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. Because of this, the first common Primitive to be introduced is *walk*. In order to define *walk* using the Operation *recur*, the following helper Operations are introduced. These two helper Operations will be used to describe the navigation into and then back out of a nested Value based on the provided Collection of identifiers.

$$
\begin{array}{|l}
\underline{\quad GetNext[V, Collection]\quad\qquad\qquad\qquad\qquad} \\
in?\,, next! : V \\
id? : Collection \\
getNext\,_- : V \times Collection \twoheadrightarrow V \\
\hline
next! = getNext(in?\,, id?\,) \;\bullet \\
\qquad = (atIndex(in?\,, head(id?\,)) \iff (array?\,(in?\,) = true) \wedge (head(id?\,) \in \mathbb{N})) \vee \\
\qquad\quad (atKey(in?\,, head(id?\,)) \iff (array?\,(in?\,) = false) \wedge (map?\,(in?\,) = true))
\end{array}
$$

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

$$
\begin{array}{|l}
\underline{\quad Merge[(V, V), Collection]\quad\qquad\qquad\qquad\qquad} \\
parent?\,, child?\,, parent! : V \\
at? : Collection \\
merge\,_- : (V \times V) \times Collection \rightarrowtail V \\
\hline
parent! = merge((parent?\,, child?\,), at?\,) \;\bullet \\
\qquad = (associate(parent?\,, head(at?\,), child?\,) \\
\qquad\qquad \iff map?\,(parent?\,) = true) \vee \\
\qquad\quad (update(parent?\,, child?\,, head(at?\,)) \\
\qquad\qquad \iff (array?\,(parent?\,) = true) \wedge (head(at?\,) \in \mathbb{N}))
\end{array}
$$

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

The helper Operations defined above are necessary for describing the traversal of a heterogeneous nested Value. Collection and KV have different Fundamental Operations for navigation and update. Their usage in *walk* is touched on within the following summary and expanded further within the formal definition.

1. navigate down into the provided value $in?$ up until the second to last value $in?_{path?_{j-1}}$ as described by the provided $path?$

$$
\begin{array}{l}
in?_{path?_{j-1}} : V \\
\vdash \\
path?_{j-1} \Rightarrow path? \vartriangleleft j \Rightarrow path? \vartriangleleft (\,\mathrm{dom}\ path? \setminus \{j\})
\end{array}
$$

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

$$
\begin{array}{l}
in?_{path?} : V \\
\vdash \\
path? \Rightarrow path?_{j-1} \cup (j, atIndex(path?\,, j))
\end{array}
$$

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

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

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 $atIndex(path?, j)$

$$child_j = first(args?) \mapsto fn!$$
$$in!?_{path?_{j-1}} = merge((in?_{path_{j-1}}, fn!), first(args?))$$
$$\vdash$$
$$child_j \in in!?_{path?_{j-1}}$$
$$child_j \notin in?_{path?_{j-1}} \iff child_j \neq args?$$
$$args? \in in?_{path?_{j-1}}$$
$$args? \notin in!?_{path?_{j-1}} \iff args? \neq child_j$$

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

$$in!?_{path?_{j-1}} \vartriangleleft first(args?) = in?_{path?_{j-1}} \vartriangleleft first(args?) \iff args? \neq child_j$$
$$\vdash$$
$$args? \neq child_j \Rightarrow second(args?) \neq second(child_j)$$
$$in!?_{path?_{j-1}} \vartriangleleft first(args?) \Rightarrow in!?_{path?_{j-1}} \vartriangleleft (\,\mathrm{dom}\; in!?_{path?_{j-1}} \setminus first(args?))$$

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

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

The summary of $walk$ given above is formalized within the schema $Walk$ bellow where $Walk$ dives deeper into the properties/constraints provided for each step. The variables names used in the summary are NOT used in all cases within $Walk$.

$$\underline{\;Walk[V, Collection, (\_ \nrightarrow \_)]\;}$$

$GetNext, Merge, Recur$
$in?, out!, fn! : V$
$path? : Collection$
$fn? : (\_ \nrightarrow \_)$
$walk\_ : V \times Collection \times (\_ \nrightarrow \_) \rightarrowtail V$

---

$walk = \langle\langle getNext\_, recur\_ \rangle^{\# path?-1}, (\_ \nrightarrow \_), \langle merge\_, recur\_ \rangle^{\# path?-1} \rangle$
$out! = walk(in?, path?, fn?) \bullet$
$\quad \forall n : i \mathinner{..} j - 1 \bullet j = first(last(path?)) \Rightarrow$
$\qquad\qquad\qquad\qquad\qquad first(j, path?_j) \mid \exists\, down_n \bullet$
$\quad\quad let \;\; path?_n == tail(path?)^{n-i}$
$\qquad\quad down_i == getNext(in?, path?_n) \Rightarrow$
$\qquad\qquad\qquad atIndex(in?, head(path?)) \lor atKey(in?, head(path?)) \iff n = i$
$\qquad\quad down_n == recur(down_i, path?_n, getNext\_)^{j-1}$
$\qquad\quad down_{j-1} == getNext(down_n, path?_n) \iff n = j - 2$
$\qquad\quad down_j == getNext(down_{j-1}, path?_n) \bullet$
$\qquad\qquad\qquad path?_n \equiv (path? \upharpoonright j) \Rightarrow \langle j \mapsto atIndex(path?, j) \rangle \iff n = j - 1$
$\quad\quad fn! = fn?(down_j)$
$\quad \forall z : p \mathinner{..} q \bullet ((p = j - 1) \land (q = i + 1)) \Rightarrow$
$\qquad\qquad\qquad\qquad ((z = p \iff n = j - 1) \land (z = q \iff n = i + 1)) \mid \exists\, up_n \bullet$
$\quad\quad let \;\; path?_{rev} == rev(path?)$
$\qquad\quad path?_z == tail(path?_{rev})^{p-z+1}$
$\qquad\quad up_p == merge((down_{j-1}, fn!), path?_z) \Rightarrow$
$\qquad\qquad\quad (path?_z \equiv tail(path?_{rev})) \land$
$\qquad\qquad\quad (associate(down_{j-1}, head(path?_z), fn!) \lor$
$\qquad\qquad\quad update(down_{j-1}, fn!, head(path?_z))) \iff z = p$
$\qquad\quad up_z == recur((down_n, up_p), path?_z, merge\_)^p \iff p = n + 1 \land z = n$
$\qquad\quad up_q == merge((down_{i+1}, up_z), path?_z) \iff z = q + 1 \Rightarrow z = i + 2$
$\qquad\quad up_i == merge((down_i, up_q), path?_z) \iff z = q \Rightarrow z = i + 1 \Rightarrow up_i = up_{q-1}$
$out! = merge((in?, up_i), path?_n) \equiv merge((in?, up_i), (path? \upharpoonright i)) \iff (n = i = q - 1)$

---

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

$\quad X = \langle x_0, x_1, x_2 \rangle \land fn! = fn(val?, idx?) = ZZZ$

$\qquad x_0 = true$
$\qquad x_1 = \langle a, b, c \rangle$
$\qquad x_2 = \langle\langle foo \mapsto \langle\langle bar \mapsto buz, x \mapsto y \rangle\rangle \rangle\rangle$
$\quad walk(X, \langle 0 \rangle, array?\_) = \langle false, x_1, x_2 \rangle \qquad\qquad\qquad [true \neg Collection]$
$\quad walk(X, \langle 2, foo, z \rangle, fn\_) = \langle x_0, x_1, \langle\langle foo \mapsto \langle\langle bar \mapsto buz, x \mapsto y, z \mapsto ZZZ \rangle\rangle \rangle\rangle \rangle$
$\quad walk(X, \langle 2, foo, x \rangle, fn\_) = \langle x_0, x_1, \langle\langle foo \mapsto \langle\langle bar \mapsto buz, x \mapsto ZZZ \rangle\rangle \rangle\rangle \rangle$
$\quad walk(X, \langle 2, qux \rangle, fn\_) = \langle x_0, x_1, (x_2 \cup qux \mapsto ZZZ) \rangle$
$\quad walk(X, \langle 1 \rangle, map(succ\_, x_1, 1)) = \langle x_0, \langle b, c, d \rangle, x_2 \rangle$
$\quad walk(X, \langle 1, 0 \rangle, succ\_) = \langle x_0, \langle b, b, c \rangle, x_2 \rangle$