0.1 Accumulate

Performs an update at *path* within *state* using the supplied *item* or $k \ \wedge \ v$

$$accumulate(state, path, item) \rightarrow state'$$

$$accumulate(state, path, k, v) \rightarrow state'$$

0.1.1 Arguments

- *state* is an Algorithm State
- *path* is a Collection of Key(s) used to navigate into *state*
- *item* is a Scalar which should be reflected within $state'$ at *path*
- k is a Value used as the target
  - Index within some Collection
  - Key within some KV such that $k \mapsto v \in KV$
- v is a Value
  - Added to some Collection at index k
  - Mapped to k within some KV such that $k \mapsto v \in KV$

0.1.2 Relevant Operations

The primitive *accumulate* uses the operations

- array?
- object?
- append
- associate
- atKey
- count

0.1.3 Summary

accumulate will do one of the following things

- replace an existing non-array Scalar or KV

  $$accumulate(state, path, item) \equiv associate(state, path, item)$$

- update an existing array Scalar or Collection

  $$accumulate(state, path, item) \equiv associate(state, path, append(state_{path}, item, count(state_{path})))$$

- updates an existing Scalar object or KV

  $accumulate(state, path, k, v) \equiv associate(state, append(path, k, count(path)), v)$

- updates an existing array Scalar or Collection

  $accumulate(state, path, k, v) \equiv associate(state, path, append(state_{path}, v, k))$

- create a new Collection containing $state_{path}$ and v

  $accumulate(state, path, k, v) \equiv associate(state, path, append(append(<>, state_{path}, 0), v, k))$

0.1.4 Usage of Operations

In order to update the argument *state* at *path* using *item* or $k \ \wedge \ v$ the first step is always retrieving the value at *path* using the operation $atKey$. This operation is used because by definition, *state* is a KV

$$state_{path} = atKey(state, path)$$

to determine its type

$$state_{path} = Object \ \vee \ KV \ \vee \ x \ \vee X$$

such that the following bullet points represent the beahvior of *accumulate* under various conditions

- $object?\,(state_{path}) = true$
  - and *item* passed in as argument

    $$updatedState = associate(state, path, item)$$

  - and $k \ \wedge v$ passed in as argument

    $$index = count(path)$$

    $$fullPath = append(path, k, index)$$

    $$updatedState = associate(state, fullPath, v)$$

- $array?\,(state_{path}) = true$
  - and *item* passed in as argument

    $$index = count(state_{path})$$

    $$updatedArray = append(state_{path}, item, index)$$

    $$updatedState = associate(state, path, updatedArray)$$

  - and $k \ \wedge \ v$ passed in as argument

$$updatedArray = append(state_{path}, v, k)$$

$$updatedState = associate(state, path, updatedArray)$$

- $array?\,(state_{path}) = false \ \wedge \ object?\,(state_{path}) = false$
  - and $item$ passed in as argument

$$updatedState = associate(state, path, item)$$

  - and $k \ \wedge \ v$ passed in as argument

$$newArray = append(<>, state_{path}, 0)$$

$$updatedArray = append(newArray, v, k)$$

$$updatedState = associate(state, path, updatedArray)$$

Which shows that *accumulate* has common steps across all conditions

$$state_{path} = atKey(state, path)$$

$$objectAtPath? = object?\,(state_{path})$$

$$arrayAtPath? = array?\,(state_{path})$$

but then the steps deviate based $item$ vs $k \ \wedge \ v$ such that the action of *accumulate* when $item$ is passed in results in either

- an overwrite of $state_{path}$ via $associate(state, path, item)$
  - $objectAtPath? = true$
  - $objectAtPath? = false \ \wedge \ arrayAtpath? = false$
- an updated $state_{path}$ via $associate(state, path, append(state_{path}, item, count(state_{path})))$
  - $arrayAtPath? = true$

and the action of *accumulate* when $k \ \wedge \ v$ is passed in results in either

- $objectAtPath? = true$
  - an update of $state_{path}$ to include $k \mapsto v$

$$associate(state, append(path, k, count(path)), v)$$

- $arrayAtPath? = true$
  - an update of $state_{path}$ to include v at index k

$$associate(state, path, append(state_{path}, v, k))$$

- $objectAtPath? = false \ \wedge \ arrayAtpath? = false$
  - creation of a new array which contains $state_{path}$ and v at index k

$$associate(state, path, append(append(<>, state_{path}, 0), v, k))$$

0.1.5 Example output

To demonstrate the functionality of *accumulate*, the following assumptions will be made

$$state =< a \mapsto< b \mapsto< 1, 2, 3 >, c \mapsto 4 > d \mapsto foo, e \mapsto< 4, 5, 6 >>$$

$$\Rightarrow$$

$$state_a =< b \mapsto< 1, 2, 3 >, c \mapsto 4 >$$

$$state_d = foo$$

$$state_e =< 4, 5, 6 >$$

such that

$$accumulate(state, < d >, baz) =< state_a,\ d \mapsto baz,\ state_e >$$

and

$$accumulate(state, < a >, baz) =< a \mapsto baz,\ state_d,\ state_e >$$

and

$$accumulate(state, < a, c >, baz) =< a \mapsto< b \mapsto< 1, 2, 3 >, c \mapsto baz >,\ state_d,\ state_e >$$

and

$$accumulate(state, < e >, 7) =< state_a,\ state_d,\ e \mapsto< 4, 5, 6, 7 >>$$

and

$$accumulate(state, < e >, < 7, 8, 9 >) =< state_a,\ state_d,\ e \mapsto< 4, 5, 6, < 7, 8, 9 >>>$$

and

$$accumulate(state, < a >, b, < 3, 2, 1 >) =< a \mapsto< b \mapsto< 3, 2, 1 >, c \mapsto 4 >,\ state_d,\ state_e >$$

and

$$accumulate(state, < a >, q, baz) =< a \mapsto< b \mapsto< 1, 2, 3 >, c \mapsto 4, q \mapsto baz >,\ state_d,\ state_e >$$

and

$$accumulate(state, < a, q >, r, baz) =< a \mapsto< b \mapsto< 1, 2, 3 >, c \mapsto 4, q \mapsto r \mapsto baz >,\ state_d,\ state_e >$$

and

$$accumulate(state, < e >, 1, 7) =< state_a,\ state_d,\ e \mapsto< 4, 7, 5, 6 >>$$

and

$$accumulate(state, < d >, 0, baz) =< state_a,\ < baz, foo >,\ state_e >$$

and

$$accumulate(state, < d >, 1, < baz, bar >) =< state_a,\ < foo, < baz, bar >>,\ state_e >$$