n within a Map but the opposite is not true. In fact, keys are their own identifiers

$$\mathrm{id}\, v_n = k_n$$
$$\mathrm{id}\, k_n \neq v_n$$
$$\mathrm{id}\, k_n = k_n$$

Given a Map $M = \langle\!\langle (k_i, v_i) \mathinner{..} (k_n, v_n) \mathinner{..} (k_j, v_j) \rangle\!\rangle$ the following demonstrates the uniqueness of Keys but the same is not true for all v within M

$$k_i \neq k_n \neq k_j$$
$$v_i = v_n \lor v_i \neq v_n \;\; v_i = v_j \lor v_i \neq v_j \;\; v_j = v_n \lor v_j \neq v_n$$

which can all be stated formally as

$$
\begin{array}{l}
[K, V] \\
\hline
Map : K \times V \rightarrowtail\!\!\!\twoheadrightarrow KV \\
\hline
Map = \langle\!\langle (k_i, v_i) \mathinner{..} (k_n, v_n) \mathinner{..} (k_j, v_j) \rangle\!\rangle \bullet \\
\quad \mathrm{dom}\, Map = \{ k_i \mathinner{..} k_n \mathinner{..} k_j \} \\
\quad \mathrm{ran}\, Map = \{ v_i \mathinner{..} v_n \mathinner{..} v_j \} \\
\quad first(k_i, v_i) \neq first(k_n, v_n) \neq first(k_j, v_j) \land \\
\quad v_i = v_n \lor v_i \neq v_n \;\; v_i = v_j \lor v_i \neq v_j \;\; v_j = v_n \lor v_j \neq v_n \land \\
\quad \mathrm{id}\; v_i = k_i \land \mathrm{id}\; v_n = k_n \land \mathrm{id}\; v_j = k_j \land \\
\quad \mathrm{id}\; k_i = k_i \land \mathrm{id}\; k_n = k_n \land \mathrm{id}\, k_j = k_j
\end{array}
$$

Given that v can be a Map M, or a Collection C, Arbitrary nesting is allowed within Maps but the properties of a Map hold at any depth.

$$M = \langle\!\langle (k_i, v_i) \mathinner{..} (k_n, \langle\!\langle (k_{ni}, v_{ni}) \rangle\!\rangle) \mathinner{..} (k_j, \langle v_{ji} \mathinner{..} \langle\!\langle (k_{jn}, v_{jn}) \rangle\!\rangle \mathinner{..} \langle v_{jji} \mathinner{..} v_{jjn} \mathinner{..} v_{jjj} \rangle \rangle) \rangle\!\rangle$$

such that $\langle\!\langle (k_{ni}, v_{ni}) \rangle\!\rangle$ and $\langle\!\langle (k_{nj}, v_{nj}) \rangle\!\rangle$ are both Maps and adhere to the constraints enumerated above.

3.1.6 Statement

Immutable Map conforming to the xAPI Specification as described in the xAPI Formal Definition section of this document. The immutability of a Statement s is demonstrated by the following which indicates that s was not altered when passed to *associate*.

$$
\begin{array}{l}
s!, s? : STATEMENT \\
k? : K \\
v? : V \\
\hline
s! = associate(s?, k?, v?) = s? \Rightarrow (k?, v?) \notin s! \Rightarrow s! = s?
\end{array}
$$

Additionally, given the schema *Statements* the following is true for all *Statement*(s)

$$
\begin{array}{l}
Statements \\
Keys : STRING \\
S : Collection \\
\hline
Keys = \{ id, actor, verb, object, result, context, attachments, timestamp, stored \} \\
\mathrm{dom}\, statement = K \lhd Keys \\
S = \langle statement_i \mathinner{..} statement_n \mathinner{..} statement_j \rangle \bullet \\
\quad atKey(statement_i, id) \neq atKey(statement_n, id) \neq atKey(statement_j, id) \Rightarrow \\
\quad id_i \neq id_n \neq id_j \iff statement_i \neq statement_n \neq statement_j
\end{array}
$$

Which confirms the constraints found in the schema *Statement* and adds an additional constraint to *Statements* such that every unique *Statement* in a *Collection* of *Statements* has a unique *id*.

3.1.7 Algorithm State

Mutable Map *state* without any domain restriction such that

$$
\begin{array}{|l}
state?\,, state! : KV \\
k? : K \\
v? : V \\
\hline
associate(state?\,, k?\,, v?\,) = state! \bullet (k, v) \in state! \Rightarrow state? \neq state!
\end{array}
$$

3.1.8 Option

Mutable Map *opt* which is used to alter the result of an Algorithm. The effect of *opt* on an Algorithm will be discussed in the Algorithm Result section bellow.

4 Operation

An Operation is a function of arbitrary arguments and is defined using Z. For example, Operations pulled directly from "The Z Notation: A Reference Manual" include

- $first$
- $second$
- $succ$
- $\min$
- $\max$
- $count \equiv \#$
- $\frown$
- rev
- $head$
- $last$
- $tail$
- $front$
- $\upharpoonleft$
- $\upharpoonright$
- $\frown/$
- $\mathsf{disjoint}$
- $\mathsf{partition}$
- $\otimes$
- $\uplus$
- $\uminus$
- $items$

4.1 Domain

The arguments passed to an Operation can be any of the following but the definition of an Operation may limit the domain to a subset of the following

- Key(s)
- Value(s)
- Set(s)
- Collection(s)
- Bag(s)
- KV(s)
- Statement(s)
- Algorithm State

4.2 Range

The result of an Operation can be any of the following but the definition of an Operation may limit this range to a subset of the following

- Key(s)
- Value(s)
- Set(s)
- Collection(s)
- Bag(s)
- KV(s)
- Statement(s)
- Algorithm State

5 Primitive

Primitives break the processing of xAPI data down into discrete units that can be composed to create new analytical functions. Primitives allow users to address the methodology of answering research questions as a sequence of generic algorithmic steps which establish the necessary data transformations, aggregations and calculations required to reach the solution in an implementation agnostic way.

Within this document, they will be defined as a Collection of Operations and/or Primitives where the output is piped from member to member. In this section, o_n and p_n can be used as to describe Primitive members but for simplicity, only o_n will be used.

$$p_{\langle i \,..\, n \,..\, j\rangle} = o_i \gg o_n \gg o_j$$

Within any given Primitive p, variables local to p and any global variables may be passed as arguments to any member of p and there is no restriction on the ordering of arguments with respect to the piping. In the following, $q?$ is a global variable whereas the rest are local.

$$\begin{array}{|l}
x?\,, y?\,, z?\,, i!\,, n!\,, j!\,, p! : Value \\
o_i : Value \nrightarrow Value \\
o_n : Value \times Value \nrightarrow Value \\
o_j, p : Value \times Value \times Value \nrightarrow Value \\
\hline
i! = o_i(x?\,) \\
n! = o_n(i!\,, y?\,) \\
j! = o_j(z?\,, n!\,, q?\,) \\
p! = j! \Rightarrow o_j(z?\,, o_n(o_i(x?\,), y?\,), q?\,)
\end{array}$$

In the rest of this document, the following notation is used to distinguish between the functionality of a Primitive and its composition. This notation should be used when defining Primitives.

$$\begin{array}{|l}
primitiveName_ : _ \nrightarrow _ \\
\hline
primitiveName = \langle primitiveName_i\,_ \,..\, primitiveName_n\,_ \,..\, primitiveName_j\,_\rangle
\end{array}$$

- The top line indicates the Primitive
  - should be written using postfix notation within other schemas
  - is at least a partial function from some input to some output
- The bottom line is an enumeration of the composing Operations and/or Primitives and their order of execution

This means the definition of p from above can be updated as follows.

$$
\begin{array}{|l}
p\_ : Value \times Value \times Value \rightarrow\!\!\!\!\!\!+ \; Value \\
\hline
p = \langle\, o_i,\, o_n,\, o_j \rangle \\
p(x?\,,y?\,,z?\,) = o_j(z?\,,o_n(o_i(x?\,),y?\,),q?\,)
\end{array}
$$

Additionally, this notation supports declaration of recursive iteration via the presence of $recur\_$ within a Primitive chain

$$
\begin{array}{|l}
primitiveName_i = \langle\langle primitiveName_{ii}\_\,, primitiveName_{in}\_\rangle,\, recur\_\rangle^{\#\_} \\
\hline
\langle\langle primitiveName_{ii}\_\,, primitiveName_{in}\_\rangle,\, recur\_\rangle^{\#\_} \Rightarrow \\
\qquad (primtiveName_{ii} \gg primitiveName_{in})^{\#\_} \bullet \\
\qquad\quad \forall\, n : i \,..\, j \bullet j = \#\_ \wedge i \leq n \leq j \mid \exists_1 p_n : \_ \rightarrow\!\!\!\!\!+ \_\_ \rightarrow\!\!\!\!\!+ \_ \bullet \\
\qquad\qquad let \quad p_i == primtiveName_{ii} \gg primitiveName_{in} \Rightarrow \\
\qquad\qquad\qquad p_i\_ = primitiveName_{in}(primitiveName_{ii}\_) \\
\qquad\qquad\quad p_n == p_i \gg primtiveName_{ii} \gg primitiveName_{in} \Rightarrow \\
\qquad\qquad\qquad p_n\_ = primitiveName_{in}(primitiveName_{ii}(p_i\_)) \\
\qquad\qquad\quad p_j == p_n \gg primtiveName_{ii} \gg primitiveName_{in} \Rightarrow \\
\qquad\qquad\qquad p_j\_ = primitiveName_{in}(primitiveName_{ii}(p_n\_)) \\
\qquad\quad p_j = (primtiveName_{ii} \gg primitiveName_{in})^{\#\_} \bullet j = 3 \Rightarrow \\
\qquad\qquad\quad (primtiveName_{ii} \gg primitiveName_{in}) \gg \\
\qquad\qquad\quad (primtiveName_{ii} \gg primitiveName_{in}) \gg \\
\qquad\qquad\quad (primtiveName_{ii} \gg primitiveName_{in}) \Rightarrow \\
\qquad\qquad\qquad primitiveName_{in}(\\
\qquad\qquad\qquad\quad primitiveName_{ii}(\\
\qquad\qquad\qquad\qquad primitiveName_{in}(\\
\qquad\qquad\qquad\qquad\quad primitiveName_{ii}(p_i\_))))
\end{array}
$$

Here, p_i was chosen to only be two primitives $primtiveName_{ii} \wedge primitiveName_{in}$ for simplicity sake. The Primitive chain can be of arbitrary length. The number of iterations is described using the count operation $\#\_$. Above $j = 3$ was used to demonstrate the piping between iterations but j is not exclusively $= 3$. Given above, the term Primitive Chain can be defined as:

$$
\begin{array}{l}
(primtiveName_i \gg primitiveName_n \gg primitiveName_j)^{\#\_} \bullet \\
\qquad \#\_ = 0 \Rightarrow primtiveName_i \gg primitiveName_n \gg primitiveName_j
\end{array}
$$

where a Primitive chain iterated to the 0 is just the chain itself hence recursion is not a requirement of, but is supported within, the definition of Primitives.

5.1 Domain

Any of the following dependent upon the Operations which compose the Primitive

- Key(s)
- Value(s)

- Set(s)
- Collection(s)
- Bag(s)
- KV(s)
- Statement(s)
- Algorithm State

5.2 Range

Any of the following dependent upon the Domain and Functionality of the Primitive

- Key(s)
- Value(s)
- Set(s)
- Collection(s)
- Bag(s)
- KV(s)
- Statement(s)
- Algorithm State

6 Algorithm

Given a Collection of statement(s) $S_{\langle a..b..c\rangle}$ and potentially option(s) *opt* and potentially an existing Algorithm State *state* an Algorithm A executes as follows

1. call *init*
2. for each $stmt \in S_{\langle a..b..c\rangle}$
   (a) *relevant*?
   (b) *accept*?
   (c) *step*
3. return *result*

with each process within A is enumerated as

```
(init [state] body)
 - init state

(relevant? [state statement] body)
 - is the statement valid for use in algorithm?

(accept? [state statement] body)
 - can the algorithm consider the current statement?

(step [state statement] body)
 - processing per statement
 - can result in a modified state

(result [state] body)
 - return without option(s) provided
 - possibly sets default option(s)

(result [state opt] body)
 - return with consideration to option(s)
```

- *body* is a collection of Primitive(s) which establishes the processing of inputs $\rightarrow$ outputs
- *state* is a mutable Map of type KV and synonymous with Algorithm State
- *statement* is a single statement within the collection of statements passed as input data to the Algorithm A
- *opt* are additional arguments passed to the algorithm A which impact the return value of the algorithm and synonymous with Option

An Algorithm must be passed an Algorithm State and a Collection of Statement(s). Option is optional.

- Statement(s)
- Algorithm State
- Option(s)

An Algorithm will return an Algorithm State.

- Algorithm State

An Algorithm can be described via its components. A formal definition for an Algorithm is presented at the end of this section. The following subsections go into more detail about the components of an Algorithm.

$$Algorithm ::= Init \gg Relevant? \gg Accept? \gg Step \gg Result$$

6.1 Initialization

First process to run within an Algorithm which returns the Algorithm State for the current iteration.

$$\begin{array}{|l}
\hline
Init[KV] \\
state?, state! : KV \\
init_ : KV \twoheadrightarrow KV \\
\hline
init = \langle body \rangle \\
state! = init(state?) \bullet state! = state? \lor state! \neq state? \\
\hline
\end{array}$$

such that some *state*! does not need to be related to its arguments *state*? but *state*! could be derived from some seed *state*?. This functionality is dependent upon the composition of *body* within *init*.

6.1.1 Domain

- Algorithm State

6.1.2 Range

- Algorithm State

6.2 Relevant?

First process that each *stmt* passes through $\Rightarrow relevant? \prec accept? \prec step$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Relevant?\,[KV, STATEMENT] \qquad\qquad\qquad\qquad}} \\
state? : KV \\
stmt? : STATEMENT \\
relevant?\ \_ : KV \times STATEMENT \rightarrow Boolean \\
\hline
relevant? = \langle body \rangle \\
relevant?\,(state?\,, stmt?\,) = true \vee false \\
\hline
\end{array}
$$

resulting in an indication of whether the *stmt* is valid within algorithm A. The criteria which determines validity of *stmt* within A is defined by the *body* of $relevant?$

6.2.1 Domain

- Statement
- Algorithm State

6.2.2 Range

- Boolean

6.3 Accept?

Second process that each *stmt* passes through $\Rightarrow relevant? \prec accept? \prec step$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Accept?\,[KV, STATEMENT] \qquad\qquad\qquad\qquad}} \\
state? : KV \\
stmt? : STATEMENT \\
accept?\ \_ : KV \times STATEMENT \rightarrow Boolean \\
\hline
accept? = \langle body \rangle \\
accept?\,(state?\,, stmt?\,) = true \vee false \\
\hline
\end{array}
$$

resulting in an indication of whether the *stmt* can be sent to *step* given the current *state*. The criteria which determines usability of *stmt* given *state* is defined by the *body* of *accept*?

6.3.1 Domain

- Statement
- Algorithm State

6.3.2 Range

- Scalar

6.4 Step

An Algorithm Step consists of a sequential composition of Primitive(s) where the output of some function is passed as an argument to the next function both within and across Primitives in *body*.

$$body = p_i \gg p_n \gg p_j \Rightarrow o_{ii} \gg o_{in} \gg o_{ij} \gg o_{ni} \gg o_{nn} \gg o_{nj} \gg o_{ji} \gg o_{jn} \gg o_{jj}$$

The selection and ordering of Operation(s) and Primitive(s) into an Algorithmic Step determines how the Algorithm State changes during iteration through Statement(s) passed as input to the Algorithm.

$$\begin{array}{|l}
P = \langle p_i \mathbin{..} p_n \mathbin{..} p_j \rangle \bullet i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j \bullet p_i \gg p_n \gg p_j \\
P' = \langle p_{i'} \mathbin{..} p_{n'} \mathbin{..} p_{j'} \rangle \bullet i' \leq n' \leq j' \Rightarrow i' \prec n' \prec j' \iff i' \neq n' \neq j' \bullet p_{i'} \gg p_{n'} \gg p_{j'} \\
P'' = \langle p_x \mathbin{..} p_y \mathbin{..} p_z \rangle \bullet x \leq y \leq z \Rightarrow x \prec y \prec z \iff x \neq y \neq z \bullet p_x \gg p_y \gg p_z \\
\hline
P = P' \iff i \mapsto i' \wedge n \mapsto n' \wedge j \mapsto j' \\
P = P'' \iff (i \mapsto x \wedge n \mapsto y \wedge j \mapsto z) \wedge (p_i \equiv p_x \wedge p_n \equiv p_y \wedge p_j \equiv p_z)
\end{array}$$

step may or may not update the input Algorithm State given the current Statement from the Collection of Statement(s).

$$\begin{array}{|l}
S : Collection \\
stmt_a, stmt_b, stmt_c : STATEMENT \\
state?, step_a!, step_b!, step_c! : KV \\
step_ : KV \times STATEMENT \twoheadrightarrow KV \\
\hline
S = \langle stmt_a .. stmt_b .. stmt_c \rangle \bullet a \leq b \leq c \Rightarrow a \prec b \prec c \iff a \neq b \neq c \\
step_a! = step(state?, stmt_a) \bullet step_a! = state? \vee step_a! \neq state? \\
step_b! = step(step_a!, stmt_b) \bullet step_b! = step_a! \vee step_b! \neq step_a! \\
step_c! = step(step_b!, stmt_c) \bullet step_c! = step_b! \vee step_c! \neq step_b!
\end{array}$$

In general, this allows *step* to be defined as

$$\begin{array}{|l}
\hline
Step[KV, STATEMENT] \\
state?, state! : KV \\
stmt? : STATEMENT \\
step_ : KV \times STATEMENT \twoheadrightarrow KV \\
\hline
step = \langle body \rangle \\
state! = step\,(state?, stmt?) = state? \vee state! \neq state? \\
\hline
\end{array}$$

A change of $state? \rightarrow state! \bullet state! \neq state?$ can be predicted to occur given

- The definition of individual Operations which constitute a Primitive
- The ordering of Operations within a Primitive
- The Primitive(s) chosen for inclusion within the body of *step*
- The ordering of Primitive(s) within the body of *step*
- The key value pair(s) in both Algorithm State and the current Statement
- The ordering of Statement(s)

6.4.1 Domain

- Statement
- Algorithm State

6.4.2 Range

- Algorithm State

6.5 Result

Last process to run within an Algorithm which returns the Algorithm State *state* when all $s \in S$ have been processed by *step*

$$\begin{aligned}&relevant? \prec accept? \prec step \prec result \prec relevant? \iff S \neq \emptyset\\&relevant? \prec accept? \prec step \prec result \iff S = \emptyset\end{aligned}$$

and does so without preventing subsequent calls of A

$$\begin{array}{|l}\hline Result[KV, KV]\\ result!, state?, opt? : KV\\ result\_ : KV \times KV \twoheadrightarrow KV\\ \hline result = \langle body \rangle\\ result! = result(state?, opt?) = state? \lor state! \neq state?\\ \hline\end{array}$$

such that if at some future point j within the timeline $i \mathinner{..} n \mathinner{..} j$

$$\begin{array}{lr}S(t_n) = \emptyset & \text{[S is empty at } t_n\text{]}\\ S(t_j) \neq \emptyset & \text{[S is not empty at } t_j\text{]}\\ S(t_{n-i}) & \text{[stmts(s) added to } S \text{ between } t_i \text{ and } t_n\text{]}\\ S(t_{j-n}) & \text{[stmts(s) added to } S \text{ between } t_n \text{ and } t_j\text{]}\\ S(t_{j-i}) = S(t_{n-i}) \ \cup \ S(t_{j-n}) & \text{[stmts(s) added to } S \text{ between } t_i \text{ and } t_j\text{]}\end{array}$$

Algorithm A can pick up from a previous $state_n$ without losing track of its own history.

$$\begin{array}{|l}state_{n-i} = A(state_i,\ S(t_{n-i}))\\ state_{n-1} = A(state_{n-2},\ S(t_{n-1}))\\ state_n = A(state_{n-1},\ S(t_n))\\ state_{j-n} = A(state_n,\ S(t_{j-n}))\\ state_j = A(state_i,\ S(t_{j-i}))\\ \hline state_n = state_{n-1} \iff S(t_n) = \emptyset \land S(t_{n-1}) \neq \emptyset\\ state_j = state_{j-n} \iff state_{n-i} = state_n = state_{n-1}\end{array}$$

Which makes A capable of taking in some $S_{\langle i..n..j..\infty \rangle}$ as not all $s \in S_{\langle i..\infty \rangle}$ have to be considered at once. In other words, the input data does not need to persist across the history of A, only the effect of s on *state* must be persisted. Additionally, the effect of *opt* is determined by the *body* within *result* such that

$$\begin{aligned}&A(state_n,\ S(t_{j-n}),\ opt)\\ &\quad\equiv A(state_i\ S(t_{j-i}))\\ &\quad\equiv A(state_i,\ S(t_{j-i}),\ opt)\\ &\quad\equiv A(state_n,\ S(t_{j-n}))\end{aligned}$$

implying that the effect of *opt* doesn't prevent backwards compatibility of *state*.

6.5.1 Domain

- Algorithm State
- Option(s)

6.5.2 Range

- Algorithm State

6.6 Algorithm Formal Definition

In previous sections, $A\_$ was used to indicate calling an Algorithm. In the rest of this document, that notation will be replaced with $algorithm\_$. This new notation is defined using the definitions of Algorithm Components presented above. The previous definition of an Algorithm

$$Algorithm ::= Init \gg Relevant? \gg Accept? \gg Step \gg Result$$

can be refined using the Operation *recur* and Primitive *algorithmIter* (defined in following subsections) to illustrate how an Algorithm processes a Collection of Statement(s).

$$\begin{array}{|l}
\hline Algorithm[KV, Collection, KV]\\
AlgorithmIter,\ Recur,\ Init,\ Result\\
opt?, state?, state! : KV\\
S? : Collection \bullet \forall s? \in S? \mid s? : STATEMENT\\
algorithm_ : KV \times Collection \times KV \twoheadrightarrow KV\\
\hline
algorithm = \langle init_, \langle algorithmIter_,\ recur_\rangle^{\#S?},\ result_\rangle\\
state! = algorithm(state?, S?, opt?) \bullet\\
\quad let\ \ init! == init(state?) \bullet\\
\quad \forall s_n \in S? \mid s_n : STATEMENT, n : \mathbb{N} \bullet i \le n \le j\ \bullet\\
\quad\quad \exists_1 state_n \mid state_n : KV \bullet\\
\quad\quad\quad let\ \ S?_n = tail(S?)^{n-i}\\
\quad\quad\quad\quad state_i = algorithmIter(init!, S?_n) \Rightarrow S?_n = S? \iff n = i\\
\quad\quad\quad\quad state_n = recur(state_i,\ S?_n,\ _algorithmIter_)^{j-1} \iff n \neq i \land n \neq j\\
\quad\quad\quad\quad state_j = recur(state_n, (\{j-1, j\} \restriction S?), _algorithmIter_) \iff n = j\\
\quad\quad\quad\quad state_{j+1} = state_j \Rightarrow recur(state_j, (j \restriction S?), _algorithmIter_) \iff n = j+1\\
\quad = result(state_j, opt?)\\
\hline
\end{array}$$

Within the schema above, the following notation is intended to show that *algorithm* is a Primitive ⇒ Collection of Primitives and/or Operations.

$$\langle\, init\_, \langle\, algorithmIter\_, recur\_\rangle^{\# S?}, result\_\rangle$$

Within that notation, the following notation is intended to represent the iteration through the Statement(s) via tail recursion.

$$\langle\, algorithmIter\_, recur\_\rangle^{\# S?}$$

which implies that each Statement is passed to $algorithmIter\_$ and the result is then passed on to the next iteration of the loop. The completion of this loop is the prerequisites of $result\_$

6.6.1 Recur

The following schema introduces the Operation *recur* which expects an accumulator (KV), a *Collection* of Value(s) (V) being iterated over and a function ($\_ \rightarrowtail \_$) which will be called as the result of *recur*. This Operation has been written to be as general purpose as possible and represents the ability to perform tail recursion. Given this intention, *recur* must only ever be the last Operation within a Primitive

$$\begin{array}{|l}
p_{i\,..\,j} : \mathrm{seq}_1 \bullet \forall o \in p \mid o : _ \rightarrowtail _ \\
\hline
p_{i\,..\,j} = \langle \forall n : \mathbb{N} \mid i \leq n \leq j \wedge o_n \in p_{i\,..\,j} \bullet \\
\qquad\qquad \exists_1 o_n \bullet o_n \neq recur \vee o_n = recur \iff n = j\rangle \Rightarrow \\
\qquad\qquad\qquad front(p_{i\,..\,j}) \upharpoonright recur = \langle\rangle
\end{array}$$

and results in a call to the passed in function where the accumulator $ack?$ and the Collection (minus the first member) are passed as arguments to $fn?$. If this would result in the empty Collection ($\langle\rangle$) being passed to $fn?$, instead the accumulator $ack?$ is returned.

$$\begin{array}{|l}
\hline
Recur[KV, Collection, (_ \rightarrowtail _)] \\
ack? : KV \\
S? : Collection \\
fn? : (_ \rightarrowtail _) \\
recur_ : KV \times Collection \times (_ \rightarrowtail _) \leftrightarrow (KV \times Collection \rightarrowtail _) \\
\hline
recur(ack?, S?, fn?) = fn?(ack?, tail(S?)) \iff tail(S?) \neq \langle\rangle \\
recur(ack?, S?, fn?) = first(ack?, tail(S?)) \iff tail(S?) = \langle\rangle \\
\hline
\end{array}$$

In the context of Algorithms,

$$\begin{array}{l}
ack? = AlgorithmState \\
S? = CollectionofStatement(s) \\
fn? = algorithmIter
\end{array}$$

6.6.2 Algorithm Iter

The following schema introduce the Primitive *algorithmIter* which demonstrates the life cycle of a single statement as its passed through the components of an Algorithm.

$$
\begin{array}{l}
\textbf{AlgorithmIter}[KV, Collection] \\
\hline
Relevant?\,,\ Accept?\,,\ Step \\
state?\,, state! : KV \\
S? : Collection \\
s? : STATEMENT \\
algorithmIter\,\_ : KV \times STATEMENT \twoheadrightarrow KV \\
\hline
algorithmIter = \langle\, relevant?\ \_\,,\ accept?\ \_\,,\ step\,\_\rangle \\
s? = head(S?\,) \\
state! = algorithmIter(state?\,, s?\,) \bullet \\
\quad let\quad relevant! == relevant?\,(state?\,, s?\,) \\
\qquad\quad accept! == accept?\,(state?\,, s?\,) \\
\qquad\quad step! == step(state?\,, s?\,) \\
\quad\ = (state? \iff relevant! = false \lor accept! = false) \lor \\
\qquad (step! \iff relevant! = true \land accept! = true)
\end{array}
$$

If a statement if both relevant and acceptable, *state*! will be the result of *step*. Otherwise, the passed in state is returned $\Rightarrow$ *step*! = *state*?.

7 Foundational Operations

The Operations in this section use the Operations pulled from the Z Reference Manual (section 1,4) within their own definitions. They are defined as Operations opposed to Primitives because they represent core functionality needed in the context of processing xAPI data given the definition of an Algorithm above. As such, these Operations are added to the global dictionary of symbols usable, without a direct reference to the components schema, within the definition of Operations and Primitives throughout the rest of this document. In general, Operations are intended to be simple, and should not contain any recursive calls. They are building blocks which are used across Primitives of varying functionality. When defining an Operation not already in the set of Foundational Operations defined here, its schema MUST be referenced at the top of all Schemas which utilize the new Operation.

7.1 Collections

Operations which expect a Collection $X = \langle x_i .. x_n .. x_j \rangle$

7.1.1 Array?

The operation *array*? will return a boolean which indicates if the passed in argument is a Collection

$$
\begin{array}{|l}
\hline
\textit{Array?}\,[V] \\
\textit{coll?} : V \\
\textit{bol!} : \textit{Boolean} \\
\textit{array?}\ \_ : V \to \textit{Boolean} \\
\hline
\textit{bol!} = \textit{array?}\,(\textit{coll?}\,) \bullet \textit{bol!} = \textit{true} \iff \textit{coll?} : \textit{Collection} \Rightarrow V \setminus (\textit{Scalar}, \textit{KV}) \\
\hline
\end{array}
$$

where $V \setminus (Scalar, KV)$ is used to indicate that *coll*? is of type V

$$V ::= Scalar \mid Collection \mid KV$$

but in order for $bol! = true$, *coll*? must not be of type $Scalar \vee KV$ such that

$$
\begin{array}{ll}
X = \langle x_0, x_1, x_2, x_3, x_4 \rangle & \\
\quad x_0 = 0 & \\
\quad x_1 = foo & \\
\quad x_2 = \langle baz,\ qux \rangle & \\
\quad x_3 = \langle\!\langle abc \mapsto 123,\ def \mapsto 456 \rangle\!\rangle & \\
\quad x_4 = \langle \langle\!\langle ghi \mapsto 789,\ jkl \mapsto 101112 \rangle\!\rangle,\ \langle\!\langle ghi \mapsto 131415,\ jkl \mapsto 161718 \rangle\!\rangle \rangle & \\
array?\,(X) = true & \text{[collection by definition]} \\
array?\,(x_2) = true & \text{[collection of } 0 \mapsto baz,\ 1 \mapsto qux\text{]}
\end{array}
$$

$$
\begin{aligned}
&array?\,(x_4) = true && \text{[collection of maps]}\\
&array?\,(x_0) = false && \text{[Scalar]}\\
&array?\,(x_1) = false && \text{[String]}\\
&array?\,(x_3) = false && \text{[Map]}
\end{aligned}
$$

7.1.2 Append

The operation *append* will return a Collection with a Value added at a specified numeric Index.

$$
\begin{array}{|l}
\hline
Append[Collection, V, \mathbb{N}]\\
coll?\,, coll! : Collection\\
v? : V\\
idx? : \mathbb{N}\\
append\_ : Collection \times V \times \mathbb{N} \rightarrowtail\!\!\!\!\rightarrow Collection\\
\hline
\#\, idx? = 1\\
coll! = append(coll?\,, v?\,, idx?\,) \bullet\\
\quad\quad let\ coll' == front(\{\, i : \mathbb{N} \mid i \in 0 \mathinner{\ldotp\ldotp} idx?\,\} \upharpoonright coll?\,) \frown v?\\
\quad\quad\quad coll'' == \{\, j : \mathbb{N} \mid j \in idx? \mathinner{\ldotp\ldotp} \#\, coll?\,\} \upharpoonright coll?\\
\quad\quad = coll' \frown coll'' \Rightarrow\\
\quad\quad\quad\quad (front(coll') \frown v? \frown coll'') \land\\
\quad\quad\quad\quad (\, v? \mapsto idx? \in coll!\,) \land\\
\quad\quad\quad\quad (\, \#\, coll! = \#\, coll? + 1)\\
\hline
\end{array}
$$

append results in the composition of $coll'$ and $coll''$ such that

$$coll! = coll' \frown coll'' \ \land\ idx? \mapsto v? \in coll!$$

- $coll'$ is the items in $coll?$ up to and including $idx?$ but the value at $idx?$ is replaced with $v?$ such that $idx? \mapsto coll?_{idx?} \notin coll'$
- $coll''$ is the items in $coll?$ from $idx?$ to $\#\, coll? \Rightarrow coll?_{idx?} \in coll''$

The following example illustrates these properties.

$$
\begin{aligned}
&X = \langle x_0, x_1, x_2 \rangle\\
&\quad x_0 = 0\\
&\quad x_1 = foo\\
&\quad x_2 = \langle a, b, c \rangle\\
&\quad v? = bar\\
&append(X, v?\,, 0) = \langle bar, 0, foo, \langle a, b, c \rangle \rangle\\
&append(X, v?\,, 1) = \langle 0, bar, foo, \langle a, b, c \rangle \rangle
\end{aligned}
$$

$$append(X, v?, 2) = \langle 0, foo, bar, \langle a, b, c \rangle \rangle$$
$$append(X, v?, 3) = \langle 0, foo, \langle a, b, c \rangle, bar \rangle$$
$$append(X, v?, 4) = append(X, v?, 3) \iff 3 \notin \operatorname{dom} X$$

7.1.3 Remove

The inverse of the *append* Operations.

$$remove(coll, idx) =^{\sim} append(coll, idx)$$

The operation *remove* will return a Collection minus the Value removed from the specified Numeric Index

$$
\begin{array}{|l}
\hline
Remove[Collection, \mathbb{N}] \\
coll?, coll! : Collection \\
idx? : \mathbb{N} \\
remove\_ : Collection \times \mathbb{N} \twoheadrightarrow Collection \\
\hline
\# idx? = 1 \\
coll! = remove(coll?, idx?) \bullet \\
\quad let\ coll' == front(\{ i : \mathbb{N} \mid i \in 0 \mathrel{..} idx? \} \upharpoonright coll?) \\
\qquad coll'' == tail(\{ j : \mathbb{N} \mid j \in idx? \mathrel{..} \# coll? \} \upharpoonright coll?) \\
\quad = coll' \frown coll'' \Rightarrow \\
\qquad (coll?_{idx?} \notin coll') \land \\
\qquad (coll?_{idx?} \notin coll'') \land \\
\qquad (\# coll! = \# coll? - 1) \\
\hline
\end{array}
$$

such that

$$X = \langle x_0, x_1, x_2 \rangle$$
$$x_0 = 0$$
$$x_1 = foo$$
$$x_2 = baz$$
$$remove(X, 0) = \langle foo, baz \rangle \qquad \text{[0 was removed from } X\text{]}$$
$$remove(X, 1) = \langle 0, baz \rangle \qquad \text{[foo was removed from } X\text{]}$$
$$remove(X, 2) = \langle 0, foo \rangle \qquad \text{[baz was removed from } X\text{]}$$
$$remove(X, 3) = \langle 0, foo, baz \rangle = X \qquad \text{[nothing at 3, X unaltered]}$$

7.1.4 At Index

The operation *atIndex* will return the Value at a specified numeric index within a Collection or an empty Collection if there is no value at the specified index.

$$\begin{array}{|l}
\multicolumn{1}{l}{AtIndex[Collection, \mathbb{N}]} \\
\hline
idx? : \mathbb{N} \\
coll? : Collection \\
atIndex\,_ : Collection \times \mathbb{N} \twoheadrightarrow V \\
\hline
\#\, idx? = 1 \\
coll! = atIndex(coll?\,, idx?\,) = (head\,(\,idx? \upharpoonright coll?\,)) \iff idx? \in coll? \\
coll! = atIndex(coll?\,, idx?\,) = \langle\rangle \iff idx? \notin coll? \\
\hline
\end{array}$$

Given the definition of the *Collection* and V free types

$$\begin{array}{l}
Collection ::== emptyColl \mid append\langle\!\langle Collection \times Scalar \vee Collection \vee KV \times \mathbb{N}\rangle\!\rangle \\
V ::= Scalar \mid Collection \mid KV
\end{array}$$

The collection member $coll?_{idx?} : V$ is implied from *append* accepting the argument of type $Scalar \vee Collection \vee KV \equiv V$ which means each Collection member is of type V. Given that extraction ($\_ \upharpoonright \_$) returns a Collection,

$$\begin{array}{|l}
\operatorname{seq} X : Collection \\
\hline
_ \upharpoonright _ : \mathbb{P}\,\mathbb{N}_1 \times \operatorname{seq} X \to \operatorname{seq} X
\end{array}$$

in order for *atIndex* to return the collection member without altering its type, the first member of $atIdx'$ must be returned, not $atIdx'$ itself.

$$\begin{array}{|l}
atIdx' : Collection \\
coll!\,, coll?_{idx?} : V \\
\hline
atIdx' = (\,idx? \upharpoonright coll?\,) \Rightarrow \langle coll?_{idx?}\,\rangle \\
coll! = head(atIdx') = coll?_{idx?}
\end{array}$$

The *head* call is made possible by restricting $idx?$ to be a single numeric value.

$$\begin{array}{l}
idx?\,, idx' : \mathbb{N} \\
\quad \#\, idx? = 1 \bullet (\,idx? \upharpoonright coll?\,) = \langle coll?_{idx?}\,\rangle \bullet \\
\qquad (head(\,idx? \upharpoonright coll?\,)) = coll?_{idx?} \qquad \text{[expected return given } idx?\text{]} \\
\quad \#\, idx' \geq 2 \bullet (\,idx' \upharpoonright coll?\,) = \langle coll?_{idx'_i} \;..\; coll?_{idx'_j}\,\rangle \bullet \\
\qquad (head(\,idx' \upharpoonright coll?\,)) = coll?_{idx'_i} \qquad \text{[unexpected return given } idx'\text{]}
\end{array}$$

Additionally, if the provided $idx? \notin coll?$ then an empty Collection will be returned given that *head* must be passed a non-empty Collection.

$$\begin{array}{|l}
head : \operatorname{seq}_1 X \to X \\
\hline
idx? \notin coll? \Rightarrow (\,idx? \upharpoonright coll?\,) = \langle\rangle \neg \operatorname{seq}_1
\end{array}$$

The properties of *atIndex* are illustrated in the following examples.

$$X = \langle x_0, x_1, x_2 \rangle$$

$$
\begin{aligned}
&\quad x_0 = 0 \\
&\quad x_1 = foo \\
&\quad x_2 = \langle a, b, c \rangle \\
&atIndex(X, 0) = 0 && [head\,(\langle\, x_0 \,\rangle)] \\
&atIndex(X, 1) = foo && [head\,(\langle\, x_1 \,\rangle)] \\
&atIndex(X, 2) = \langle a, b, c \rangle && [head\,(\langle\, x_2 \,\rangle)] \\
&atIndex(X, 3) = \langle\rangle && [3 \notin X \Rightarrow x_3 \notin X]
\end{aligned}
$$

7.1.5 Update

The operation *update* will return a Collection $coll!$ which is the same as the input Collection $coll?$ except for at index $idx?$. The existing member $coll?_{idx?}$ is replaced by the provided Value $v?$ at $idx?$ in $coll!$ such that

$$idx? \mapsto v? \in coll! \land idx? \mapsto coll?_{idx?} \notin coll!$$

which is equivalent to $remove \gg append$

$$update(coll?\,, v?\,, idx?\,) \equiv append(remove(coll?\,, idx?\,), v?\,, idx?\,)$$

The functionality of *update* is further explained in the following schema.

$$
\begin{array}{|l}
\multicolumn{1}{l}{Update[Collection, V, \mathbb{N}]} \\
\hline
idx? : \mathbb{N} \\
coll?\,, coll! : Collection \\
v? : V \\
update\_ : Collection \times V \times \mathbb{N} \rightarrowtail Collection \\
\hline
1 = \#\, idx? \\
coll! = update(coll?\,, v?\,, idx?\,) \bullet \\
\qquad let\ coll' == \{\, i : \mathbb{N} \mid i \in 0 \mathrel{..} idx? \,\} \upharpoonright coll? \\
\qquad\quad coll'' == head(coll') \frown v? \\
\qquad\quad coll''' == \{\, j : \mathbb{N} \mid j \in idx? + 1 \mathrel{..} \#\, coll? \,\} \upharpoonright coll? \\
\qquad = coll'' \frown coll'' \Rightarrow \\
\qquad\qquad (append(remove(coll', idx?\,), v?\,, idx?\,) \frown coll'') \land \\
\qquad\qquad (\, v? \mapsto idx? \in coll!\,) \land \\
\qquad\qquad (\,\#\, coll! = \#\, coll?\,) \land \\
\hline
\end{array}
$$

The value which previously existed at $idx? \in coll?$ is replaced with $v?$ to result in $coll!$

- $coll'$ is the items in $coll?$ up to and including $idx?$
- $coll''$ is the items in $coll?$ except the item at $idx?$ has been replaced with $v?$

- $coll'''$ is the items in $coll?$ from $idx? + 1$ to $\# coll? \Rightarrow coll?_{idx?} \notin coll''$

The following example illustrates these properties.

$$
\begin{aligned}
&X = \langle x_0, x_1, x_2 \rangle \\
&\quad x_0 = 0 \\
&\quad x_1 = foo \\
&\quad x_2 = \langle a, b, c \rangle \\
&\quad v? = bar \\
&update(X, v?, 0) = \langle bar, foo, \langle a, b, c \rangle \rangle \\
&update(X, v?, 1) = \langle 0, bar, \langle a, b, c \rangle \rangle \\
&update(X, v?, 2) = \langle 0, foo, bar \rangle \\
&update(X, v?, 3) = \langle 0, foo, \langle a, b, c \rangle, bar \rangle \\
&update(X, v?, 4) = append(X, v?, 3) = update(X, v?, 3) \iff 3 \notin \operatorname{dom} X
\end{aligned}
$$

7.2 Key Value Pairs

Operations which expect a Map $M = \langle\!\langle k_i v_{k_i} .. k_n v_{k_n} .. k_j v_{k_j} \rangle\!\rangle$

7.2.1 Map?

The operation $map?$ will return a boolean which indicates if the passed in argument is a KV

$$
\begin{array}{|l}
\hline
Map?\,[V] \\
m? : V \\
bol! : Boolean \\
map?\,\_ : V \rightarrow Boolean \\
\hline
bol! = map?\,(m?) \bullet bol! = true \iff m? : KV \Rightarrow V \setminus (Scalar, Collection) \\
\hline
\end{array}
$$

where $V \setminus (Scalar, Collection)$ is used to indicate that $m?$ is of type V

$$V ::= Scalar \mid Collection \mid KV$$

but in order for $bol! = true$, $m?$ must not be of type $Scalar \vee Collection$ such that

$$
\begin{aligned}
&X = \langle\!\langle x_0, x_1, x_2, x_3, x_4 \rangle\!\rangle \\
&\quad x_0 = 0 \\
&\quad x_1 = foo \\
&\quad x_2 = \langle baz,\ qux \rangle \\
&\quad x_3 = \langle\!\langle abc \mapsto 123,\ def \mapsto 456 \rangle\!\rangle
\end{aligned}
$$

$$x_4 = \langle\langle\!\langle ghi \mapsto 789,\ jkl \mapsto 101112 \rangle\!\rangle,\ \langle\!\langle ghi \mapsto 131415,\ jkl \mapsto 161718 \rangle\!\rangle\rangle$$

$map?(X) = true$ [KV by definition]

$map?(x_3) = true$ [KV]

$map?(x_2) = false$ [Collection]

$map?(x_4) = false$ [Collection of maps]

$map?(x_0) = false$ [Scalar]

$map?(x_1) = false$ [String]

7.2.2 Associate

The operation *associate* establishes a relationship between $k?$ and $v?$ at the top level of $m!$.

$$
\begin{array}{|l}
\hline
Associate[KV, K, V] \\
m?, m!, m' : KV \\
k? : K \\
v? : V \\
associate\_ : KV \times K \times V \rightarrowtail KV \\
\hline
m! = associate(m?, k?, v?) \bullet \\
\quad let\ m' == m? \ntriangleleft k? \Rightarrow \\
\qquad\qquad (\mathrm{dom}\ m' = \mathrm{dom}\,(m? \setminus k?)) \land \\
\qquad\qquad (m? \setminus m' = k? \iff k? \in m?) \land \\
\qquad\qquad (m? \setminus m' = \emptyset \iff k? \notin m? \Rightarrow m? = m') \\
\quad = \langle\!\langle k? \mapsto v? \rangle\!\rangle \cup m' \\
\hline
\end{array}
$$

This implies that any existing mapping at $k? \in m?$ will be overwritten by *associate* but an existing mapping is not a precondition.

$$
\begin{array}{|l}
(k?, m?_{k?}) \in m? \lor (k?, m?_{k?}) \notin m? \\
(k?, m?_{k?}) \notin m! \\
(k?, v?) \in m! \\
\hline
m! = associate(m?, k?, v?)
\end{array}
$$

associate does not alter any other mappings within $m?$ and this property is illustrated by the definition of local variable m'

$$
\begin{array}{|l}
m' : KV \mid m' = m? \ntriangleleft k? \Rightarrow m' \lhd (m? \setminus k?) \\
\hline
\mathrm{dom}\ m? = \{ k_i : K \mid 0 \mathinner{..} \# m? \bullet k_i \in m? \land 0 \leq i \leq \# m? \} \\
\mathrm{dom}\ m' = \{ k_i' : K \mid 0 \mathinner{..} \# m' \bullet k_i' \in m? \land k_i' \neq k? \land 0 \leq i \leq \# m' \} \\
\mathrm{dom}\ m' = \mathrm{dom}\ m? \iff k? \notin m? \Rightarrow \forall k_i \in m? \mid k_i \neq k? \\
\# m' = \# m? \iff k? \notin m? \\
\# m' = \# m? - 1 \iff k? \in m?
\end{array}
$$

and its usage within the definition of *associate*.

$$m! = m? \cup \langle\!\langle\, k? \mapsto v?\, \rangle\!\rangle \Rightarrow k? \notin m?$$
$$m! = m' \,\cup\, \langle\!\langle\, k? \mapsto v?\, \rangle\!\rangle \Rightarrow m' \neq m? \wedge k? \in m?$$

The following examples demonstrate the intended functionality of *associate*.

$$\begin{aligned}
&M = \langle\!\langle k_0 v_{k_0}, k_1 v_{k_1} \rangle\!\rangle \\
&\quad k_0 = abc \;\wedge v_{k_0} = 123 && [k_0 v_{k_0} = abc \mapsto 123] \\
&\quad k_1 = def \;\wedge v_{k_1} = xyz \mapsto 456 && [k_1 v_{k_1} = def \mapsto xyz \mapsto 456] \\
&associate(M, baz, foo) = \langle\!\langle abc \mapsto 123,\; def \mapsto xyz \mapsto 456,\; baz \mapsto foo \rangle\!\rangle \\
&associate(M, abc, 321) = \langle\!\langle abc \mapsto 321,\; def \mapsto xyz \mapsto 456 \rangle\!\rangle
\end{aligned}$$

7.2.3 Dissociate

The operation *dissociate* will remove some $k \mapsto v$ from KV given $k \in KV$

$$\begin{array}{|l}
\hline
Dissociate[KV, K] \\
m?, m! : KV \\
k? : K \\
dissociate_ : KV \times K \twoheadrightarrow KV \\
\hline
m! = dissociate(m?, k?) \bullet m! = m? \lhd k? \Rightarrow \\
\qquad (\mathrm{dom}\ m! = \mathrm{dom}\,(\, m? \setminus k?\,)) \wedge \\
\qquad (\, m? \setminus m! = k? \iff k? \in m?\,) \wedge \\
\qquad (\, m? \setminus m! = \emptyset \iff k? \notin m? \Rightarrow m? = m!\,) \wedge \\
\qquad ((k?, m?_{k?}\,) \notin m!\,) \\
\hline
\end{array}$$

such that every mapping in $m?$ is also in $m!$ except for $k? \mapsto m?_{k?}$.

$$\begin{aligned}
&M = \langle\!\langle k_0 v_{k_0}, k_1 v_{k_1} \rangle\!\rangle \\
&\quad k_0 = abc \;\wedge v_{k_0} = 123 && [k_0 v_{k_0} = abc \mapsto 123] \\
&\quad k_1 = def \;\wedge v_{k_1} = xyz \mapsto 456 && [k_1 v_{k_1} = def \mapsto xyz \mapsto 456] \\
&dissociate(M, abc) = \langle\!\langle def \mapsto xyz \mapsto 456 \rangle\!\rangle \\
&dissociate(M, def) = \langle\!\langle abc \mapsto 123 \rangle\!\rangle \\
&dissociate(M, xyz) = M && [xyz \notin M]
\end{aligned}$$

7.2.4 At Key

The operation *atKey* will return the Value v at some specified Key k.

$$
\begin{array}{l}
\underline{\quad AtKey[KV, K] \quad} \\
m? : KV \\
v! : V \\
k? : K \\
atKey\_ : KV \times K \twoheadrightarrow V \\
\hline
v! = atKey(m?, k?) \bullet \\
\qquad\qquad let\ coll == ((\operatorname{seq} m?) \upharpoonright (k?, m?_{k?})) \Rightarrow \langle (k?, m?_{k?}) \rangle \iff k? \in \operatorname{dom} m? \\
\quad = (second(head(coll)) \iff k? \mapsto m?_{k?} \in coll) \vee \\
\qquad (\emptyset \iff k? \notin \operatorname{dom} m?)
\end{array}
$$

In the schema above, $coll$ is the result of filtering for $(k?, m?_{k?})$ within $\operatorname{seq} m?$. If the mapping was in the original $m?$, it will also be in the sequence of mappings. This means we can filter over the sequence to look for the mapping and if found, it is returned as $\langle (k?, m?_{k?}) \rangle$. To return the mapping itself, $head(coll)$ is used to extract the mapping such that the value mapped to $k?$ can be returned.

$$v! = atKey(m?, k?) = second(head(coll)) = m?_{k?} \bullet m?_{k?} : V \iff k? \in \operatorname{dom} m?$$

The following examples demonstrate the properties of $atKey$

$$
\begin{array}{ll}
M = \langle\!\langle k_0 v_{k_0}, k_1 v_{k_1} \rangle\!\rangle & \\
\quad k_0 = abc \ \wedge v_{k_0} = 123 & [k_0 v_{k_0} = abc \mapsto 123] \\
\quad k_1 = def \ \wedge v_{k_1} = xyz \mapsto 456 & [k_1 v_{k_1} = def \mapsto xyz \mapsto 456] \\
atKey(M, abc) = 123 & \\
atKey(M, def) = xyz \mapsto 456 & \\
atKey(M, foo) = \emptyset &
\end{array}
$$

7.3 Utility

Operations which are useful in many Statement processing contexts.

7.3.1 Map

The *map* operation takes in a function $fn?$, Collection $coll?$ and additional Arguments $args?$ (as necessary) and returns a modified Collection $coll!$ with members $fn!_n$. The ordering of $coll?$ is maintained within $coll!$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Map[(\_ \pfun \_), Collection, V] \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
fn?: (\_ \nrightarrow \_) \\
args?: V \\
coll?, coll!: Collection \\
map\_: (\_ \nrightarrow \_) \times Collection \times V \twoheadrightarrow Collection \\
\hline
coll! = map(fn?, coll?, args?) \bullet \\
\qquad\qquad \langle \forall n : i \,..\, j \in coll? \mid i \leq n \leq j \land j = \# coll? \bullet \\
\qquad\qquad\qquad \exists_1 fn!_n : V \mid fn!_n = \\
\qquad\qquad\qquad\qquad (fn?\,(coll?_n, args?) \iff args? \neq \emptyset) \lor \\
\qquad\qquad\qquad\qquad (fn?\,(coll?_n) \iff args? = \emptyset)\rangle \Rightarrow fn!_i \frown fn!_n \frown fn!_j \\
\hline
\end{array}
$$

Above, $fn!_n$ is introduced to handle the case where $fn?$ only requires a single argument. Additional arguments may be necessary but if they are not ($args? = \emptyset$) then only $coll?_n$ is passed to $fn?$.

$$
\begin{aligned}
&X = \langle 1, 2, 3 \rangle \\
&\quad map\,(succ, X) = \langle 2, 3, 4 \rangle && [\text{increment each member of } X] \\
&\quad map\,(+, X, 2) = \langle 3, 4, 5 \rangle && [\text{add 2 to each member of } X]
\end{aligned}
$$

7.3.2 Iso To Unix Epoch

The $isoToUnix$ operation converts an ISO 8601 Timestamp (see the xAPI Specification) to the number of seconds that have elapsed since January 1, 1970

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad IsoToUnix \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
Timestamp \\
seconds!: \mathbb{N} \\
isoToUnix\_: \mathbb{F}_1 \rightarrow \mathbb{N} \\
\hline
seconds! = isoToUnix(timestamp) \\
\hline
\end{array}
$$

$$
\begin{aligned}
&ts = 2015 - 11 - 18T12 : 17 : 00 + 00 : 00 \equiv 2015 - 11 - 18T12 : 17 : 00Z \\
&isoToUnixEpoch(ts) = 1447849020 && [\text{ISO 8601} \rightarrow \text{Epoch time}]
\end{aligned}
$$

7.3.3 Timeunit To Number of Seconds

The operation $toSeconds$ will return the number of seconds corresponding to the input $Timeunit$

$$Timeunit ::= second \mid minute \mid hour \mid day \mid week \mid month \mid year$$

such that the following schema defines $toSeconds$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ToSeconds[Timeunit] \qquad\qquad\qquad\qquad\qquad\qquad}} \\
t? : Timeunit \\
toSeconds\_ : Timeunit \nrightarrow \mathbb{N} \\
\hline
toSeconds(t?) = 1 \iff t? = second \\
toSeconds(t?) = 60 \iff t? = minute \\
toSeconds(t?) = 3600 \iff t? = hour \\
toSeconds(t?) = 86400 \iff t? = day \\
toSeconds(t?) = 604800 \iff t? = week \\
toSeconds(t?) = 2629743 \iff t? = month \\
toSeconds(t?) = 31556926 \iff t? = year \\
\hline
\end{array}
$$

7.4 Rate Of

The Operation $rateOf$ calculates the number of times something occurs within an interval of time given a unit of time.

$$rateOf(nOccurances, start, end, unit)$$

Where the output translates to: the rate of occurrence per unit within interval

- $nOccurances$ is the number of times something happened and should be an Integer (called $nO?$ bellow)
- $start$ is an ISO 8601 timestamp which serves as the first timestamp within the interval
- end is an ISO 8601 timestamp which servers as the last timestamp within the interval
- $unit$ is a String Enum representing the unit of time

This can be seen in the definition of $rateOf$ bellow.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad RateOf[\mathbb{N}, TIMESTAMP, TIMESTAMP, TIMEUNIT] \quad}} \\
nO? : \mathbb{N} \\
rate! : \mathbb{Z} \\
start?, end? : TIMESTAMP \\
unit? : TIMEUNIT \\
rateOf\_ : \mathbb{N} \times TIMESTAMP \times TIMESTAMP \times TIMEUNIT \rightarrow \mathbb{Z} \\
\hline
rate! = rateOf(nO?, start?, end?, unit?) \bullet \\
\qquad let \quad interval == isoToUnix(end) - isoToUnix(start) \\
\qquad\qquad unitS == toSeconds(unit?) \\
\qquad\; = nO? \div (interval \div units) \\
\hline
\end{array}
$$

The only other functionality required by $rateOf$ is supplied via basic arithmetic

$$
\begin{aligned}
&start = 2015-11-18T12:17:00Z\\
&end = 2015-11-18T14:17:00Z\\
&unit = second\\
&nO? = 10\\
&\quad startN = isoToUnix(start) = 1447849020\\
&\quad endN = isoToUnix(end) = 1447856220\\
&\quad interval = endN - StartN = 7200\\
&\quad unitN = toSeconds(unit) = 60\\
&0.001389 = rateOf(nO?, start, end, unit) \Rightarrow 10 \div (7200 \div 60)\\
&5 = rateOf(nO?, start, end, hour) \Rightarrow 10 \div (7200 \div 3600)
\end{aligned}
$$

8 Common Primitives

There will be many Primitives used within Algorithm definitions in DAVE but navigation into a nested *Collection* or *KV* is most likely to be used across nearly all Algorithm definitions. In the following section, helper Operations are introduced for navigation into and back out of a nested Value. These Operations are then used to define the common Primitives centered around traversal of nested data structures ie. xAPI Statements and Algorithm State.

8.1 Traversal Operations

$$
\begin{array}{l}
\underline{\quad Get[V, Collection] \qquad\qquad\qquad\qquad\qquad\qquad} \\
in?, v! : V \\
id? : Collection \\
get\_ : V \times Collection \twoheadrightarrow V \\
\hline
v! = get(in?, id?) \bullet \\
\qquad = (atIndex(in?, head(id?)) \iff (array?(in?) = true) \land (head(id?) \in \mathbb{N})) \lor \\
\qquad\quad (atKey(in?, head(id?)) \iff (array?(in?) = false) \land (map?(in?) = true)) \\
\hline
\end{array}
$$

- retrieval of a V located at $id?$ within $in?$ where $in?$ can be a *Collection* or *KV*

$$
\begin{array}{l}
\underline{\quad Merge[(V, V), Collection] \qquad\qquad\qquad\qquad\qquad} \\
parent?, child?, parent! : V \\
at? : Collection \\
merge\_ : (V \times V) \times Collection \rightarrowtail\!\!\!\!\rightarrow V \\
\hline
parent! = merge((parent?, child?), at?) \bullet \\
\qquad = (associate(parent?, head(at?), child?) \\
\qquad\qquad \iff map?(parent?) = true) \lor \\
\qquad\quad (update(parent?, child?, head(at?)) \\
\qquad\qquad \iff (array?(parent?) = true) \land (head(at?) \in \mathbb{N})) \\
\hline
\end{array}
$$

- Updating of $parent?$ to include $child?$ at location indicated by $head(at?)$

$$
\begin{array}{l}
\underline{\quad Conj[V, V] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
parent?, data? : V \\
conj! : Collection \\
conj\_ : V \times V \rightarrowtail\!\!\!\!\rightarrow Collection \\
\hline
conj! = conj(parent?, data?) \bullet \\
\qquad\qquad let\ \ j == first(last(parent?)) \\
\qquad\qquad\qquad parent?_{coll} == append(\langle\rangle, parent?, 0) \\
\qquad = (append(\ parent?\ ,\ data?, (j+1)) \iff array?(parent?) = true) \lor \\
\qquad\quad (append(parent?_{coll}\ ,\ data?, (j+1)) \iff array(parent?) = false) \\
\hline
\end{array}
$$

- $conj!$ is a collection with $data?$ at the last index $conj!_j = data?$.

8.2 Traversal Primitives

The helper Operations defined above are used to describe the traversal of a heterogeneous nested Value. In the following subsections, examples which demonstrate the functionality of Primitives will be passed X as $in?$.

$$
\begin{aligned}
& X = \langle x_0, x_1, x_2 \rangle \\
& \quad x_0 = true \\
& \quad x_1 = \langle a, b, c \rangle \\
& \quad x_2 = \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto y, z \mapsto \langle 3, 2, 1 \rangle \rangle\!\rangle \rangle\!\rangle \\
& fn! = fn(X_{\langle path?_i \,..\, path?_{j-1} \rangle}, v?) \bullet \\
& \qquad \forall X_{\langle path?_i \,..\, path?_{j-1} \rangle} \wedge v? \mid fn! = ZZZ \qquad \text{[always return } ZZZ\text{]}
\end{aligned}
$$

8.2.1 Get In

Collection and KV have different Fundamental Operations for navigation into, value extraction from and application of updates to. Navigation into an arbitrary Value without concern for its type is a useful tool to have and has been defined as the Primitive $getIn$.

$$
\begin{array}{|l}
\hline
GetIn[V, Collection] \\
Get, Recur \\
in?, atPath! : V \\
path? : Collection \\
getIn\_ : V \times Collection \twoheadrightarrow V \\
\hline
getIn = \langle get\_, recur\_ \rangle^{\# path? - 1} \\
\\
atPath! = getIn(in?, path?) \bullet \\
\;\forall n : i \,..\, j-1 \bullet j = first(last(path?)) \Rightarrow first(j, path?_j) \mid \exists\, down_n \bullet \\
\qquad let \;\; path?_n == tail(path?)^{n-i} \\
\qquad\quad down_i == get(in?, path?_n) \Rightarrow \\
\qquad\qquad\qquad atIndex(in?, head(path?)) \vee \\
\qquad\qquad\qquad atKey(in?, head(path?)) \iff n = i \\
\qquad\quad down_n == recur(down_i, path?_n, get\_)^{j-1} \\
\qquad\quad down_{j-1} == get(down_n, path?_n) \iff n = j-2 \\
\\
atPath! = down_j = get(down_{j-1}, path?_n) \bullet \\
\qquad\qquad\qquad path?_n \equiv (path? \upharpoonright j) \Rightarrow \\
\qquad\qquad\qquad\quad \langle j \mapsto atIndex(path?, j) \rangle \iff n = j-1 \\
\hline
\end{array}
$$

The following examples demonstrate the functionality of the Primitive $getIn$

$$getIn(X, \langle 1, 1 \rangle) = b$$

$getIn(X, \langle 0 \rangle) = true$

$getIn(X, \langle 2, foo, z, 0 \rangle) = 3$

Additionally, the propagation of an update, starting at some depth within a passed in Value and bubbling up to the top level, such that the update is only applied to values along a specified path as necessary, is also a useful tool to have. The following sections introduce Primitives which address performing these types of updates and ends with a summary of the functional steps described in the sections below. *replaceAt* is introduced first and serves as a point of comparison when describing the more abstract Primitives *backProp* and *walkBack*.

8.2.2 Replace At

The schema *ReplaceAt* uses the helper Operation *merge* to apply updates while climbing up from some arbitrary depth.

$$
\begin{array}{l}
ReplaceAt[V, Collection, V] \\
\hline
GetIn, Merge \\
in?, with?, out! : V \\
path? : Collection \\
replaceAt\_ : V \times Collection \times V \rightarrowtail V \\
\hline
replaceAt = \langle\langle getIn\_,\ merge\_\rangle,\ recur\_\rangle^{\#\, path? - 1} \\
\\
out! = replaceAt(in?, path?, with?) \bullet \\
\quad \forall n : i \mathrel{..} j-1 \bullet (i = first(head(path?))) \wedge (j = first(last(path?))) \mid \exists\, parent_n \bullet \\
\qquad\qquad let\ \ path?_n == tail(path?)^{n-i} \\
\qquad parent_n = recur(parent_{n-1}, path?_n, get\_)^{j-1} \Rightarrow \\
\qquad\qquad let\ \ parent_i == getIn(in?, path?_n) \iff n = i \\
\qquad\qquad\quad parent_{i+1} == getIn(parent_i, path?_n) \iff n = i+1 \\
\qquad\qquad\quad parent_{j-1} == getIn(parent_{j-2}, path?_n) \iff n = j-1 \\
\qquad\qquad parent_j = getIn(parent_{j-1}, (path? \upharpoonright j)) \\
\\
\quad \forall z : p \mathrel{..} q \bullet (p = j-1) \wedge (q = i+1) \Rightarrow \\
\qquad\qquad\quad ((z = p \iff n = j-1) \wedge (z = q \iff n = i+1)) \mid \exists\, child_z \bullet \\
\qquad\qquad let\ \ path?_{rev} == rev(path?) \\
\qquad\qquad\quad path?_z == tail(path?_{rev})^{p-z+1} \\
\qquad child_z = recur((parent_n, child_{n+1}), path?_z, merge\_) \\
\qquad\qquad let\ \ child_p == merge((parent_n, with?), path?_z) \iff z = p \Rightarrow n = j-1 \\
\qquad\qquad\quad child_{p+1} == merge((parent_n, child_p), path?_z) \iff n = j-2 \wedge p = j-1 \\
\qquad\qquad\quad child_q == merge((parent_n, child_{q+1}), path?_z) \iff z = q \Rightarrow n = i+1 \\
\\
out! = merge((in?, child_q), path?_n) \equiv merge((in?, child_q), (path? \upharpoonright i)) \iff (n = i = q-1)
\end{array}
$$

- The range of indices $i \mathrel{..} j-1$ is used to describe navigation into some Value given *path*?

  - Used to reference preceding level of depth
  - keeps track of parent from previous steps

- The range of indices $p \mathbin{..} q$ is used to describe navigation up from target depth indicated by *path*?
  - Used to reference current level of depth
  - keeps track of child after the update has been applied

- The propagation of the update starts with $child_p$
  - *with*? is added to $parent_{j-1}$ at $get\,(path?\,,\langle\, j\,\rangle)$
  - parent nodes need to be notified of the change within their children

The following examples demonstrate the functionality of the Primitive *replaceAt*

$$replaceAt(X, \langle 2, foo, q\rangle, fn!\,) = \langle x_0, x_1, \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto y, q \mapsto ZZZ\rangle\!\rangle\rangle\!\rangle\rangle$$
$$replaceAt(X, \langle 2, foo, x\rangle, fn!\,) = \langle x_0, x_1, \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto ZZZ\rangle\!\rangle\rangle\!\rangle\rangle$$

This Primitive can be made more general purpose by replacing *merge* with a placeholder fn? representing a passed in Operation or Primitive.

8.2.3 Back Prop

Being able to pass a function as an argument allows for, in this context, the arbitrary handling of how update(s) are applied at each level of nesting. The arbitrary fn? should expect a (Parent, Child) tuple and a Collection of indices as arguments and return a potentially modified version of the parent.

$$
\begin{array}{l}
\textit{BackProp}\,[V, \textit{Collection}, V, (\_ \nrightarrow \_)] \\
\hline
\textit{GetIn} \\
\textit{in?}\,, \textit{fnSeed?}\,, \textit{out!} : V \\
\textit{path?} : \textit{Collection} \\
\textit{fn?} : (\_ \nrightarrow \_) \\
\textit{backProp}\,\_ : V \times \textit{Collection} \times V \times (\_ \nrightarrow \_) \rightarrowtail V \\
\hline
\textit{backProp} = \langle\langle \textit{getIn}\,\_, \textit{fn?}\ \_\rangle, \textit{recur}\,\_\rangle^{\#\,\textit{path?}-1} \\
\\
\textit{out!} = \textit{backProp}\,(\textit{in?}\,, \textit{path?}\,, \textit{fnSeed?}\,, \textit{fn?}\,) \bullet \\
\quad \forall n : i \mathrel{..} j-1 \bullet (i = \textit{first}(\textit{head}(\textit{path?}\,))) \wedge (j = \textit{first}(\textit{last}(\textit{path?}\,))) \mid \exists\, \textit{parent}_n \bullet \\
\qquad\qquad \textit{let}\ \ \textit{path?}_n == \textit{tail}(\textit{path?}\,)^{n-i} \\
\qquad\quad \textit{parent}_n = \textit{recur}(\textit{parent}_{n-1}, \textit{path?}_n\,, \textit{get}\,\_)^{j-1} \Rightarrow \\
\qquad\qquad \textit{let}\ \ \textit{parent}_i == \textit{getIn}(\textit{in?}\,, \textit{path?}_n\,) \iff n = i \\
\qquad\qquad\quad \textit{parent}_{i+1} == \textit{getIn}(\textit{parent}_i, \textit{path?}_n\,) \iff n = i+1 \\
\qquad\qquad\quad \textit{parent}_{j-1} == \textit{getIn}(\textit{parent}_{j-2}, \textit{path?}_n\,) \iff n = j-1 \\
\qquad\qquad \textit{parent}_j = \textit{getIn}(\textit{parent}_{j-1}, (\textit{path?} \upharpoonright j)) \\
\\
\quad \forall z : p \mathrel{..} q \bullet (p = j-1) \wedge (q = i+1) \Rightarrow \\
\qquad\qquad\quad ((z = p \iff n = j-1) \wedge (z = q \iff n = i+1)) \mid \exists\, \textit{child}_z \bullet \\
\qquad\qquad \textit{let}\ \ \textit{path?}_{rev} == \textit{rev}(\textit{path?}\,) \\
\qquad\qquad\quad \textit{path?}_z == \textit{tail}(\textit{path?}_{rev}\,)^{p-z+1} \\
\qquad\quad \textit{child}_z = \textit{recur}((\textit{parent}_n, \textit{child}_{n+1}), \textit{path?}_z\,, \textit{fn?}\,) \\
\qquad\qquad \textit{let}\ \ \textit{child}_p == \textit{fn?}\,((\textit{parent}_n, \textit{fnSeed?}\,), \textit{path?}_z\,) \iff z = p \Rightarrow n = j-1 \\
\qquad\qquad\quad \textit{child}_{p+1} == \textit{fn?}\,((\textit{parent}_n, \textit{child}_p), \textit{path?}_z\,) \iff n = j-2 \wedge p = j-1 \\
\qquad\qquad\quad \textit{child}_q == \textit{fn?}\,((\textit{parent}_n, \textit{child}_{q+1}), \textit{path?}_z\,) \iff z = q \Rightarrow n = i+1 \\
\\
\textit{out!} = \textit{fn?}\,((\textit{in?}\,, \textit{child}_q), \textit{path?}_n\,) \equiv \textit{fn?}\,((\textit{in?}\,, \textit{child}_q), (\textit{path?} \upharpoonright i)) \iff (n = i = q-1)
\end{array}
$$

The schema *ReplaceAt* was introduced before *BackProp* so the process underlying both could be explicitly demonstrated and defined. The hope is that this made the introduction of the more abstract Primitive *backProp* easier to follow. A quick comparison of *ReplaceAt* and *BackProp* reveals that the only major difference between them is $\textit{fn?}$ vs $\textit{merge}\,\_$. This implies the Primitive *backProp* can be used to replicate *replaceAt*.

$$
\begin{array}{l}
\textit{replaceAt}(\textit{in?}\,, \textit{path?}\,, \textit{with?}\,) \equiv \\
\quad \textit{backProp}\,(\textit{in?}\,, \textit{path?}\,, \textit{fnSeed?}\,, \textit{merge}\,\_) \iff \textit{with?} = \textit{fnSeed?}
\end{array}
$$

Above highlights the arguments $\textit{with?} \wedge \textit{fnSeed?}$ which serve the same purpose within *backProp* and *replaceAt*.

- Within *ReplaceAt*, the naming $\textit{with?}$ indicates its usage with respect to *merge* and the overall functionality of the Primitive
- Within *BackProp*, the naming $\textit{fnSeed?}$ indicates that the usage of the variable within $\textit{fn?}$ is unknowable but this value will be passed to $\textit{fn?}$ on the very first iteration of the Primitive

In both cases, the variable is put into a tuple and passed to $fn?$.

$$backProp(X, \langle 2, foo, x\rangle, fn!, merge\_) = \langle x_0, x_1, \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto ZZZ\rangle\!\rangle\rangle\!\rangle\rangle$$

The notable limitation of $backProp$ are enumerated in the bullets bellow and the Primitive $walkBack$ is introduced to address them.

- expectation of a seeding value ($fnSeed?$) as a passed in argument
- the general dismissal of the value ($parent_j$) located at $path?$ which is potentially being overwritten

8.2.4 Walk Back

In the Primitive $walkBack$, $fnSeed?$ is assumed to be the result of a function $fn?_\delta$ which is passed in as an argument. $fn?_\delta$ will be passed $parent_j$ as an argument in order to produce $fnSeed?$. This Value will then be used exactly as it was in $backProp$ given $walkBack$ expects another function argument $fn?_{nav}$.

$$walkBack(in?, path?, fn?_\delta, fn?_{nav})$$

In fact, the usage of $fn?_{nav}$ in $WalkBack$ is exactly the same as the usage of $fn?$ in $BackProp$ as $fn?_{nav}$ is passed to $backProp$ as $fn?$.

$$
\begin{array}{|l}
\hline
WalkBack[V, Collection, (\_ \mathrel{\ooalign{$\rightarrow$\cr\hfil$\mapstochar$\hfil}} \_), (\_ \mathrel{\ooalign{$\rightarrow$\cr\hfil$\mapstochar$\hfil}} \_)] \\
BackProp \\
in?, out! : V \\
path? : Collection \\
fn?_\delta, fn?_{nav} : (\_ \nrightarrow \_) \\
walkBack\_ : V \times Collection \times (\_ \nrightarrow \_) \times (\_ \nrightarrow \_) \rightarrowtail\!\!\!\!\rightarrow V \\
\hline
walkBack = \langle getIn\_, \ fn?_\delta\ \_, \ backProp\_\rangle \\
\\
out! = walkBack(in?, path?, fn?_\delta, fn?_{nav}) \bullet \\
\quad let \ \ fnSeed == fn?_\delta\,(getIn(in?, path?)) \\
\quad = backProp(in?, path?, fnSeed, fn?_{nav}) \\
\hline
\end{array}
$$

By replacing $fnSeed?$ with $fn?_\delta$ as an argument

- $walkBack$ can be used to describe predicate based traversal of $in?$
- $walkBack$ can be used to update Values at arbitrary nesting within $in?$ and at the same time describe how those changes affect the rest of $in?$

$walkBack$ serves as a graph traversal template Primitive whose behavior is defined in terms of the nodes within $in?$ and the interpretation of those nodes via $fn?_\delta$ and $fn?_{nav}$. This establishes the means for defining Primitives which can make longitudinal updates as needed before making horizontal movements through some $in?$. In order for $backProp$ to be used in the same way, the required state must be managed by

- fn_{nav}
- some higher level Primitive that contains $backProp$ (see $WalkBack$)

This important difference means $walkBack$ can be used to replicate $backProp$ but the opposite is not always true.

$$\begin{aligned}&walkBack(in?\,, path?\,, fn?_\delta\,, fn?_{nav}\,) \equiv \\ &\quad backProp(in?\,, path?\,, fnSeed?\,, fn?_{nav}\,) \iff fnSeed? = fn?_\delta\,(getIn(in?\,, path?\,))\end{aligned}$$

This means $replaceAt$ can also be replicated.

$$\begin{aligned}&replaceAt(in?\,, path?\,, with?\,) \equiv \\ &\quad (backProp\,(in?\,, path?\,, fnSeed?\,, merge\_) \iff with? = fnSeed?\,) \equiv \\ &\qquad walkBack(in?\,, path?\,, fn?_\delta\,, merge\_) \iff \\ &\qquad\quad fn?_\delta\,(getIn(in?\,, path?\,)) = fnSeed? = with?\end{aligned}$$

The following examples demonstrate the functionality of $walkBack$

$$\begin{aligned}&walkBack(X, \langle 0\rangle, array?\,\_\,, merge\_) = \langle false, x_1, x_2\rangle \\ &walkBack(X, \langle 2, qux\rangle, fn\_\,, merge\_) = \langle x_0, x_1, (x_2 \cup qux \mapsto ZZZ)\rangle \\ &walkBack(X, \langle 1, 0\rangle, succ\_, merge\_) = \langle x_0, \langle b, b, c\rangle, x_2\rangle\end{aligned}$$

8.3 Summary

The following is a summary of the general process which has been described in the previous sections. The variable names here are NOT intended to be 1:1 with those in the formal definitions (but there is some overlap) and the summary utilizes the Traversal Operations defined at the start of the section.

1. navigate down into the provided value $in?$ up until the second to last value $in?_{path?_{j-1}}$ as described by the provided $path?$

$$\begin{array}{l} in?_{path?_{j-1}} : V \\ \hline path?_{j-1} \Rightarrow path? \lhd j \Rightarrow path? \lhd (\operatorname{dom} path? \setminus \{j\}) \end{array}$$

2. extract any existing data mapped to $atIndex(path?\,, j)$ from the result of step 1

$$\begin{array}{l} in?_{path?} : V \\ \hline path? \Rightarrow path?_{j-1} \cup (j, atIndex(path?\,, j)) \end{array}$$

3. create the mapping $(atIndex(path?\,, j), in?_{path?}\,)$ labeled here as $args?$

$$\begin{array}{l} args? = (atIndex(path?\,, j), in?_{path?}\,) \\ \hline args? \in in?_{path?_{j-1}} \\ first(args?\,) = atIndex(path?\,, j) \end{array}$$

4. pass $in?_{path?}$ to the provided function $fn?$ to produce some output $fn!$

$$fn! = fn?\,(second(args?\,)) = fn?\,(in?_{path?}\,)$$

5. replace the previous mapping $args?$ within $in?_{path?_{j-1}}$ with $fn!$ at $atIndex(path?\,, j)$

$$\begin{array}{l} child_j = first(args?\,) \mapsto fn! \\ in!\,?_{path?_{j-1}} = merge((in?_{path_{j-1}}\,, fn!\,), first(args?\,)) \\ \hline \vdash \\ child_j \in in!\,?_{path?_{j-1}} \\ child_j \notin in?_{path?_{j-1}} \iff child_j \neq args? \\ args? \in in?_{path?_{j-1}} \\ args? \notin in!\,?_{path?_{j-1}} \iff args? \neq child_j \end{array}$$

6. retrace navigation back up from $in!\,?_{path?_{j-1}}$, updating the mapping at each $path?_n \in path?$ without touching any other mappings.

$$\begin{array}{l} in!\,?_{path?_{j-1}} \lhd first(args?\,) = in?_{path?_{j-1}} \lhd first(args?\,) \iff args? \neq child_j \\ \hline \vdash \\ args? \neq child_j \Rightarrow second(args?\,) \neq second(child_j) \\ in!\,?_{path?_{j-1}} \lhd first(args?\,) \Rightarrow in!\,?_{path?_{j-1}} \lhd (\,\mathrm{dom}\ in!\,?_{path?_{j-1}} \setminus first(args?\,)) \end{array}$$

7. return $out!$ after the final update is made to $in?$.

$$\begin{array}{l} child_i = atIndex(path?\,, i) \mapsto in!\,?_{path?_i} \\ in!\,?_{path?_i} = merge((in?_{path?_i}\,, in!\,?_{path?_{i+1}}\,), atIndex(path?\,, i+1)) \\ \hline \vdash \\ out! = merge((in?\,, second(child_i)), first(child_i)) \bullet \\ \quad in? \lhd head(path?\,) \ = \ out! \lhd head(path?\,) \Rightarrow \\ \quad\quad \forall (a,b) \in path? \bullet b = atIndex(path?\,, a) \mid \exists\, a \bullet in?_a = out!_a \iff a \neq head(path?\,) \end{array}$$

8.4 Replace At, Append At and Update At

In the summary of $walkBack$ above, the update at the target location within $in?$ takes place at step 4. The result of step 4, $fn!$, will overwrite the mapping $args$ such that $fn!$ replaces $in?_{path?}$ due to $fn?_{nav} = merge_{-}$. This results in the replacement of one mapping at each level of nesting such that the overall structure, composition and size of $out!$ is comparable to $in?$ unless $fn?_{\delta}$ dictates otherwise. While the functionality of fn_{nav} has been constrained here to always be an overwriting process, the same constraint is not placed on $fn?_{\delta}$.

8.4.1 Replace At

The Primitive $replaceAt$ was first defined in terms of the Traversal Operations and then served as the starting point for abstracting away aspects of functionality and delegating their responsibility to some passed in function until $WalkBack$ was reached. An alternate form of this formal definition is presented below such that $replaceAt$ is defined in terms of $walkBack$.

$$
\begin{array}{|l}
\hline
\textit{ReplaceAt}[V, Collection, V] \\
WalkBack, Merge \\
in?\,, with?\,, out!\,, fn!_\delta : V \\
path? : Collection \\
fn_\delta : V \nrightarrow V \\
replaceAt\_ : V \times Collection \times V \rightarrowtail V \\
\hline
replaceAt = \langle walkBack\_ \rangle \\
\\
out! = replaceAt(in?\,, path?\,, with?\,) = walkBack(in?\,, path?\,, fn_\delta, merge\_) \bullet \\
\quad let\ \ fn!_\delta == fn_\delta(getIn(in?\,, path?\,)) = with? \Rightarrow \\
\qquad walkBack(in?\,, path?\,, fn_\delta, merge\_) \equiv \\
\qquad\quad backProp(in?\,, path?\,, fn!_\delta\,, merge\_) \equiv \\
\qquad\qquad backProp(in?\,, path?\,, with?\,, merge\_) \\
\hline
\end{array}
$$

- fn_δ is defined within *ReplaceAt* as it performs a very simple task; ignore $getIn(in?\,, path?\,)$ and return $with?$
- Here, fn_δ represents one of the main general categories of update; replacement of a value such that the result of the replacement is in no way dependent upon the thing being replaced.

The following examples were pulled from the section containing the first version of *ReplaceAt* as they still hold true.

$$
\begin{aligned}
&replaceAt(X, \langle 2, foo, q \rangle, fn!\,) = \langle x_0, x_1, \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto y, q \mapsto ZZZ \rangle\!\rangle \rangle\!\rangle \rangle \\
&replaceAt(X, \langle 2, foo, x \rangle, fn!\,) = \langle x_0, x_1, \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto ZZZ \rangle\!\rangle \rangle\!\rangle \rangle
\end{aligned}
$$

8.4.2 Append At

In order to define the Primitive *appendAt*, the Traversal Operation *conj* is used. In order to demonstrate the usage of *conj* as $fn?_\delta$ of *walkBack*, a syntax not yet formally defined in this document is defined. It is an extension of the shorthand $val_{index} = get(Val, index)$ as seen in examples like

$$
\begin{aligned}
&conj(x_0, false) = \langle true, false \rangle = \langle x_0, false \rangle \\
&conj(X, X) = \langle x_0, x_1, x_2, \langle x_0, x_1, x_2 \rangle \rangle
\end{aligned}
$$

The following expands that usage to describe following some $path?$ into a $Collection$ or KV.

$$
\begin{array}{|l}
X_{path?} = getIn(X, path?\,) \\
\hline
X_{\langle 1 \rangle} = x_1 = \langle a, b, c \rangle \\
X_{\langle 1,0 \rangle} = a
\end{array}
$$

This syntax is used for the placeholder $X_{path?}$ so that the role of $fn?_\delta$ can be demonstrated within the arguments passed to *walkBack*. This notation can be used to describe how arguments passed to a top level function get used within component functions without writing the equivalent Z schema. This shorthand can also be used within Z schemas.

$$
\begin{aligned}
&walkBack(X, \langle 1 \rangle, map\,\_(conj\,\_, X_{\langle 1 \rangle}, a), merge\,\_) = \langle x_0, \langle\langle a, a\rangle, \langle b, a\rangle, \langle c, a\rangle\rangle, x_2\rangle \\
&walkBack(X, \langle 1 \rangle, conj\,\_(X_{\langle 1 \rangle}, a), merge\,\_) = \langle x_0, \langle a, b, c, a\rangle, x_2\rangle
\end{aligned}
$$

Additive updates are another common type of updating encountered when working with xAPI data. *Conj* is a derivative of $\frown$ but scoped to DAVE and used to define the Primitive *appendAt*.

$$
\begin{array}{|l}
\hline
AppendAt[V, Collection, V] \\
WalkBack, Conj, Merge \\
in?\,, toEnd?\,, out! : V \\
path? : Collection \\
appendAt\,\_ : V \times Collection \times V \rightarrowtail V \\
\hline
appendAt = \langle walkBack\,\_\rangle \\
\\
out! = appendAt(in?\,, path?\,, toEnd?\,) \equiv \\
\qquad walkBack(in?\,, path?\,, conj\,\_(in?_{path?}\,, toEnd?\,)\,, merge\,\_) \Rightarrow \\
\qquad\quad backProp(in?\,, path?\,, fn!_{\delta}\,, merge\,\_) \iff \\
\qquad\qquad fn!_{\delta} = fn?_{\delta}\,(in?_{path?}\,, toEnd?\,) \bullet \\
\qquad\qquad\quad fn?_{\delta}\,\_(in?_{path?}\,, toEnd?\,) = fn?_{\delta} \leftrightarrow (in?_{path?}\,, toEnd?\,) \bullet \\
\qquad\qquad\quad conj\,\_(in?_{path?}\,, toEnd?\,) = conj\,\_ \leftrightarrow (in?_{path?}\,, toEnd?\,) \Rightarrow \\
\qquad\qquad\qquad (fn?_{\delta} = conj\,\_) \land \\
\qquad\qquad\qquad (fn!_{\delta} \neq conj\,\_(in?_{path?}\,, toEnd?\,)) \land \\
\qquad\qquad\qquad (fn!_{\delta} = conj(in?_{path?}\,, toEnd?\,)) \\
\hline
\end{array}
$$

This schema features a new notation which highlights evaluation nuances.

- $fn?_{\delta}$ is used to represent the function itself
- $fn?_{\delta}\,\_(in?_{path?}\,, toEnd?\,)$ is used to represent the relationship between the function and the arguments it WILL be passed
- $fn!_{\delta} \equiv fn?_{\delta}\,(in?_{path?}\,, toEnd?\,)$ is used to represent the output of $fn?_{\delta}$ given the passed in arguments

Such that the following are all equivalent expressions.

$$
\begin{aligned}
&appendAt(in?\,, path?\,, toEnd?\,) \equiv \\
&\quad walkBack(in?\,, path?\,, fn?_{\delta}\,, merge\,\_) \equiv \\
&\quad walkBack(in?\,, path?\,, conj\,\_(in?_{path?}\,, toEnd?\,), merge\,\_) \equiv \\
&\quad walkBack(in?\,, path?\,, fn?_{\delta}\,\_(in?_{path?}\,, toEnd?\,), merge\,\_) \equiv \\
&\qquad backProp(in?\,, path?\,, fn!_{\delta}\,, merge\,\_) \equiv \\
&\qquad backProp(in?\,, path?\,, conj(in?_{path?}\,, toEnd?\,), merge\,\_)
\end{aligned}
$$

The following example demonstrates this usage.

$$walkBack(X, \langle 1\rangle, map\_(append\_, X_{\langle 1\rangle}, a), merge\_) = \langle x_0, \langle\langle a, a\rangle, \langle b, a\rangle, \langle c, a\rangle\rangle, x_2\rangle$$

$$map\_(append\_, X_{\langle 1\rangle}, a) \equiv map\_(append\_(X_{\langle 1,n\rangle}, a), X_{\langle 1\rangle}, a) \bullet n \in \operatorname{dom} X_{\langle 1\rangle}$$

The following examples demonstrate the functionality of *appendAt*.

$$appendAt(X, \langle 1\rangle, e) = \langle x_0, \langle a, b, c, e\rangle, x_2\rangle$$
$$appendAt(X, \langle 2\rangle, \langle 1, 2, 3\rangle) = \langle x_0, x_1, \langle x_2, \langle 1, 2, 3\rangle\rangle\rangle$$
$$appendAt(X, \langle 0\rangle, bar) = \langle\langle x_0, bar\rangle, x_1, x_2\rangle$$

8.4.3 Update At

The Primitive *updateAt* does not make any assumptions about how the relationship between $getIn(in?, path?)$ and $fn!_\delta$ is established. This makes it possible to define both *replaceAt* and *appendAt* using *updateAt*.

$UpdateAt[V, Collection, (\_ \nrightarrow \_)]$

$WalkBack, Merge$
$in?, out! : V$
$path? : Collection$
$fn?_\delta : (\_ \nrightarrow \_)$
$updateAt\_ : V \times Collection \times (\_ \nrightarrow \_) \rightarrowtail V$

$updateAt = \langle walkBack\_\rangle$

$$out! = updateAt(in?, path?, fn?_\delta) = walkBack(in?, path?, fn?_\delta, merge\_) \Rightarrow backProp(in?, path?, fn!_\delta, merge\_)$$

- The item found at the target path $getIn(in?, path?)$ is passed to $fn?_\delta$ such that the calculation of the replacement $fn!_\delta$ CAN be dependent upon $getIn(in?, path?)$.

The following examples demonstrate the functionality of the Primitive *updateAt*

$$updateAt(X, \langle 0\rangle, array?\ \_) = \langle false, x_1, x_2\rangle$$
$$updateAt(X, \langle 1, 0\rangle, fn?_\delta\ \_(X_{\langle 1,0\rangle})) = \langle x_0, \langle z, b, c\rangle, x_2\rangle \iff fn?_\delta\,(X_{\langle 1,0\rangle}) = z$$

and the following shows how *updateAt* can be used to define *appendAt*

$$appendAt(in?, path?, toEnd?) \equiv updateAt(in?, path?, conj\_(in?_{path?}, toEnd?))$$

and *replaceAt*.

$$replaceAt(in?, path?, with?) \equiv updateAt(in?, path?, merge\_((in?_{path?_{j-1}}, with?), \langle path?_j\rangle))$$

9 Rate of Completions

As learners engage in activities supported by a learning ecosystem, they will build up a history of learning experiences. When the digital resources of that learning ecosystem adhere to a framework dedicated to supporting and understanding the learner, such as the Total Learning Architecture (TLA), the data produced by the learning ecosystem will contribute to each learner's digital footprint. One way that footprint can be made actionable is through analysis of trends and/or patterns of activity. The following Algorithm does exactly this but scoped to:

- events describing or asserting that a learner completed a learning activity or exercise.
- events which happened within some target window of time

9.1 Alignment to DAVE Algorithm Definition

The schema $RateOfCompletions$ serves as the first formal definition of an Algorithm which implements the definition of a DAVE Algorithm presented in the section Algorithm Formal Definition(6.6) on page 32. $RateOfCompletions$ is used to introduce the alignment between the generic components of $Algorithm$ and their corresponding definitions within this domain specific use case. In general, all DAVE Algorithm definitions must reference the schema $Algorithm$ and the schemas corresponding to the different components of $Algorithm$. Within $RateOfCompletions$, both $Algorithm.algorithm.algorithmIter$ and $ROCalgorithmIter$ are fully expanded for clarity. This is not a requirement of alignment schemas, but alignment schemas should feature:

- an expanded definition of the use case specific $algorithmIter$
- binding of the use case specific $algorithmIter$ to $Algorithm.algorithm.algorithmIter$

Typically, an alignment schema would be defined after its component schemas but because $RateOfCompletions$ is the first of its kind, it is featured first to introduce the notation by example and set the stage for the following component definitions. The alignments established in $RateOfCompletions$ are further expanded upon within the corresponding definition of each individual component.

9.1.1 Components

Within each component definition, in order to connect the dots between

- $Algorithm$ and its components
- $RateOfCompletions$ and its components

the symbol $\rightsquigarrow$ is used. This establishes that the constraints defined in the more generic component formal definitions apply to the schema being binded to. This is formalized within each of the *RateOfCompletions* component schemas via

$$
\begin{array}{|l}
genericSchema.primitiveName = \langle body \rangle \\
\hline
\langle body \rangle \rightsquigarrow localSchema.primitiveName = localSchema.primitiveChain
\end{array}
$$

9.2 Formal Definition

The application of the notation described above to *RateOfCompletions* results in the following definition with respect to schemas

$$
\begin{aligned}
&RateOfCompletions ::= \\
&\qquad Algorithm \fatsemi RateOfCompletions \Rightarrow \\
&\qquad\qquad (Init \fatsemi RateOfCompletionsInit) \land \\
&\qquad\qquad (Relevant? \fatsemi RateOfCompletionsRelevant?) \land \\
&\qquad\qquad (Accept? \fatsemi RateOfCompletionsAccept?) \land \\
&\qquad\qquad (Step \fatsemi RateOfCompletionsStep) \land \\
&\qquad\qquad (Result \fatsemi RateOfCompletionsResult)
\end{aligned}
$$

such that the $\langle body \rangle$ within each of the generic schema definitions is substituted for the Primitive chain defined within each of the local schemas. Here, the components of *RateOfCompletions* use a naming scheme of Container + AlgorithmComponent but this pattern is not required. It is used here strictly for additional highlighting of the syntax introduced above for connecting the generic definition of an Algorithm to an Implementation of that methodology much like the concepts underlying Java Interfaces.

$$
\begin{array}{l}
\Delta RateOfCompletions[KV, Collection, KV] \\
\hline
Algorithm \\
RateOfCompletionsInit \\
RateOfCompletionsRelevant? \\
RateOfCompletionsAccept? \\
RateOfCompletionsStep \\
RateOfCompletionsResult \\
rateOfCompletions\_ : KV \times Collection \times KV \twoheadrightarrow KV \\
state?, opt?, state! : KV \\
S? : Collection \\
\hline
Algorithm.algorithm.algorithmIter = \langle relevant?\ \_, accept?\ \_, step\ \_\rangle \\
ROCalgorithmIter = \langle RateOfCompletionsRelevant?.relevant?\ \_, \\
\qquad\qquad RateOfCompletionsAccept?.accept?\ \_, \\
\qquad\qquad RateOfCompletionsStep.step\ \_\rangle \\
\\
Algorithm.algorithm.algorithmIter\ \_ \leadsto ROCalgorithmIter\ \_ \Rightarrow \\
\quad (Algorithm.algorithm.algorithmIter.relevant?\ \_ \leadsto \\
\qquad\quad RateOfCompletionsRelevant?.relevant?\ \_) \land \\
\quad (Algorithm.algorithm.algorithmIter.accept?\ \_ \leadsto \\
\qquad\quad RateOfCompletionsAccept?.accept?\ \_) \land \\
\quad (Algorithm.algorithm.algorithmIter.step\ \_ \leadsto \\
\qquad\quad RateOfCompletionsStep.step\ \_) \\
\\
state! = rateOfCompletions(state?, S?, opt?) \equiv algorithm(state?, S?, opt?) \iff \\
\qquad (Algorithm.algorithm.init\ \_ \leadsto RateOfCompletionsInit.init\ \_) \land \\
\qquad (Algorithm.algorithm.algorithmIter\ \_ \leadsto ROCalgorithmIter\ \_) \land \\
\qquad (Algorithm.algorithm.result\ \_ \leadsto RateOfCompletionsResult.result\ \_) \\
\hline
\end{array}
$$

- the . notation is used to reference components within a schema
- the $\leadsto$ represents alignment between components of $Algorithm$ and $RateOfCompletions$
- the Δ in the schema name indicates that $RateOfCompletions$ alters the state space of $Algorithm$ due to usage of $\leadsto$

9.3 Initialization

The first example of a component to component alignment is found within $RateOfCompletionsInit$ which shows how the primitive $RateOfCompletionsInit.init$ is bound to $\langle body\rangle$ within $Algorithm.algorithm.init$. Specifically, the schema $RateOfCompletionsInit$ uses the Primitive $updateAt$ such that $init_\delta$ can be used to establish the initialization logic.

9.3.1 Formal Definition

In the following, $init_\delta$ could have been a stand alone Operation referenced within $RateOfCompletionsInit$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad RateOfCompletionsInit[KV] \quad}} \\
Init, UpdateAt \\
state?, state! : KV \\
init\_ : KV \twoheadrightarrow KV \\
init_\delta : V \twoheadrightarrow KV \\
\hline
Init.init = \langle body \rangle \\
init = \langle updateAt\_ \rangle \\
Init.init \leadsto init \Rightarrow \langle body \rangle \equiv \langle updateAt\_ \rangle \\
\\
init_\delta ! = init_\delta(state?_{\langle roc, completions \rangle}) \bullet \\
\qquad = (\emptyset \iff \langle roc, completions \rangle \notin state?) \vee \\
\qquad\quad (state?_{\langle roc, completions \rangle} \iff \langle roc, completions \rangle \in state?) \\
\\
state! = init(state?) = updateAt(state?, \langle roc, completions \rangle, init_\delta) \bullet \\
\qquad = (\langle\!\langle roc \mapsto completions \mapsto \emptyset \rangle\!\rangle \cup state? \iff init_\delta ! = \emptyset) \vee \\
\qquad\quad (state? \iff init_\delta ! \neq \emptyset) \\
\hline
\end{array}
$$

The output of $RateOfCompletionsInit.init$ is $state!$ which can be one of two things given the definition of $init_\delta$

- $state! = \langle\!\langle roc \mapsto completions \mapsto \emptyset \rangle\!\rangle \cup state?$
- $state! = state?$

This means that the result of any previous runs of $rateOfCompletions$ will not be overwritten but if this is the first iteration of the Algorithm, the necessary storage location is established within the Algorithm State such that

- $RateOfCompletionsStep.step$ can write its output to $state!_{\langle roc, completions \rangle}$
- $RateOfCompletionsResult.result$ can read from $state!_{\langle roc, completions \rangle}$

and by defining $RateOfCompletionsInit.init$ in this way, it allows for chaining of calls to $rateOfCompletion$ such that

- the Algorithm can pick back up from the result of a previous iteration
- Other Algorithms can use the result of $rateOfCompletions$ within their processing

which highlights the importance of establishing a unique path for individual Algorithms to write their results to. The example $path?$ of $\langle roc, completions \rangle$ is very simple but is sufficient for the current Algorithm. This $path?$ can be made more complex to support more advanced $init_\delta$ definitions. For example, each run of $rateOfCompletions$ could have its own unique subpath. In this scenario, $init_\delta$ could be updated to look for the most recent run of $rateOfCompletions$ and use it as the seed state for the current iteration among other things.

- $\langle roc, completions, run1 \rangle$
- $\langle roc, completions, run2 \rangle$

9.3.2 Big Picture

When Algorithms write to a unique location within an Algorithm State, high level Algorithms can be designed which chain together individual Algorithms such that the result of one is used to seed the next. Chaining together of Algorithms is a subject not yet covered within this report and its exact form is still under active development. It is mentioned here to highlight the ideal usage of Algorithm State in the context of *init*; Algorithm State is a mutable Map which serves as a storage location for a collection of Algorithm(s) to write to and/or read from such that an Algorithm can

- pick up from a previous iteration
- use the output of other Algorithm(s) to initialize the current state
- process quantities of data too large to store in local memory all at once

9.4 Relevant?

Given that the purpose of *relevant*? is to determine if the current Statement (*stmt*?) is valid for use within *step* of $rateOfCompletions$, the validation check itself can be implemented in several different ways but ideally, the predicate logic is expressed using the xAPI Profiles spec.

9.4.1 xAPI Profile Validation

The specification defines xAPI Statement Templates which feature a built in xAPI property predicate language for defining the uniquely identifying properties of an xAPI Statement. These requirements are used within validation logic aligned to/based off of the Statement Template Validation Logic defined in the spec. The formal definition of Statement Template validation logic is outside the scope of this document but the following basic type is introduced to represent an xAPI Statement Template

$$[TEMPLATE_{stmt}]$$

such that the following is an Operation definition for validation of an xAPI Statement *stmt*? against an xAPI Statement Template.

$$
\begin{array}{|l}
\hline
ValidateStatement[STATEMENT, TEMPLATE_{stmt}] \\
stmt? : STATEMENT \\
template? : TEMPLATE_{stmt} \\
validateStmt! : Boolean \\
validateStmt\_ : STATEMENT \times TEMPLATE_{stmt} \rightarrow Boolean \\
\hline
validateStmt! = validateStmt(stmt?, template?) = true \vee false \\
\hline
\end{array}
$$

This Operation can be composed with other xAPI Profile centered Operations to define more complex predicate/validation logic like:

- $stmt?$ matches target xAPI Statement Template(s) defined within some xAPI Profile(s)
- $stmt?$ matches pred (ie, any/none/etc.) xAPI Statement Template(s) defined within some xAPI Profile(s)
- $stmt?$ matches target/pred xAPI Statement Template(s) within target/pred xAPI Pattern(s) defined within some xAPI Profile(s)

9.4.2 xAPI Predicates

In order to avoid brining in additional xAPI Profile complexity, the logic of $RateOfCompletionsRelevant?$ is implemented using predicates which correspond to checks which would happen during $validateStmt$ given Statement Templates containing the following constraints.

- is the Object of the Statement an Activity?
- is the Verb indicative of a completion event?
- is Result.completion used to indicate completion?

In general, each of these Primitives navigates into a Statement to retrieve the value at a target $path?$ and check it against the predicate defined in the schema. This generic functionality is defined as the Primitive $stmtPred$.

$$
\begin{array}{l}
\textbf{StatementPredicate}[STATEMENT, Collection, (\_ \rightarrow\!\!\!\!\!\!+ \_)] \\
\hline
GetIn \\
stmt? : STATEMENT \\
path? : Collection \\
fn_{pred}! : Boolean \\
fn_{pred}? : (\_ \rightarrow\!\!\!\!\!\!+ \_) \\
stmtPred\_ : STATEMENT \times Collection \times (\_ \rightarrow\!\!\!\!\!\!+ \_) \rightarrow Boolean \\
\hline
stmtPred = \langle getIn\_, fn_{pred}?\,(stmt?_{path?}) \rangle \\
\\
fn_{pred}! = stmtPred(stmt?, path?, fn_{pred?}) \\
\qquad = fn_{pred}?\,(getIn(stmt?, path?)) \bullet \\
\qquad = true \lor false \\
\hline
\end{array}
$$

This Primitive covers the most basic kind of check performed when validating an xAPI Statement against an xAPI Statement Template; does the Statement property found at $stmt?_{path?}$ adhere to the expectation(s) defined within the provided predicate. The next three schemas will define the statement predicates used within $RateOfCompletionsRelevant?$ but these predicates could have been contained within some number of xAPI Statement Template(s).

$$
\begin{array}{l}
\textit{ActivityObject?}\,[\textit{STATEMENT}] \\
\hline
\textit{StatementPredicate} \\
\textit{stmt?} : \textit{STATEMENT} \\
\textit{path?} : \textit{Collection} \\
fn_{pred}! : \textit{Boolean} \\
fn_{pred}\_ : V \rightarrow \textit{Boolean} \\
\textit{activityObject?}\,\_ : \textit{STATEMENT} \rightarrow \textit{Boolean} \\
\hline
\textit{activityObject?} = \langle \textit{stmtPred}\,\_ \rangle \\
\\
\textit{path?} = \langle \textit{object}, \textit{objectType} \rangle \\
\\
fn_{pred}! = \textit{activityObject?}\,(\textit{stmt?}) \\
\qquad = \textit{stmtPred}(\textit{stmt?}, \textit{path?}, fn_{pred}) \\
\qquad = fn_{pred}(\textit{stmt?}_{\textit{path?}}) \\
\qquad = \textit{true} \iff \textit{stmt?}_{\textit{path?}} = \textit{Activity} \vee \emptyset
\end{array}
$$

- Determine if the Object of *stmt*? is an Activity

$$
\begin{array}{l}
\textit{CompletionVerb?}\,[\textit{STATEMENT}] \\
\hline
\textit{StatementPredicate} \\
\textit{stmt?} : \textit{STATEMENT} \\
\textit{path?} : \textit{Collection} \\
fn_{pred}! : \textit{Boolean} \\
fn_{pred}\_ : V \rightarrow \textit{Boolean} \\
\textit{completionVerb?}\,\_ : \textit{STATEMENT} \rightarrow \textit{Boolean} \\
\hline
\textit{completionVerb?} = \langle \textit{stmtPred}\,\_ \rangle \\
\\
\textit{path?} = \langle \textit{verb}, \textit{id} \rangle \\
\\
fn_{pred}! = \textit{completionVerb?}\,(\textit{stmt?}) \\
\qquad = \textit{stmtPred}(\textit{stmt?}, \textit{path?}, fn_{pred}) \\
\qquad = fn_{pred}(\textit{stmt?}_{\textit{path?}}) \\
\qquad = \textit{true} \iff \textit{stmt?}_{\textit{path?}} = \\
\qquad\qquad http : //adlnet.gov/expapi/verbs/passed \vee \\
\qquad\qquad https : //w3id.org/xapi/dod-isd/verbs/answered \vee \\
\qquad\qquad http : //adlnet.gov/expapi/verbs/completed
\end{array}
$$

- Determine if the Verb id within *stmt*? is one of
  - passed
  - answered
  - completed
- List of target Verb ids can be expanded as needed

$$
\begin{array}{|l}
\hline
\textit{CompletionResult?}\,[STATEMENT] \\
\textit{StatementPredicate} \\
stmt? : STATEMENT \\
path? : \textit{Collection} \\
fn_{pred}! : \textit{Boolean} \\
fn_{pred\,\_} : V \rightarrow \textit{Boolean} \\
\textit{completionResult?}\ \_ : STATEMENT \rightarrow \textit{Boolean} \\
\hline
\textit{completionResult?} = \langle stmtPred\_ \rangle \\
\\
path? = \langle result, completion \rangle \\
\\
fn_{pred}! = \textit{completionResult?}\,(stmt?) \\
\qquad = stmtPred(stmt?, path?, fn_{pred}) \\
\qquad = fn_{pred}(stmt?_{path?}) \\
\qquad = true \iff stmt?_{path?} = true \\
\hline
\end{array}
$$

- Determine if completion is set to true within result field of an xAPI Statement

9.4.3 Formal Definition

The xAPI Predicates defined above are used within $RateOfCompletionsRelevant?$ to establish the logic which decides if $stmt?$ is

- passed on to the next step
- discarded for the next Statement in the batch passed to $rateOfCompletions$

$$
\begin{array}{|l}
\hline
\Xi \textit{RateOfCompletionsRelevant?}\,[KV, STATEMENT] \\
\textit{Relevant?} \\
state? : KV \\
stmt? : STATEMENT \\
relevant! : \textit{Boolean} \\
relevant?\ \_ : KV \times STATEMENT \rightarrow \textit{Boolean} \\
\hline
\textit{Relevant.relevant?} = \langle body \rangle \\
\langle body \rangle \rightsquigarrow relevant? = \langle\, activityObject?\ \_, \langle\, completionVerb?\ \_,\ completionResult?\ \_ \rangle\rangle \\
\\
relevant! = relevant?\,(state?, stmt?) \\
\qquad = true \iff (activityObject(stmt?) = true) \land \\
\qquad\qquad\qquad ((completionVerb?\,(stmt?) = true) \lor \\
\qquad\qquad\qquad (completionResult?\,(stmt?) = true)) \\
\hline
\end{array}
$$

The schema prefix Ξ is used to indicate that here, $relevant?$ does not modify $state?$. Regardless, in order for $relevant?$ to return true

- The object of *stmt*? must be an activity
- The Verb of *stmt*? has an id which matches one of the target IDs
- The Result of *stmt* indicates that a completion happened

9.5 Accept?

The *Accept*? component of a DAVE Algorithm is a secondary validation check prior to the potential passing of *stmt*? off to *Step*. At this point, *stmt*? has been validated to be relevant to the execution of an Algorithm so the final check is based off of the current Algorithm State *state*?. In many cases this check will not be necessary but this step matters when the ability to process *stmt*? is dependent upon some property of *state*?. This component of an Algorithm could be used to establish the placeholder mapping within *state*! if it doesn't exist for the current *stmt*? but this can also be handled within *step* as done in the schema *ProcessCompletionStatement* defined later on.

$$
\begin{array}{l}
\Xi RateOfCompletionsAccept?\,[KV, STATEMENT] \\
\hline
Accept? \\
state? : KV \\
stmt? : STATEMENT \\
accept! : Boolean \\
fn_{pred} : KV \times STATEMENT \rightarrow Boolean \\
accept?\ \_ : KV \times STATEMENT \rightarrow Boolean \\
\hline
Accept?\,.accept? = \langle body \rangle \\
accept? = \langle fn_{pred}\ \_ \rangle \\
Accept?\,.accept? \leadsto accept? \Rightarrow \langle body \rangle \equiv \langle fn_{pred}\ \_ \rangle \\
\\
accept! = accept?\,(state?\,, stmt?\,) \\
\qquad\quad = fn_{pred}(state?\,, stmt?\,) = true
\end{array}
$$

The Algorithm *rateOfCompletions* does not need to check *state*? before passing *stmt*? to *step* so fn_{pred} will always return true. If this was not the case, fn_{pred} would be defined as a predicate describing the relationship between *state*? and *stmt*? which determines if true or false is returned. Additionally, if false would be returned, *Accept* can take the appropriate steps to ensure *state*! is updated such that $accept?\,(state!\,, stmt?\,) = true$.

9.6 Step

The actual processing of *stmt*? happens within *step* and may or may not result in an updated Algorithm State *state*!. In the case of *rateOfCompletions*, each call to *step* is expected to return an altered state such that $state! \neq state?$ and the schema *RateOfCompletionsStep* is prefixed with Δ accordingly. The updated *state*! will either have an existing mapping for $objectId \in state?$ altered or a completely new mapping for *objectId* added to *state*?.

9.6.1 Processing Summary

The execution of *step* can be summarized as:

1. parse the relevant information from *stmt*
   - $currentTime$
   - $objectName$
   - $objectId$
2. resolve previous state (if it exists) given $objectId$
3. update the range of time to include $currentTime$ if not already within the existing interval for $objectId$
4. update the counter tracking the number of times $objectId$ has been in a $stmt?$ passed to *step*
5. add $objectName$ to the set of names associated with $objectId$ if not already a member.

9.6.2 Helper Functions

The following Operations and Primitives are defined for abstracting the functionality of each process within *step* in order to reduce the noise within $RateOfCompletionsStep$.

$$
\begin{array}{l}
\textbf{schema } ParseCompletionStatement[STATEMENT] \\
\quad GetIn \\
\quad stmt? : STATEMENT \\
\quad currentTime : TIMESTAMP \\
\quad objectName, parseStmt! : KV \\
\quad objectId : STRING \\
\quad parseStmt\_ : STATEMENT \rightarrowtail KV \\
\hline
\quad parseStmt = \langle getIn\_,\ associate\_\rangle^2 \\
\\
\quad currentTime = getIn(stmt?, \langle timestamp \rangle) \\
\quad objectName = getIn(stmt?, \langle object, definition, name \rangle) \\
\quad objectId = getIn(stmt?, \langle object, id \rangle) \\
\\
\quad parseStmt! = parseStmt(stmt?) \bullet \\
\qquad let\ \ withTime == associate(\langle\!\langle\rangle\!\rangle, currentT, currentTime) \\
\qquad\qquad withName == associate(withTime, objName, objectName) \\
\qquad\qquad\quad = associate(withName, objId, objectId) \Rightarrow \\
\qquad \langle\!\langle currentT \mapsto currentTime,\ objName \mapsto objectName,\ objId \mapsto objectId \rangle\!\rangle
\end{array}
$$

- parse timestamp, object name and object id from $stmt?$

$$
\begin{array}{|l}
\multicolumn{1}{l}{ResolvePreviousCompletionState[KV, KV]} \\
GetIn \\
state?, parsed?, prevState! : KV \\
getPreviousState\_ : KV \times KV \nrightarrow KV \\
\hline
getPreviousState = \langle getIn\_, getIn\_ \rangle \\
objectId = getIn(parsed?, objId) \\
prevState! = getPreviousState(state?, parsed?) = getIn(state?, \langle roc, completions, objectId \rangle) \\
\hline
\end{array}
$$

- look in $state?$ for any previous record of $objectId$

$$
\begin{array}{|l}
\multicolumn{1}{l}{IntervalValGiven[TIMESTAMP, TIMESTAMP(\_ \nrightarrow \_)]} \\
IsoToUnix \\
stmt_{ts}, state_{ts}, intervalValGiven! : TIMESTAMP \\
fn_{pred} : (\_ \nrightarrow \_) \\
fn_{pred}! : \mathbb{N} \\
intervalValGiven\_ : TIMESTAMP \times TIMESTAMP \times (\_ \nrightarrow \_) \nrightarrow TIMESTAMP \\
\hline
intervalValGiven = \langle isoToUnix\_, isoToUnix\_, fn_{pred}\_ \rangle \\
\\
nSeconds_{stmt} = isoToUnix(stmt_{ts}) \\
nSeconds_{state} = isoToUnix(state_{ts}) \\
fn_{pred}! = fn_{pred}(nSeconds_{stmt}, nSeconds_{state}) \\
intervalValGiven! = intervalValGiven(stmt_{ts}, state_{ts}, fn_{pred}) \\
\qquad = (stmt_{ts} \iff fn_{pred}! = nSeconds_{stmt}) \lor \\
\qquad\quad (state_{ts} \iff fn_{pred}! = nSeconds_{state}) \\
\hline
\end{array}
$$

- return $stmt_{ts}$ or $state_{ts}$ based on result of fn_{pred}

$$
\begin{array}{|l}
\multicolumn{1}{l}{ReturnIntervalStart[TIMESTAMP, TIMESTAMP]} \\
IntervalValGiven \\
stmt_{ts}, state_{ts}, interval_{start} : TIMESTAMP \\
fn_{\delta}! : \mathbb{N} \\
fn_{\delta} : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N} \\
returnIntervalStart\_ : TIMESTAMP \times TIMESTAMP \nrightarrow TIMESTAMP \\
\hline
returnIntervalStart = \langle intervalValGiven\_ \rangle \\
\\
fn_{\delta}! = fn_{\delta}(nSeconds_{stmt}, nSeconds_{state}) \\
\quad = (nSeconds_{stmt} \iff nSeconds_{stmt} \leq nSeconds_{state}) \lor \\
\quad\quad (nSeconds_{state} \iff nSeconds_{stmt} > nSeconds_{state}) \\
interval_{start} = intervalValGiven(stmt_{ts}, state_{ts}, fn_{\delta}) \\
\qquad = (stmt_{ts} \iff fn_{\delta}! = nSeconds_{stmt}) \lor \\
\qquad\quad (state_{ts} \iff fn_{\delta}! = nSeconds_{state}) \\
\hline
\end{array}
$$

- return $stmt_{ts}$ or $state_{ts}$, whichever one is further back in the past.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ ReturnIntervalEnd[TIMESTAMP, TIMESTAMP]\ \_\_\_\_\_\_\_\_} \\
IntervalValGiven \\
stmt_{ts}, state_{ts}, interval_{end} : TIMESTAMP \\
fn_{\delta}! : \mathbb{N} \\
fn_{\delta} : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N} \\
returnIntervalEnd\_ : TIMESTAMP \times TIMESTAMP \nrightarrow TIMESTAMP \\
\hline
returnIntervalEnd = \langle intervalValGiven\_ \rangle \\
\\
fn_{\delta}! = fn_{\delta}(nSeconds_{stmt}, nSeconds_{state}) \\
\quad = (nSeconds_{stmt} \iff nSeconds_{stmt} \geq nSeconds_{state}) \lor \\
\quad\quad (nSeconds_{state} \iff nSeconds_{stmt} < nSeconds_{state}) \\
\\
interval_{end} = intervalValGiven(stmt_{ts}, state_{ts}, fn_{\delta}) \\
\quad\quad = (stmt_{ts} \iff fn_{\delta}! = nSeconds_{stmt}) \lor \\
\quad\quad\quad (state_{ts} \iff fn_{\delta}! = nSeconds_{state}) \\
\hline
\end{array}
$$

- return $stmt_{ts}$ or $state_{ts}$, whichever one is later on chronologically

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ ReturnUpdatedCount[V]\ \_\_\_\_\_\_\_\_} \\
count? : V \\
count! : \mathbb{N} \\
returnUpdatedCount\_ : V \rightarrow \mathbb{N} \\
\hline
count! = returnUpdatedCount(count?) \\
\quad = (count? + 1 \iff count? \neq \emptyset) \lor \\
\quad\quad (1 \iff (count? = 0) \lor (count? = \emptyset)) \\
\hline
\end{array}
$$

- return an incremented value or 1 otherwise

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ ReturnUpdatedNames[Collection, STRING]\ \_\_\_\_\_\_\_\_} \\
names?, names! : Collection \\
targetName : STRING \\
returnUpdatedNames\_ : Collection \times STRING \twoheadrightarrow Collection \\
\hline
names?_{targetName} = names? \upharpoonright targetName \\
names! = returnUpdatedNames(names?, targetName) \\
\quad = (names? \frown targetName \iff names?_{targetName} = \emptyset \Rightarrow targetName \notin names?) \lor \\
\quad\quad (names? \iff names?_{targetName} \neq emptyset \Rightarrow targetName \in names?) \\
\hline
\end{array}
$$

- add $targetName$ to the end of $names?$ if $targetName \notin names?$

9.6.3 Formal Definition

The schema $ProcessCompletionStatement$ is used to define the core functionality of $RateOfCompletionsStep.step$ using the Primitive $replaceAt$ to produce $state!$.

$$\Delta ProcessCompletionStatement[STATEMENT, KV]$$

$$\begin{array}{l}
ReplaceAt \\
ParseCompletionStatement \\
ResolvePreviousCompletionState \\
ReturnIntervalStart \\
ReturnIntervalEnd \\
ReturnUpdatedCount \\
ReturnUpdatedNames \\
stmt? : STATEMENT \\
state?, state!, state_{objectId} : KV \\
processStatement_ : STATEMENT \times KV \rightarrowtail KV
\end{array}$$

$$\begin{array}{l}
processStatement = \langle \langle parseStmt_,\ getPreviousState_ \rangle, \\
\qquad \langle returnIntervalStart_,\ returnIntervalEnd_,\ replaceAt_ \rangle, \\
\qquad \langle returnUpdatedCount_,\ replaceAt_ \rangle, \\
\qquad \langle returnUpdatedNames_,\ replaceAt_ \rangle \rangle \\
\\
parsed_{stmt?} = parseStmt(stmt?) \\
stmt_{timestamp} = get(parsed_{stmt?}, currentT) \\
stmt_{objName} = get(parsed_{stmt?}, objName) \\
stmt_{objId} = get(parsed_{stmt?}, objId) \\
\\
state_{objectId} = getPreviousState(state?, parsed_{stmt?}) \\
interval_{start} = getIn(state_{objectId}, \langle domain, start \rangle) \\
interval_{end} = getIn(state_{objectId}, \langle domain, end \rangle) \\
state_{nStmts} = get(state_{objectId}, nStmts) \\
state_{names} = get(state_{objectId}, names) \\
\\
interval_{start}! = returnIntervalStart(stmt_{timestamp}, interval_{start}) \\
interval_{end}! = returnIntervalEnd(stmt_{timestamp}, interval_{end}) \\
interval! = \langle\!\langle start \mapsto interval_{start}!, end \mapsto interval_{end}! \rangle\!\rangle \\
\\
nStmts! = returnUpdatedCount(state_{nStmts}) \\
names! = returnUpdatedNames(state_{names}, stmt_{objName}) \\
\\
state! = processStatement(stmt?, state?) \bullet \\
\quad let\ \ interval_{\delta} == replaceAt(state?, \langle roc, completions, stmt_{objId}, domain \rangle, interval!) \\
\qquad nStmts_{\delta} == replaceAt(interval_{\delta}, \langle roc, completions, stmt_{objId}, nStmts \rangle, nStmts!) \\
\qquad = replaceAt(nStmts_{\delta}, \langle roc, completions, stmt_{objId}, names \rangle, names!)
\end{array}$$

- update *state*! to include a mapping with Key $stmt_{objId}$ or update an existing mapping identified by $stmt_{objId}$

The schema *RateOfCompletionsStep* introduces the alignment with *Algorithm.step* such that $\langle body \rangle = processStatement$ as defined by *ProcessCompletionStatement*.

$$
\begin{array}{l}
\hline
\Delta RateOfCompletionsStep[KV, STATEMENT] \\
Step \\
ProcessCompletionStatement \\
state?, state! : KV \\
stmt? : STATEMENT \\
step\_ : KV \times STATEMENT \twoheadrightarrow KV \\
\hline
Step.step = \langle body \rangle \\
step = \langle processStatement\_ \rangle \\
Step.step \leadsto step \Rightarrow \\
\qquad \langle body \rangle \equiv \langle \langle parseStmt\_, getPreviousState\_ \rangle, \\
\qquad\qquad \langle returnIntervalStart\_, returnIntervalEnd\_, replaceAt\_ \rangle, \\
\qquad\qquad \langle returnUpdatedCount\_, replaceAt\_ \rangle, \\
\qquad\qquad \langle returnUpdatedNames\_, replaceAt\_ \rangle \rangle \\
\\
state! = step(state?, stmt?) = processStatement(stmt?, state?) \bullet \\
\qquad state! \neq state? \wedge \\
\qquad getIn(state!, \langle roc, completions, stmt_{objId} \rangle) \neq \emptyset \\
\hline
\end{array}
$$

For each unique $stmt_{objId}$ passed to $step$, there should be a corresponding mapping in $state_{\langle roc, completions \rangle}$ which looks like

$$
\begin{array}{l}
stmt_{objId} \mapsto \langle\!\langle domain \mapsto \langle\!\langle (start, interval_{start}!), (end, interval_{end}!) \rangle\!\rangle \\
\qquad\qquad nStmts \mapsto nStmts! \\
\qquad\qquad names \mapsto names! \rangle\!\rangle
\end{array}
$$

9.7 Result

The interval of $interval_{start}!$ to $interval_{end}!$ can be partitioned based on the passed in opt named $timeUnit$ such that for each unique $stmt_{objId}$, the metric n completions per time unit can be calculated and added to $state_{\langle roc, completions, stmt_{objId}, rate \rangle}$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad RateOfCompletionsResult[KV, KV] \qquad\qquad\qquad\qquad}} \\
Result \\
RateOf \\
replaceAt \\
opt?, state?, result! : KV \\
result\_ : KV \times KV \twoheadrightarrow KV \\
\hline
Result.result = \langle body \rangle \\
\langle body \rangle \rightsquigarrow result = \langle replaceAt\_ \rangle^{\#\,\mathrm{dom}(state_{\langle roc, completions \rangle})} \\
\\
state! = result(state?, opt?) \bullet \\
\qquad\qquad let\ timeUnit == atKey(opt?, timeUnit) \bullet \\
\quad \forall stmt_{objId} \in state_{\langle roc, completions \rangle} \mid \exists_1 state!_{objId} \bullet \\
\qquad\qquad\qquad let\ \ nO == getIn(state?, \langle roc, completions, stmt_{objId}, nStmts \rangle) \\
\qquad\qquad\qquad\qquad start == getIn(state?, \langle roc, completions, stmt_{objId}, domain, start \rangle) \\
\qquad\qquad\qquad\qquad end == getIn(state?, \langle roc, completions, stmt_{objId}, domain, end \rangle) \\
\qquad\qquad\qquad\qquad rate_{objId} == rateOf(nO, start, end, timeUnit) \\
\qquad\quad state!_{objId} = replaceAt(state?, \langle roc, completions, stmt_{objId}, rate \rangle, rate_{objId}) \\
\quad = \sum state!_{objId} \bullet \forall stmt_{objId} \mid getIn(state!, \langle roc, completions, stmt_{objId}, rate \rangle) \in \mathbb{Z} \\
\hline
\end{array}
$$

- $state! = \sum state!_{objId}$ is an alternative way to describe the aggregation of all changes made to $state?$.
- $rateOf$ performs the calculation which is used to update $state?$ with consideration to $opt?$
- $timeUnit$ will default to day if not specified within $opt?$

The output of $rateOfCompletions$ is a $state!$ which contains a mapping of following shape for each unique $stmt_{objId}$ passed to $rateOfCompletions$, each of which can be found at the path $\langle roc, completions, stmt_{objId} \rangle$.

$$
\begin{aligned}
stmt_{objId} \mapsto \langle\!\langle & domain \mapsto \langle\!\langle (start, interval_{start}!), (end, interval_{end}!) \rangle\!\rangle \\
& nStmts \mapsto nStmts! \\
& names \mapsto names! \\
& rate \mapsto rate_{objId} \rangle\!\rangle
\end{aligned}
$$

Any mapping within this structure can be used within a corresponding visualization but the core piece of information the visualization should convey is $rate_{objId}$.

9.8 Conclusion

This concludes the first example of a DAVE Algorithm formal definition. The conventions established within this section should be used across all DAVE Algorithm formal definitions in this document. If any aspect of this section requires further explanation or clarification, please post an issue to the DAVE github repo describing the issue or reach out to the Author(s) of this report via some other medium.

Appendix A: Visualization Exemplars

Appendix A includes a typology of data visualizations which may be supported within DAVE workbooks. These visualizations can either be one to one or one to many in regards to the algorithms defined within this document. Future iterations of this document will increasingly include these typologies within the domain-question template exemplars.

Line Charts

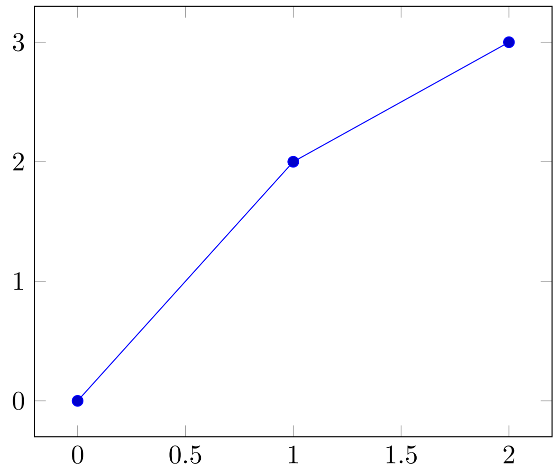

Figure 1: Line Chart

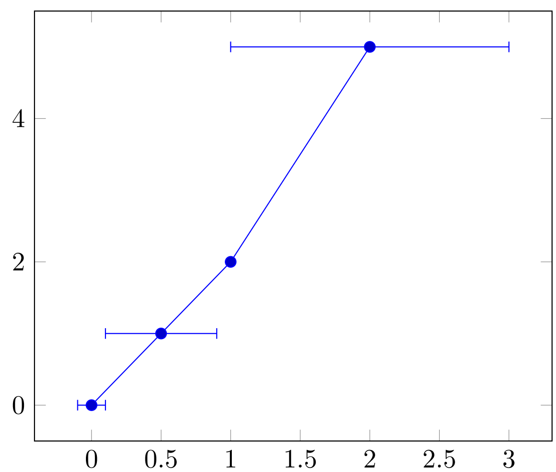

Figure 2: Line Chart with Error

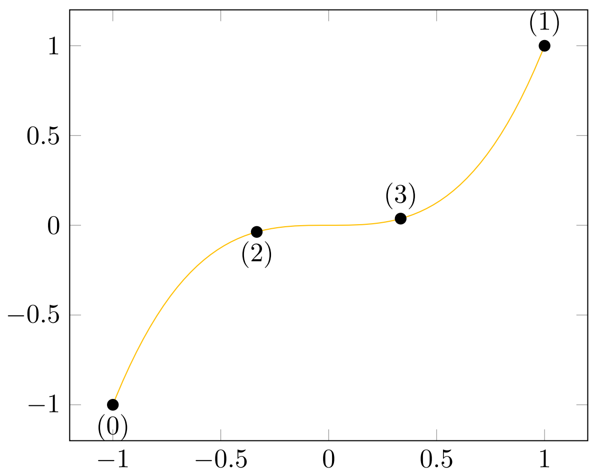

Figure 3: Spline Chart

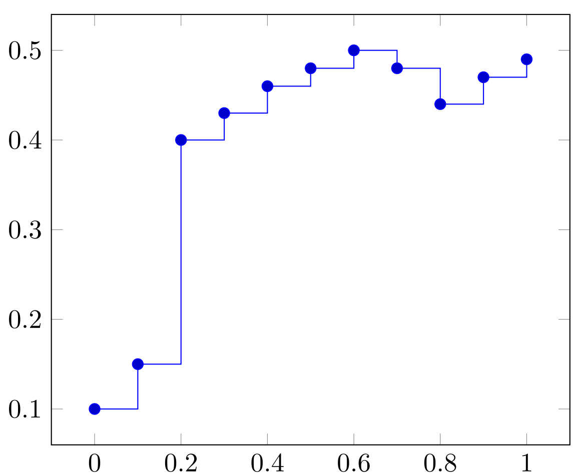

Figure 4: Quiver Chart

Grouping Charts

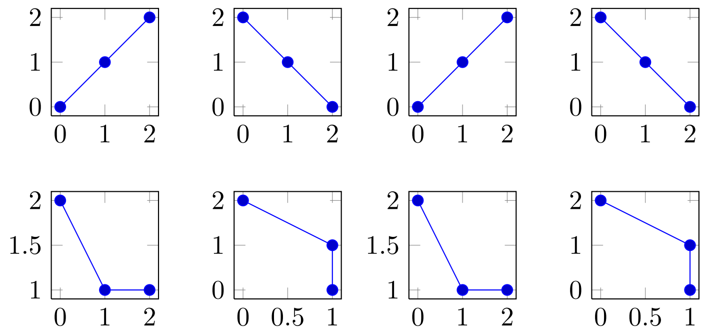

Figure 5: Grouped Line Charts

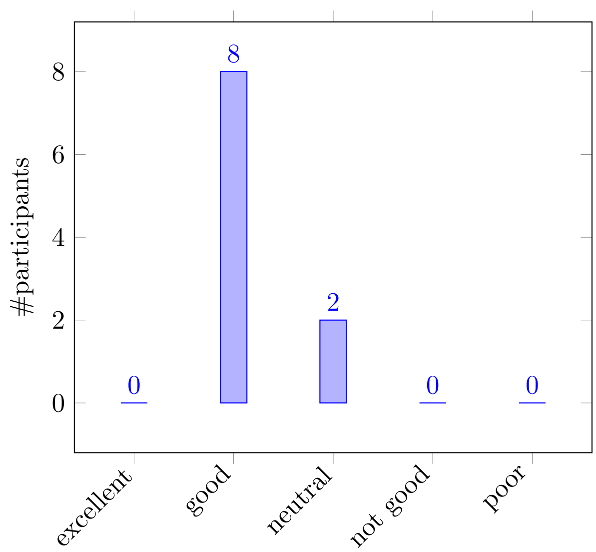

Figure 6: Histogram

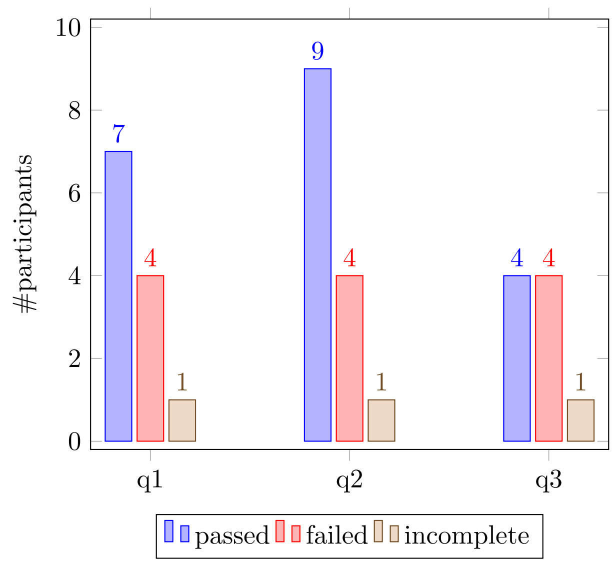

Figure 7: Bar Chart

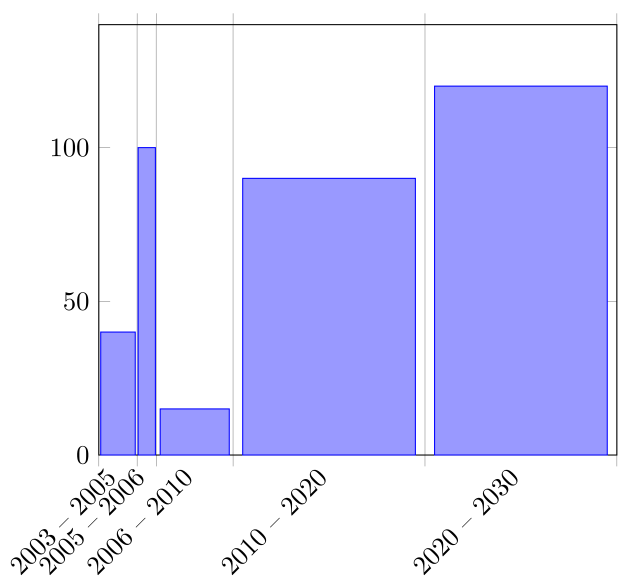

Figure 8: Bar Chart Grouped by Time Range

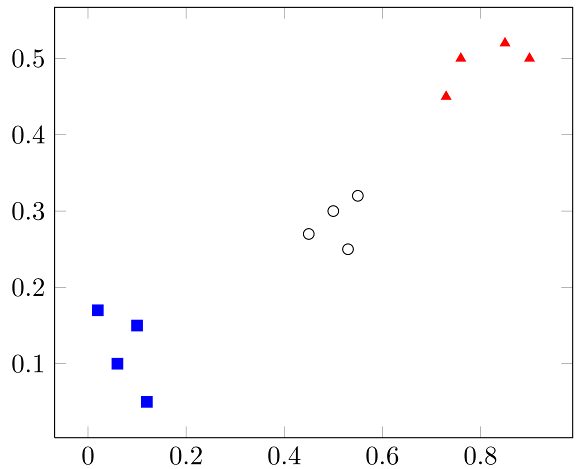

Figure 9: Scatter Plot

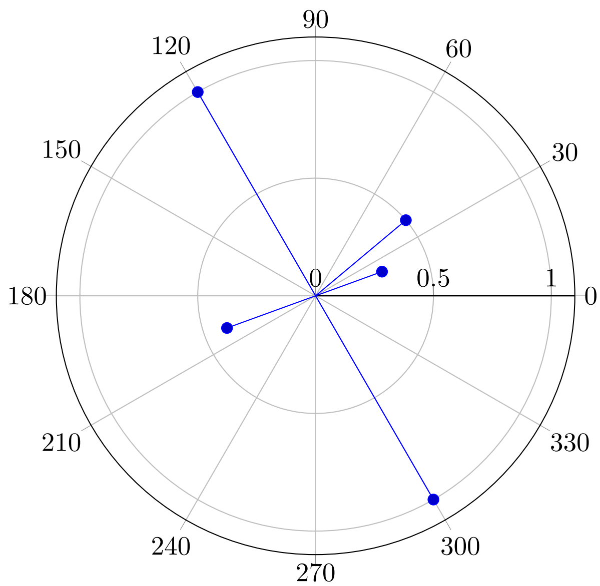

Figure 10: Polar Chart

Specialized Charts

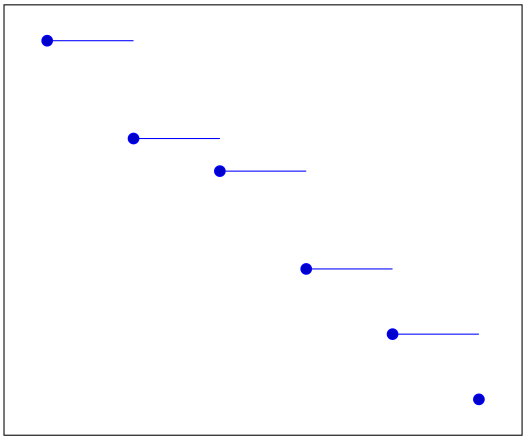

Figure 11: Gantt Chart

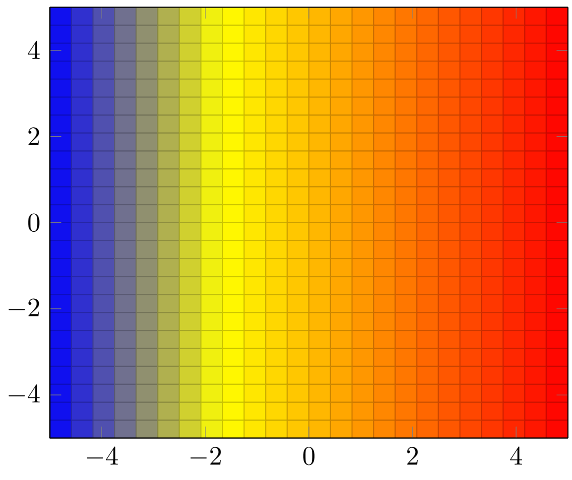

Figure 12: Heat Map

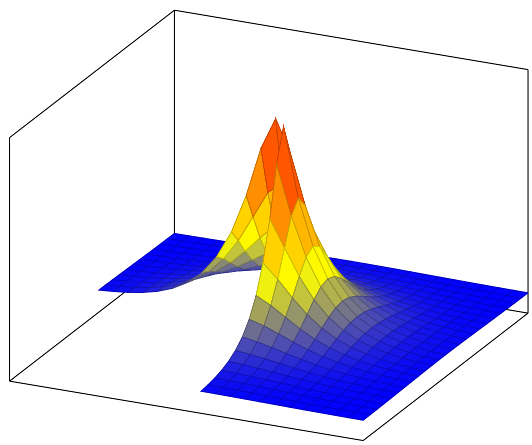

Figure 13: 3D Plot

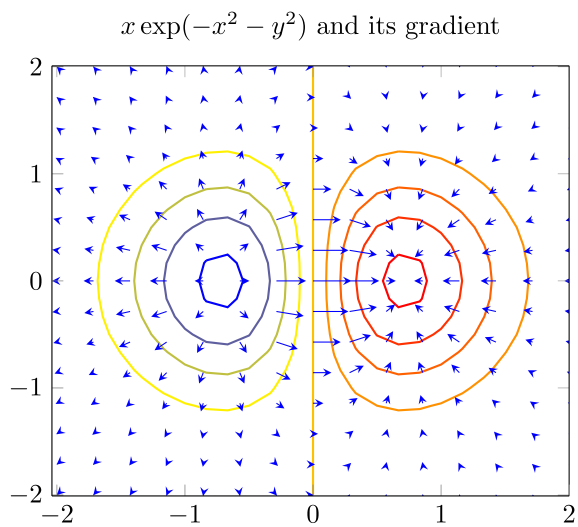

Figure 14: Gradient Plot