1 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.

1.1 Terminology

Within this document, (s) indicates one or more. When talking about some $x \in X$ at some index within a range $i \mathinner{..} n \mathinner{..} j$, the notation $i_x \vee n_x, \vee j_x$ may be used in cases where it is a more concise version of an equivalant expression.

1.1.1 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 respectively.

1.1.2 Collection

a sequence $\langle ... \rangle$ of items c such that

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

And the following free type is introducted for collections

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

$emptyColl$	[the empty Collection $\langle\rangle$]
$append$	[is a constructor and is infered to be an injection]
Map	[a free type introduced bellow]

$$append(emptyColl, c?, 0) = \langle c_0 \rangle \Rightarrow \{\, 0 \mapsto c? \,\}$$

1.1.3 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, Map) but the Scalar null should be avoided. The following free type is introduced for keys.

$$K ::= Scalar \,|\, Collection \,|\, Map$$

1.1.4 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, Map) The following free type is introduced for values.

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

1.1.5 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

$$\begin{array}{l} \mathrm{id}\, v_n = k_n \\ \mathrm{id}\, k_n \neq v_n \\ \mathrm{id}\, k_n = k_n \end{array}$$

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

$$\begin{array}{l} i_k \neq n_k \neq j_k \\ i_v = n_v \lor i_v \neq n_v \;\; i_v = j_v \lor i_v \neq j_v \;\; j_v = n_v \lor j_v \neq n_v \end{array}$$

which can all be stated formally as

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

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.

1.1.6 Statement

Immutable Map conforming to the xAPI Specification as described in the xAPI Formal Definition section of this document. The imutability 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*.

1.1.7 Algorithm State

Mutable Map *state* without any domain restriction such that

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

1.1.8 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.

2 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
- $head$
- $last$
- $tail$
- $front$
- $\upharpoonleft$
- $\upharpoonright$
- $\frown/$
- $\mathsf{disjoint}$
- $\mathsf{partition}$
- $\otimes$
- $\uplus$
- $\boxminus$
- $items$

2.1 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)
- Map(s)
- Statement(s)
- Algorithm State

2.2 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)
- Map(s)
- Statement(s)
- Algorithm State

3 Primitive

A collection of Operations where the output of an Operation o is passed as an argument to the next Operation.

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

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

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

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.

3.1 Domain

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

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

3.2 Range

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

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

4 Algorithm

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

1. call *init*
2. for each $stmt \in S_{<a..b..c>}$
   (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

Such that the execution of A can be described as

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

4.1 Domain

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

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

4.2 Range

An Algorithm will return an Algorithm State.

- Algorithm State

4.3 Initialization

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

$$
\begin{array}{|l}
\hline
\textit{Init}[KV] \\
\textit{state}?, \textit{state}! : KV \\
\textit{init} : KV \rightarrow KV \\
\hline
\textit{state}! = \textit{init}(\textit{state}?) \bullet \textit{state}! = \textit{state}? \vee \textit{state}! \neq \textit{state}? \\
\hline
\end{array}
$$

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 *body* of an Algorithms *init*

4.3.1 Domain

- Algorithm State

4.3.2 Range

- Algorithm State

4.4 Relevant?

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

$$
\begin{array}{|l}
\hline
\textit{Relevant}?[KV, STATEMENT] \\
\textit{state}? : KV \\
\textit{stmt}? : STATEMENT \\
\textit{relevant}? : KV \times STATEMENT \rightarrow \textit{Boolean} \\
\hline
\textit{relevant}?(\textit{state}?, \textit{stmt}?) = \textit{true} \vee \textit{false} \\
\hline
\end{array}
$$

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 $relevant?$

4.4.1 Domain

- Statement
- Algorithm State

4.4.2 Range

- Boolean

4.5 Accept?

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

$$\begin{array}{|l}
\hline
Accept?\,[KV, STATEMENT] \\
state? : KV \\
stmt? : STATEMENT \\
accept? : KV \times STATEMENT \rightarrow Boolean \\
\hline
accept?\,(state?\,, stmt?\,) = true \vee false \\
\hline
\end{array}$$

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*?

4.5.1 Domain

- Statement
- Algorithm State

4.5.2 Range

- Scalar

4.6 Step

An Algorithm Step consists of a sequential composition of Primitive(s) and therefore sequential composition of all Operation(s) where the output of a Primitive is passed as the argument to the next Primitive.

$$p_i \mathbin{⨾} p_n \mathbin{⨾} p_j \Rightarrow o_{ii} \mathbin{⨾} o_{in} \mathbin{⨾} o_{ij} \mathbin{⨾} o_{ni} \mathbin{⨾} o_{nn} \mathbin{⨾} o_{nj} \mathbin{⨾} o_{ji} \mathbin{⨾} o_{jn} \mathbin{⨾} o_{jj}$$

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.

4.6.1 Domain

- Statement
- Algorithm State

4.6.2 Range

- Algorithm State

4.6.3 Formal Definition

A collection of Primitive(s)

$$P =< p_i..p_n..p_j >$$

where

$$i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j$$

and

$$Z_i = p_i(Args) \Rightarrow O_{ij}(O_{in}(O_{ii}(Args)))$$

where

$$ii \leq in \leq ij \Rightarrow ii \prec in \prec ij \iff ii \neq in \neq ij$$

such that for each $stmt_b$ within a collection of Statement(s) S defined as

$$S =< stmt_a..stmt_b..stmt_c >$$

where

$$a \leq b \leq c \Rightarrow a \prec b \prec c \iff a \neq b \neq c$$

and

$$a \not\mapsto i \ \wedge b \not\mapsto n \ \wedge c \not\mapsto j$$

The output of *step* given a $stmt_b$ and $state_b$ is defined as

$$step(state_b, stmt_b) = p_j(p_n(Z_{ib}))$$

where

$$Z_{ib} = p_i(Args) \Rightarrow p_i(state_b, stmt_b) \Rightarrow O_{ij}(O_{in}(O_{ii}(state_b, stmt_b)))$$

and subsequently

$$Z_{nb} = p_n(Z_{ib})$$

which establishes that

$$Z_{jb} = p_j(Z_{nb}) \Rightarrow p_j(p_n(p_i(state_b, stmt_b)))$$

such that for a given $stmt_b$, $P_{<i..n..j>}$ will always result in a Z_{jb} but

$$Z_{ib} = state_b \vee state'_{ib} \iff state_b \neq state'_{ib}$$

which means

$$Z_{nb} = p_n(state_b, stmt_b) \vee p_n(state'_{ib}, stmt_b)$$

$$\Rightarrow$$

$$Z_{nb} = state_b \ \vee state'_{ib} \ \vee state'_{nb}$$

$$\Rightarrow$$

$$Z_{nb} = Z_{ib} \vee state'_{nb} \quad \iff \quad state_b \neq state'_{ib} \neq state'_{nb}$$

and concludes with

$$Z_{jb} = p_j(state_b, stmt_b) \vee p_j(state'_{ib}, stmt_b) \vee p_j(state'_{nb}, stmt_b)$$

$$\Rightarrow$$

$$Z_{jb} = state_b \vee state'_{ib} \vee state'_{nb} \vee state'_{jb}$$

$$\Rightarrow$$

$$Z_{jb} = Z_{nb} \vee state'_{jb} \quad \iff \quad state_b \neq state'_{ib} \neq state'_{nb} \neq state'_{jb}$$

such that

$$Z_{jb} \equiv state'_b$$

$$\Rightarrow$$

$$state'_b = state_b \vee state'_{ib} \vee state'_{nb} \vee state'_{jb} \iff state_b \neq state'_{ib} \neq state'_{nb} \neq state'_{jb}$$

the impact being that iteration through all $stmt \in S< a..b..c >$ results in a return of Z_{jc} such that

$$Z_{ja} = step(state_a, stmt_a) \ \Rightarrow \ state'_a \equiv Z_{ja} = state_a \vee state'_{ia} \vee state'_{na} \vee state'_{ja}$$

and

$$Z_{jb} = step(Z_{ja}, stmt_b) \;\Rightarrow\; state'_b \equiv Z_{jb} = Z_{ja} \vee state'_{ib} \vee state'_{nb} \vee state'_{jb}$$

meaning

$$Z_{jc} = step(Z_{jb}, stmt_c) \;\Rightarrow\; state'_c \equiv Z_{jc} = Z_{jb} \vee state'_{ic} \vee state'_{nc} \vee state'_{jc}$$

such that each $stmt \in S< a..b..c >$ may not result in a mutation of $state$ from $state \rightarrow state'$

$$state'_c = Z_{jc}$$

$$\Rightarrow$$

$$state'_c = state_a \vee state_{ia} \vee state'_{na} \vee state'_{ja} \vee state'_{ib} \vee state'_{nb} \vee state'_{jb} \vee state'_{ic} \vee state'_{nc} \vee state'_{jc}$$

$$\Rightarrow$$

$$state'c = state_a \vee state'c \neq statea$$

The no-op scenario described above is only a possibility of $step(state_a, stmt \in S_{<a..b..c>})$ but can be predicted to occur given

- The definition of individual Operations O which constitute a Primitive p

$$Operation(X) = Y \wedge Operation(X') = Y' \Rightarrow Y = Y' \iff X = X'$$

- The ordering of $O_{<i..n..j>}$ within p

$$i \prec n \prec j$$

- The Primitive(s) p chosen for inclusion within $P_{<i..n..j>}$

$$Z_i = p_i(Args) \Rightarrow O_{ij}(O_{in}(O_{ii}(Args)))$$

$$Z_j = p_j(Args) \Rightarrow O_{jj}(O_{jn}(O_{ji}(Args)))$$

$$\forall Args \; \exists \; Z_i = Z_j \iff O_{ij}(O_{in}(O_{ii}(Args))) \equiv O_{jj}(O_{jn}(O_{ji}(Args)))$$

$$< p_i, p_j > \equiv < p_j, p_i > \iff Z_i = Z_j$$

- The ordering of $p \in P_{i..n..j}$ which implies the ordering of $O \in p_{<i..n..j>} \in P_{<ii..ij..ni..nj..ji..jj>}$

$$i \prec n \prec j \Rightarrow ii \prec in \prec ij \Rightarrow ii \prec ij \prec ni \prec nj \prec ji \prec jj$$

$$P_{i..n..j} = P_{x..y..z} \Rightarrow < p_i, p_n, p_j > \equiv < p_x, p_y, p_z > \iff p_i \equiv p_x \wedge p_n \equiv p_y \wedge p_j \equiv p_z$$

$$\Rightarrow$$

$$P_{i..n..j} = P_{x..y..z} \iff i \mapsto x \wedge n \mapsto y \wedge j \mapsto z \wedge Z_i = Z_x \wedge Z_n = Z_y \wedge Z_j = Z_z$$

- The Key Value pair(s) $kv \in stmt \in S_{<a..b..c>}$
- The ordering of Statement(s) $stmt \in S_{<a..b..c>}$ such that $a \prec b \prec c$

4.7 Result

Last process to run within an Algorithm which returns the Algorithm State *state* without preventing subsequent call of A

$$relevant? \prec accept? \prec step \prec result \prec relevant? \iff S \neq \emptyset$$

$$\Rightarrow$$

$$relevant? \prec accept? \prec step \prec result \iff S = \emptyset$$

such that if $S(t_n) = \emptyset$ and at some future point j within the timeline $i..n..j$ this is no longer true $S(t_j) \neq \emptyset$ then

$$A(state_{n-1},\ S(t_{n-1})) = state_n = A(init(),\ S(t_{n-i})) \iff A(state_n,\ S(t_n)) = state_n$$

such that the statement(s) added to S between t_i and t_n is

$$S(t_{n-i})$$

and the statement(s) added to S between t_n and t_j be

$$S(t_{j-n})$$

such that

$$S(t_{n-i}) \ \cup \ S(t_{j-n}) = S(t_{j-i})$$

which means

$$A(init(),\ S(t_{j-i})) = state_j$$

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

$$A(result(state_n),\ S(t_{j-n})) = A(init(),\ S(t_{j-i})) = state_j$$

$$\iff$$

$$result(state_n) = A(init(),\ S(t_{n-i})) = state_n$$

Which makes A capable of taking in some $S_{<i..n..j..\infty>}$ as not all $s \in S_{<i..\infty>}$ 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 *opts* is determined by the *body* within *result* such that

$$A(result(state_n),\ S(t_{j-n}),\ opts)$$

$$\equiv$$

$$A(init(),\ S(t_{j-i}))$$

$$\equiv$$

$$A(init(),\ S(t_{j-i}),\ opts)$$

$$\equiv$$

$$A(result(state_n),\ S(t_{j-n}))$$

Which implies that *opts* may have an effect on *state* but not in a way which prevents backwards compatibility of *state*

4.7.1 Domain

- Algorithm State
- Option(s)

4.7.2 Range

- Algorithm State