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.

0.1 Traversal Operations

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

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

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

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

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

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

0.2 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 $in?$.

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

0.2.1 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 $getIn$.

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

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

$$
\begin{aligned}
& getIn(X, \langle 1, 1 \rangle) = b \\
& getIn(X, \langle 0 \rangle) = true \\
& getIn(X, \langle 2, foo, z, 0 \rangle) = 3
\end{aligned}
$$

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 bellow. *replaceAt* is introduced first and serves as a point of comparison when describing the more abstract Primitives *backProp* and *walkBack*.

0.2.2 Replace At

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

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

- 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 $p \mathrel{..} q$ 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.

0.2.3 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}
\multicolumn{1}{l}{\text{\_\_} \mathit{BackProp}\,[V, \mathit{Collection}, V, (\_ \nrightarrow \_)] \text{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}} \\
\mathit{GetIn} \\
\mathit{in}?\,, \mathit{fnSeed}?\,, \mathit{out}! : V \\
\mathit{path}? : \mathit{Collection} \\
\mathit{fn}? : (\_ \nrightarrow \_) \\
\mathit{backProp}\_ : V \times \mathit{Collection} \times V \times (\_ \nrightarrow \_) \rightarrowtail V \\
\hline
\mathit{backProp} = \langle\langle \mathit{getIn}\,\_, \mathit{fn}?\ \_\rangle, \mathit{recur}\,\_\rangle^{\#\,\mathit{path}?-1} \\
\\
\mathit{out}! = \mathit{backProp}\,(\mathit{in}?\,, \mathit{path}?\,, \mathit{fnSeed}?\,, \mathit{fn}?\,) \bullet \\
\quad \forall n : i \mathinner{..} j-1 \bullet (i = \mathit{first}(\mathit{head}(\mathit{path}?\,))) \wedge (j = \mathit{first}(\mathit{last}(\mathit{path}?\,))) \mid \exists\, \mathit{parent}_n \bullet \\
\qquad\qquad \mathit{let}\ \ \mathit{path}?_n == \mathit{tail}(\mathit{path}?\,)^{n-i} \\
\qquad\quad \mathit{parent}_n = \mathit{recur}(\mathit{parent}_{n-1}, \mathit{path}?_n\,, \mathit{get}\,\_)^{j-1} \Rightarrow \\
\qquad\qquad \mathit{let}\ \ \mathit{parent}_i == \mathit{getIn}(\mathit{in}?\,, \mathit{path}?_n\,) \iff n = i \\
\qquad\qquad\quad \mathit{parent}_{i+1} == \mathit{getIn}(\mathit{parent}_i, \mathit{path}?_n\,) \iff n = i+1 \\
\qquad\qquad\quad \mathit{parent}_{j-1} == \mathit{getIn}(\mathit{parent}_{j-2}, \mathit{path}?_n\,) \iff n = j-1 \\
\qquad\qquad \mathit{parent}_j = \mathit{getIn}(\mathit{parent}_{j-1}, (\mathit{path}? \upharpoonright j)) \\
\\
\quad \forall z : p \mathinner{..} 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\, \mathit{child}_z \bullet \\
\qquad\qquad \mathit{let}\ \ \mathit{path}?_{rev} == \mathit{rev}(\mathit{path}?\,) \\
\qquad\qquad\quad \mathit{path}?_z == \mathit{tail}(\mathit{path}?_{rev}\,)^{p-z+1} \\
\qquad\quad \mathit{child}_z = \mathit{recur}((\mathit{parent}_n, \mathit{child}_{n+1}), \mathit{path}?_z\,, \mathit{fn}?\,) \\
\qquad\qquad \mathit{let}\ \ \mathit{child}_p == \mathit{fn}?\,((\mathit{parent}_n, \mathit{fnSeed}?\,), \mathit{path}?_z\,) \iff z = p \Rightarrow n = j-1 \\
\qquad\qquad\quad \mathit{child}_{p+1} == \mathit{fn}?\,((\mathit{parent}_n, \mathit{child}_p), \mathit{path}?_z\,) \iff n = j-2 \wedge p = j-1 \\
\qquad\qquad\quad \mathit{child}_q == \mathit{fn}?\,((\mathit{parent}_n, \mathit{child}_{q+1}), \mathit{path}?_z\,) \iff z = q \Rightarrow n = i+1 \\
\\
\mathit{out}! = \mathit{fn}?\,((\mathit{in}?\,, \mathit{child}_q), \mathit{path}?_n\,) \equiv \mathit{fn}?\,((\mathit{in}?\,, \mathit{child}_q), (\mathit{path}? \upharpoonright i)) \iff (n = i = q-1) \\
\hline
\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 Primtive *backProp* easier to follow. A quick comparison of *ReplaceAt* and *BackProp* reveals that the only major difference between them is $fn?$ vs $merge\,\_$. This implies the Primitive *backProp* can be used to replicate *replaceAt*.

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

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

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

0.2.4 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 $fn?$.

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

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 interpertation 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 horizonal 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*

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

0.3 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.

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

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

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

0.4 Replace At, Append At and Update At

In the summary of $walkBack$ above, the update at the target location within $in?$ takes place at setp 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 overwritting process, the same constraint is not placed on $fn?_\delta$.

0.4.1 Replace At

The Primitive $replaceAt$ was first defined in terms of the Traversal Operations and then servered 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 bellow such that $replaceAt$ is defined in terms of $walkBack$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ReplaceAt[V, Collection, V] \qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
WalkBack, Merge \\
in?, with?, out!, fn!_\delta : V \\
path? : Collection \\
fn_\delta : V \rightarrow\!\!\!\!\!\!+ \; V \\
replaceAt\_ : V \times Collection \times V \rightarrowtail\!\!\!\!\!\!\rightarrow V \\
\hline
replaceAt = \langle walkBack\_\rangle \\
\\
out! = replaceAt(in?, path?, with?) = walkBack(in?, path?, fn_\delta, merge\_) \bullet \\
\quad let\ \ fn!_\delta == fn_\delta(getIn(in?, path?)) = with? \Rightarrow \\
\qquad walkBack(in?, path?, fn_\delta, merge\_) \equiv \\
\qquad\quad backProp(in?, path?, fn!_\delta, merge\_) \equiv \\
\qquad\qquad backProp(in?, path?, with?, merge\_) \\
\hline
\end{array}
$$

- 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.

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

0.4.2 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

$$conj(x_0, false) = \langle true, false\rangle = \langle x_0, false\rangle$$
$$conj(X, X) = \langle x_0, x_1, x_2, \langle x_0, x_1, x_2\rangle\rangle$$

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.

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

Addative 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.

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

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*.

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

0.4.3 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*.

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