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 $i \,..\, n \,..\, j$, the notation $i_x \lor n_x, \lor j_x$ 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 introduced for collections

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

$emptyColl$ [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

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

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

$$i_k \neq n_k \neq j_k$$
$$i_v = n_v \vee i_v \neq n_v \;\; i_v = j_v \vee i_v \neq j_v \;\; j_v = n_v \vee j_v \neq n_v$$

which can all be stated formally as

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

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.

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

such that $\langle\!\langle\!\langle (k_{ni}, v_{ni}) \rangle\!\rangle\!\rangle$ and $\langle\!\langle\!\langle (k_{nj}, v_{nj}) \rangle\!\rangle\!\rangle$ 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)

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

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
- $head$
- $last$
- $tail$
- $front$
- $\uparrow$
- $\upharpoonright$
- $\frown/$
- disjoint
- partition
- $\otimes$
- $\uplus$
- $\uplus$
- $items$

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) $S_{\langle a..b..c\rangle}$ 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 → 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

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

9

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.

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

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 $\Rightarrow relevant? \prec accept? \prec step$

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

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 $\Rightarrow relevant? \prec accept? \prec step$

$$
\begin{array}{|l}
\underline{\ Accept? \,[KV, STATEMENT]\ \rule[0pt]{8cm}{0.4pt}} \\
state? : KV \\
stmt? : STATEMENT \\
accept? : KV \times STATEMENT \rightarrow Boolean \\
\hline
accept? \,(state? \,, stmt? \,) = true \vee false
\end{array}
$$

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 an argument to the next Primitive.

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

11

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.

$$P = \langle p_i \,..\, p_n \,..\, p_j \rangle \bullet i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j \bullet p_i \gg p_n \gg p_j$$
$$P' = \langle p_{i'} \,..\, p_{n'} \,..\, p_{j'} \rangle \bullet i' \leq n' \leq j' \Rightarrow i' \prec n' \prec j' \iff i' \neq n' \neq j' \bullet p_{i'} \gg p_{n'} \gg p_{j'}$$
$$P'' = \langle p_x \,..\, p_y \,..\, p_z \rangle \bullet x \leq y \leq z \Rightarrow x \prec y \prec z \iff x \neq y \neq z \bullet p_x \gg p_y \gg p_z$$

$$P = P' \iff i \mapsto i' \wedge n \mapsto n' \wedge j \mapsto j'$$
$$P = P'' \iff (i \mapsto x \wedge n \mapsto y \wedge j \mapsto z) \wedge (p_i \equiv p_x \wedge p_n \equiv p_y \wedge p_j \equiv p_z)$$

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

$$S : Collection$$
$$stmt_a, stmt_b, stmt_c : STATEMENT$$
$$state?, step_a!, step_b!, step_c! : KV$$
$$step : KV \times STATEMENT \to KV$$

$$S = \langle stmt_a \,..\, stmt_b \,..\, stmt_c \rangle \bullet a \leq b \leq c \Rightarrow a \prec b \prec c \iff a \neq b \neq c$$
$$step_a! = step(state?, stmt_a) \bullet step_a! = state? \vee step_a! \neq state?$$
$$step_b! = step(step_a!, stmt_b) \bullet step_b! = step_a! \vee step_b! \neq step_a!$$
$$step_c! = step(step_b!, stmt_c) \bullet step_c! = step_b! \vee step_c! \neq step_b!$$

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

$$\underline{Step[KV, STATEMENT]}\rule{7cm}{0.4pt}$$
$$state?, state! : KV$$
$$stmt? : STATEMENT$$
$$step : KV \times STATEMENT \to KV$$

$$state! = step\,(state?, stmt?) = state? \vee \neq state?$$

A change of $state? \to state! \bullet state! \neq state?$ 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) p chosen for inclusion within $P_{\langle i \,..\, n \,..\, j \rangle}$

- The ordering of $p \in P_{\langle i \,..\, n \,..\, j \rangle}$ which implies the ordering of $o \in p_{\langle i \,..\, n \,..\, j \rangle} \in P_{\langle ii \,..\, ij \,..\, ni \,..\, nj \,..\, ji \,..\, jj \rangle}$

- The key value pair(s) $(K, V) \in stmt$

- The ordering of Statement(s) $stmt \in S_{\langle a \,..\, b \,..\, c \rangle}$

4.6.1 Domain

- Statement

- Algorithm State

4.6.2 Range

- Algorithm State

4.7 Result

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

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

and does so without preventing subsequent calls of A

$result[KV, KV]$ ——————————————————————
$result!, state?, opt? : KV$
$result : KV \times KV \rightarrow KV$
———————————————
$result! = result(state?, opt?) = state? \vee \neq state?$
---

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

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

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

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

Which makes A capable of taking in some $S_{\langle i \mathinner{.\,.} n \mathinner{.\,.} j \mathinner{.\,.} \infty \rangle}$ as not all $s \in S_{\langle i \mathinner{.\,.} \infty \rangle}$ 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

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

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

4.7.1 Domain

- Algorithm State

- Option(s)

4.7.2 Range

- Algorithm State

4.8 Algorithm Formal Definition

Given the definitions of Algorithm Components presented above, the previous definition of an Algorithm A

$$Algorithm ::= Init \,\raisebox{0.3ex}{\tiny^{o}}_9\, Relevant? \,\raisebox{0.3ex}{\tiny^{o}}_9\, Accept? \,\raisebox{0.3ex}{\tiny^{o}}_9\, Step \,\raisebox{0.3ex}{\tiny^{o}}_9\, Result$$

can be refined using the helper function $recur$

$$
\begin{array}{|l}
\underline{\;recur[KV, Collection]\;}\rule{8cm}{0pt} \\
\quad state?, state! : KV \\
\quad S?, S! : Collection \\
\quad recur! : (KV, COllection) \\
\quad recur : KV \times Collection \to KV \times Collection \\
\hline
\quad recur! = recur(state?, S?) \Rightarrow (state!, S!) \bullet \\
\qquad state? = state! \wedge S! = tail(S?) \\
\end{array}
$$

to illustrate how an Algorithm processes a Collection of Statement(s)

$$\boxed{\begin{array}{l}
\underline{Algorithm[KV, Collection]}\\[2pt]
Init[KV]\\
Relevant?\,[KV, STATEMENT]\\
Accept?\,[KV, STATEMENT]\\
Step[KV, STATEMENT]\\
result[KV, KV]\\
recur[KV, Collection]\\
opt?\,,state?\,,state_s!\,,state_S!\,:KV\\
S?\,:Collection \bullet \forall s? \in S?\,\mid s?\,:STATEMENT\\
algorithmIter:KV \times Collection \times KV \to KV \times Collection\\
algorithm:KV \times Collection \times KV \to KV\\[2pt]
\hline\\[-6pt]
state_s! = algorithmIter(state?\,,S?\,,opt?\,)\bullet\\
\quad let\quad s_{current} == head(S?\,)\\
\qquad\quad init! == init(state?\,)\\
\qquad\quad relevant! == relevant?\,(init!\,,s_{current})\\
\qquad\quad passon_{relevant}? == recur(init!\,,S?\,) \iff relevant! = false\\
\qquad\quad accept! == accept?\,(init!\,,s_{current}) \iff relevant! = true\\
\qquad\quad passon_{accept}? == recur(init!\,,S?\,) \iff accept! = false\\
\qquad\quad step! == step(init!\,,s_{current}) \iff accept! = true\\
\qquad\quad passon_{step}? == recur(step!\,,S?\,) \iff tail(S?\,) \neq \emptyset\\
\qquad\quad result! == result(step!\,,opt?\,) \iff tail(S?\,) = \emptyset\\
\qquad\quad passon_{result}? == recur(result!\,,S?\,)\\
\qquad\quad passon_s? == passon_{relevant}? \iff relevant! = false \vee\\
\qquad\qquad\qquad passon_{accept}? \iff accept! = false \vee\\
\qquad\qquad\qquad passon_{step}? \iff relevant! = true \wedge accept! = true \wedge tail(S?\,) \neq \emptyset\\
\qquad\qquad\qquad passon_{result}? \iff relevant! = true \wedge accept! = true \wedge tail(S?\,) = \emptyset\\
\qquad\qquad\qquad\; \Rightarrow\; = init!\,\mathbin{\mathring{,}}relevant!\; \vee\\
\qquad\qquad\qquad\qquad = init!\,\mathbin{\mathring{,}}relevant!\,\mathbin{\mathring{,}}accept!\; \vee\\
\qquad\qquad\qquad\qquad = init!\,\mathbin{\mathring{,}}relevant!\,\mathbin{\mathring{,}}accept!\,\mathbin{\mathring{,}}step!\; \vee\\
\qquad\qquad\qquad\qquad = init!\,\mathbin{\mathring{,}}relevant!\,\mathbin{\mathring{,}}accept!\,\mathbin{\mathring{,}}step!\,\mathbin{\mathring{,}}result!\\
\qquad\quad = passon_s? \Rightarrow (first(passon_{relevant}? \vee passon_{accept}? \vee passon_{step} \vee passon_{result}), tail(S?\,))\\
state_S! = algorithm(state?\,,S?\,,opt?\,)\\
\qquad = first(algorithmIter(state?\,,S?\,,opt?\,)^{\#S?})\bullet\\
\qquad\qquad \forall\,s? \in S?\,\exists_1\,passon_s:KV \times Collection \bullet\\
\qquad\qquad\quad passon_s = algorithmIter(state?\,,S?\,,opt?\,)\bullet\\
\qquad\qquad\qquad let\,(state_s == first(passon_s))\bullet\\
\qquad\qquad\qquad\quad passon_{s'} = algorithmIter(state_s,tail(S?\,),opt?\,) \Rightarrow (state_{s'},tail(tail(S?\,)))\\
\qquad\qquad\qquad\quad passon_{s''} = algorithmIter(first(passon_{s'}),second(passon_{s'}),opt?\,)
\end{array}}$$

The above schema introduces the *algorithmIter* function to represent the lifecycle per statement $s \in S$ which illustrates when processing has finished for the current s and the next iteration of *algorithmIter* can start. This results in *algorithm* being defined as an iteration of *algorithmIter* over S such that there is a $passon_s$ for all $s \in S$ and the result of $algorithm(state, S, opt)$ is

$$first(algorithmIter(state, S, opt)^{\#S}) \Rightarrow first(passon_{result}) \iff second(passon_{result}) = \emptyset$$