

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, two 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

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

- Navigation down into either a *Collection* or *KV* based on the type of *in?*

$\frac{\text{Merge}[(V, V), \text{Collection}]}{\begin{array}{l} parent?, child?, parent! : V \\ at? : \text{Collection} \\ merge\_ : (V \times V) \times \text{Collection} \mapsto V \end{array}}$
$\begin{array}{l} parent! = merge((parent?, child?), at?) \bullet \\ \quad = (associate(parent?, head(at?), child?) \\ \quad \quad \iff map?(parent?) = true) \vee \\ \quad \quad (update(parent?, child?, head(at?)) \\ \quad \quad \iff (array?(parent?) = true) \wedge (head(at?) \in \mathbb{N})) \end{array}$

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

## 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?*.

$$\begin{array}{l} X = \langle x_0, x_1, x_2 \rangle \\ x_0 = true \\ x_1 = \langle a, b, c \rangle \\ x_2 = \langle\langle foo \mapsto \langle\langle bar \mapsto buz, x \mapsto y, z \mapsto \langle 3, 2, 1 \rangle \rangle \rangle \rangle \rangle \\ fn! = fn(X_{path?_{j-1}}, arg?) = ZZZ \quad [fn\_ \text{ always returns } ZZZ] \end{array}$$

### 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*.

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

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

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

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}
\text{ReplaceAt}[V, \text{Collection}, V] \text{-----} \\
\text{GetIn, Merge} \\
\text{in?}, \text{with?}, \text{out!} : V \\
\text{path?} : \text{Collection} \\
\text{replaceAt}_- : V \times \text{Collection} \times V \rightarrow V \\
\hline
\text{replaceAt} = \langle \langle \text{getIn}_-, \text{merge}_- \rangle, \text{recur}_- \rangle \# \text{path?}^{-1} \\
\\
\text{out!} = \text{replaceAt}(\text{in?}, \text{path?}, \text{with?}) \bullet \\
\quad \forall n : i..j-1 \bullet (i = \text{first}(\text{head}(\text{path?}))) \wedge (j = \text{first}(\text{last}(\text{path?}))) \mid \exists \text{parent}_n \bullet \\
\quad \quad \text{let } \text{path?}_n == \text{tail}(\text{path?})^{n-i} \\
\quad \quad \text{parent}_n = \text{recur}(\text{parent}_{n-1}, \text{path?}_n, \text{get}_-)^{j-1} \Rightarrow \\
\quad \quad \text{let } \text{parent}_i == \text{getIn}(\text{in?}, \text{path?}_n) \iff n = i \\
\quad \quad \text{parent}_{i+1} == \text{getIn}(\text{parent}_i, \text{path?}_n) \iff n = i + 1 \\
\quad \quad \text{parent}_{j-1} == \text{getIn}(\text{parent}_{j-2}, \text{path?}_n) \iff n = j - 1 \\
\quad \quad \text{parent}_j = \text{getIn}(\text{parent}_{j-1}, (\text{path?} \upharpoonright j)) \\
\\
\quad \forall z : p..q \bullet (p = j - 1) \wedge (q = i + 1) \Rightarrow \\
\quad \quad ((z = p \iff n = j - 1) \wedge (z = q \iff n = i + 1)) \mid \exists \text{child}_z \bullet \\
\quad \quad \text{let } \text{path?}_{rev} == \text{rev}(\text{path?}) \\
\quad \quad \text{path?}_z == \text{tail}(\text{path?}_{rev})^{p-z+1} \\
\quad \quad \text{child}_z = \text{recur}((\text{parent}_n, \text{child}_{n+1}), \text{path?}_z, \text{merge}_-) \\
\quad \quad \text{let } \text{child}_p == \text{merge}((\text{parent}_n, \text{with?}), \text{path?}_z) \iff z = p \Rightarrow n = j - 1 \\
\quad \quad \text{child}_{p+1} == \text{merge}((\text{parent}_n, \text{child}_p), \text{path?}_z) \iff n = j - 2 \wedge p = j - 1 \\
\quad \quad \text{child}_q == \text{merge}((\text{parent}_n, \text{child}_{q+1}), \text{path?}_z) \iff z = q \Rightarrow n = i + 1 \\
\\
\text{out!} = \text{merge}((\text{in?}, \text{child}_q), \text{path?}_n) \equiv \text{merge}((\text{in?}, \text{child}_q), (\text{path?} \upharpoonright i)) \iff (n = i = q - 1)
\end{array}$$

- The range of indices  $i..j-1$  is used to describe navigation into some Value given  $\text{path?}$ 
  - Used to reference preceding level of depth
  - keeps track of parent from previous steps
- The range of indices  $p..q$  is used to describe navigation up from target depth indicated by  $\text{path?}$ 
  - Used to reference current level of depth
  - keeps track of child after the update has been applied
- The propagation of the update starts with  $\text{child}_p$ 
  - $\text{with?}$  is added to  $\text{parent}_{j-1}$  at  $\text{get}(\text{path?}, \langle j \rangle)$
  - parent nodes need to be notified of the change within their children

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

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

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

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

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

can be used to replicate *replaceAt*.

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

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

- Within *ReplaceAt*, the naming *with?* indicates its usage with respect to *merge* and the overall functionality of the Primitive
- Within *BackProp*, the naming *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?*.

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

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<sub>j</sub>*) 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?<sub>δ</sub>* which is passed in as an argument. *fn?<sub>δ</sub>* will be passed *parent<sub>j</sub>* 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?<sub>nav</sub>*.

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

In fact, the usage of *fn?<sub>nav</sub>* in *WalkBack* is exactly the same as the usage of *fn?* in *BackProp* as *fn?<sub>nav</sub>* is passed to *backProp* as *fn?*.

$$\begin{array}{l} \text{WalkBack}[V, \text{Collection}, (- \rightarrow -), (- \rightarrow -)] \text{ —————} \\ \text{BackProp} \\ in?, out! : V \\ path? : \text{Collection} \\ fn?_{\delta}, fn?_{nav} : (- \rightarrow -) \\ \text{walkBack\_} : V \times \text{Collection} \times (- \rightarrow -) \times (- \rightarrow -) \rightarrow V \\ \hline \text{walkBack} = \langle \text{getIn\_}, fn?_{\delta}\_, \text{backProp\_} \rangle \\ \\ out! = \text{walkBack}(in?, path?, fn?_{\delta}, fn?_{nav}) \bullet \\ \quad \text{let } fnSeed == fn?_{\delta}(\text{getIn}(in?, path?)) \\ \quad = \text{backProp}(in?, path?, fnSeed, fn?_{nav}) \end{array}$$

By replacing *fnSeed?* with *fn?<sub>δ</sub>* as an argument

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

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

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

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

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

This means *replaceAt* can also be replicated.

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

The following examples demonstrate the functionality of *walkBack*

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

### 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?*

$$\frac{in?_{path?_{j-1}} : V}{path?_{j-1} \Rightarrow path? \triangleleft j \Rightarrow path? \triangleleft (\text{dom } path? \setminus \{j\})}$$

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

$$\frac{in?_{path?} : V}{path? \Rightarrow path?_{j-1} \cup (j, atIndex(path?, j))}$$

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

$$\frac{args? = (atIndex(path?, j), in?_{path?})}{args? \in in?_{path?_{j-1}}}$$

$$first(args?) = atIndex(path?, j)$$

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  $atIndex(path?, j)$

$$\frac{child_j = first(args?) \mapsto fn!}{in!_{path?_{j-1}} = merge((in?_{path?_{j-1}}, fn!), first(args?))}$$

$$child_j \in in!_{path?_{j-1}} \iff child_j \neq args?$$

$$args? \in in?_{path?_{j-1}} \iff args? \neq child_j$$

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

$$\frac{in!_{path?_{j-1}} \triangleleft first(args?) = in?_{path?_{j-1}} \triangleleft first(args?) \iff args? \neq child_j}{args? \neq child_j \Rightarrow second(args?) \neq second(child_j)}$$

$$in!_{path?_{j-1}} \triangleleft first(args?) \Rightarrow in!_{path?_{j-1}} \triangleleft (\text{dom } in!_{path?_{j-1}} \setminus first(args?))$$

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

$$\frac{child_i = atIndex(path?, i) \mapsto in!_{path?_i}}{in!_{path?_i} = merge((in?_{path?_i}, in!_{path?_{i+1}}), atIndex(path?, i+1))}$$

$$out! = merge((in?, second(child_i)), first(child_i)) \bullet$$

$$in? \triangleleft head(path?) = out! \triangleleft head(path?) \Rightarrow$$

$$\forall (a, b) \in path? \bullet b = atIndex(path?, a) \mid \exists a \bullet in?_a = out!_a \iff a \neq head(path?)$$