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.

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 $\vee$ KV respectively.

1.1.2 Collection

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

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

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 $\mathbb{N}$]

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, KV) except for the Scalar null. The following free type is introduced for keys.

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

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, KV) The following free type is introduced for values.

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

1.1.5 Map

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

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

This definition is more accurately

$$KV ::= base \mid associate\langle\langle KV \times K \times V \rangle\rangle$$

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

$$\begin{aligned} &\mathrm{id}\, v_n = k_n \\ &\mathrm{id}\, k_n \neq v_n \\ &\mathrm{id}\, k_n = k_n \end{aligned}$$

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

$$\begin{aligned} &k_i \neq k_n \neq k_j \\ &v_i = v_n \lor v_i \neq v_n \; v_i = v_j \lor v_i \neq v_j \; v_j = v_n \lor v_j \neq v_n \end{aligned}$$

which can all be stated formally as

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

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 immutability of a Statement s is demonstrated by the following which indicates that s was not altered when passed to *associate*.

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

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

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

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

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

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

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)
- KV(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)
- KV(s)
- Statement(s)
- Algorithm State

3 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.

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

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.

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

- 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.

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

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

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

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:

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

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.

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)
- KV(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)
- KV(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 $\rightarrow$ 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.

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

4.1 Initialization

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

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

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*.

4.1.1 Domain

- Algorithm State

4.1.2 Range

- Algorithm State

4.2 Relevant?

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

$$
\begin{array}{|l}
\hline
Relevant?\,[KV, STATEMENT] \\
state? : KV \\
stmt? : STATEMENT \\
relevant?\ \_ : KV \times STATEMENT \to Boolean \\
\hline
relevant? = \langle body \rangle \\
relevant?\,(state?\,, stmt?\,) = true \lor false \\
\hline
\end{array}
$$

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*?

4.2.1 Domain

- Statement
- Algorithm State

4.2.2 Range

- Boolean

4.3 Accept?

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

$$
\begin{array}{|l}
\hline
Accept?\,[KV, STATEMENT] \\
state? : KV \\
stmt? : STATEMENT \\
accept?\ \_ : KV \times STATEMENT \to Boolean \\
\hline
accept? = \langle body \rangle \\
accept?\,(state?\,, stmt?\,) = true \lor false \\
\hline
\end{array}
$$

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*?

4.3.1 Domain

- Statement
- Algorithm State

4.3.2 Range

- Scalar

4.4 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.

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

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

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

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

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

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)

4.4.1 Domain

- Statement
- Algorithm State

4.4.2 Range

- Algorithm State

4.5 Result

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

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

and does so without preventing subsequent calls of A

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

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

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

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

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

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*.

4.5.1 Domain

- Algorithm State
- Option(s)

4.5.2 Range

- Algorithm State

4.6 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).

$$\begin{array}{|l} \textit{Algorithm}[KV, Collection, KV] \\ \hline Algorithm\, Iter,\ Recur,\ Init,\ Result \\ opt?\,, state?\,, state!: KV \\ S?: Collection \bullet \forall s? \in S? \mid s?: STATEMENT \\ algorithm\_ : KV \times Collection \times KV \nrightarrow KV \\ \hline algorithm = \langle\, init\_\,, \langle\, algorithmIter\_\,,\ recur\_ \rangle^{\#\, S?},\ result\_ \rangle \\ state! = algorithm(state?\,, S?\,, opt?\,) \bullet \\ \quad let\ \ init! == init(state?\,) \bullet \\ \quad \forall s_n \in S? \mid s_n : STATEMENT, n : \mathbb{N} \bullet i \leq n \leq j\ \bullet \\ \quad\quad \exists_1 state_n \mid state_n : KV \bullet \\ \quad\quad\quad let\ \ S?_n = tail(S?\,)^{n-i} \\ \quad\quad\quad\quad state_i = algorithmIter(init!\,, S?_n\,) \Rightarrow\ S?_n = S? \iff n = i \\ \quad\quad\quad\quad state_n = recur(state_i,\ S?_n\,,\ \_algorithmIter\_)^{j-1} \iff n \neq i \land n \neq j \\ \quad\quad\quad\quad state_j = recur(state_n, (\{j-1, j\} \upharpoonright S?\,),\ \_algorithmIter\_) \iff n = j \\ \quad\quad\quad\quad state_{j+1} = state_j \Rightarrow recur(state_j, (j \upharpoonright S?\,),\ \_algorithmIter\_) \iff n = j+1 \\ \quad = result(state_j, opt?\,) \\ \hline \end{array}$$

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 $result\_$

4.6.1 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 ($\_ \pfun \_$) 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

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

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.

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

In the context of Algorithms,

$$ack? = AlgorithmState$$
$$S? = CollectionofStatement(s)$$
$$fn? = algorithmIter$$

4.6.2 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.

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

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