

# Formalization of an Algorithm for Greedily Computing Associative Aggregations on Sliding Windows

Lukas Heimes      Dmitriy Traytel      Joshua Schneider

December 14, 2021

## Abstract

Basin et al.'s sliding window algorithm (SWA) [1] is an algorithm for combining the elements of subsequences of a sequence with an associative operator. It is greedy and minimizes the number of operator applications. We formalize the algorithm and verify its functional correctness. We extend the algorithm with additional operations and provide an alternative interface to the slide operation that does not require the entire input sequence.

## Contents

<b>1</b>	<b>Sliding Window Algorithm</b>	<b>1</b>
<b>2</b>	<b>Correctness</b>	<b>4</b>
2.1	Correctness of the Slide Function . . . . .	5
2.2	Correctness of the Sliding Window Algorithm . . . . .	7
2.3	Summary of the Correctness Proof . . . . .	7
<b>3</b>	<b>Alternative Slide Interface and Additional Operations</b>	<b>7</b>
3.1	Alternative Slide Interface . . . . .	7
3.2	Updating all Values in the Tree . . . . .	8
3.3	Updating the Rightmost Leaf of the Tree . . . . .	9

## 1 Sliding Window Algorithm

```
datatype 'a tree =  
  Leaf  
| Node (l: nat) (r: nat) (val: 'a option) (lchild: 'a tree) (rchild: 'a tree)  
where  
  l Leaf = 0  
| r Leaf = 0  
| val Leaf = None  
| lchild Leaf = Leaf  
| rchild Leaf = Leaf
```

```
lemma neq_Leaf_if_l_gt0: 0 < l t  $\implies$  t  $\neq$  Leaf  
  <proof>
```

```
primrec discharge :: 'a tree  $\Rightarrow$  'a tree where  
  discharge Leaf = Leaf  
| discharge (Node i j _ t u) = Node i j None t u
```

```

instantiation option :: (semigroup_add) semigroup_add begin

fun plus_option :: 'a option ⇒ 'a option ⇒ 'a option where
  plus_option None _ = None
| plus_option _ None = None
| plus_option (Some a) (Some b) = Some (a + b)

instance ⟨proof⟩

end

fun combine :: 'a :: semigroup_add tree ⇒ 'a tree ⇒ 'a tree where
  combine t Leaf = t
| combine Leaf t = t
| combine t u = Node (l t) (r u) (val t + val u) (discharge t) u

lemma combine_non_Leaves:  $\llbracket t \neq \text{Leaf}; u \neq \text{Leaf} \rrbracket \implies \text{combine } t \ u = \text{Node } (l \ t) \ (r \ u) \ (\text{val } t + \text{val } u)$ 
  (discharge t) u
  ⟨proof⟩

lemma r_combine_non_Leaves:  $\llbracket t \neq \text{Leaf}; u \neq \text{Leaf} \rrbracket \implies r \ (\text{combine } t \ u) = r \ u$ 
  ⟨proof⟩

type_synonym window = nat × nat

definition window :: 'a list ⇒ window ⇒ bool where
  window as = (λ(l, r). 0 < l ∧ l ≤ r ∧ r ≤ length as)

definition windows :: 'a list ⇒ window list ⇒ bool where
  windows as ws = ((∀ w ∈ set ws. window as w) ∧
    sorted (map fst ws) ∧ sorted (map snd ws))

function reusables :: 'a tree ⇒ window ⇒ 'a tree list where
  reusables t w = (if fst w > r t then [] else if fst w = l t then [t]
    else let v = lchild t; u = rchild t in if fst w ≥ l u then
      reusables u w else u # reusables v w)
  ⟨proof⟩

termination
  ⟨proof⟩

declare reusables.simps[simp del]

lemma reusables_Leaf[simp]: 0 < fst w ⇒ reusables Leaf w = []
  ⟨proof⟩

primrec well_shaped :: 'a tree ⇒ bool where
  well_shaped Leaf = True
| well_shaped (Node i j _ t u) = (i ≤ j ∧ (i = j → t = Leaf ∧ u = Leaf) ∧
  (i < j → t ≠ Leaf ∧ u ≠ Leaf ∧ well_shaped t ∧ well_shaped u ∧
    i = l t ∧ j = r u ∧ Suc (r t) = l u))

lemma l_lchild_eq_l_if_well_shaped[simp]:
   $\llbracket \text{well\_shaped } t; l \ t < r \ t \rrbracket \implies l \ (\text{lchild } t) = l \ t$ 
  ⟨proof⟩

lemma r_rchild_eq_r_if_well_shaped[simp]:
   $\llbracket \text{well\_shaped } t; l \ t < r \ t \rrbracket \implies r \ (\text{rchild } t) = r \ t$ 
  ⟨proof⟩

```

**lemma** *r\_lchild\_eq\_l\_rchild\_if\_well\_shaped*:  
 $\llbracket \text{well\_shaped } t; l\ t < r\ t \rrbracket \implies r\ (\text{lchild } t) = l\ (\text{rchild } t) - 1$   
 ⟨proof⟩

**lemma** *r\_lchild\_le\_r*:  $\text{well\_shaped } t \implies r\ (\text{lchild } t) \leq r\ t$   
 ⟨proof⟩

**lemma** *well\_shaped\_lchild[simp]*:  $\text{well\_shaped } t \implies \text{well\_shaped } (\text{lchild } t)$   
 ⟨proof⟩

**lemma** *well\_shaped\_rchild[simp]*:  $\text{well\_shaped } t \implies \text{well\_shaped } (\text{rchild } t)$   
 ⟨proof⟩

**definition** *adjacent where*

$\text{adjacent } w\ ts = (\text{Leaf} \notin \text{set } ts \wedge$   
 $\text{list\_all2 } (\lambda t\ u.\ l\ t = \text{Suc } (r\ u))\ (\text{butlast } ts)\ (tl\ ts) \wedge$   
 $(ts = [] \vee (l\ (\text{last } ts) = \text{fst } w \wedge r\ (\text{hd } ts) = \text{snd } w)))$

**lemma** *adjacent\_Nil[simp]*:  $\text{adjacent } w\ []$   
 ⟨proof⟩

**lemma** *adjacent\_Cons*:  $\text{adjacent } w\ (t \# ts) =$   
 $(t \neq \text{Leaf} \wedge r\ t = \text{snd } w \wedge (\text{case } ts\ \text{of } [] \Rightarrow l\ t = \text{fst } w$   
 $| u \# us \Rightarrow \text{adjacent } (\text{fst } w, r\ u)\ ts \wedge l\ t = \text{Suc } (r\ u)))$   
 ⟨proof⟩

**lemma** *adjacent\_ConsI*:  $\llbracket t \neq \text{Leaf}; r\ t = \text{snd } w;$   
 $(\text{case } ts\ \text{of } [] \Rightarrow l\ t = \text{fst } w$   
 $| u \# us \Rightarrow \text{adjacent } (\text{fst } w, r\ u)\ ts \wedge l\ t = \text{Suc } (r\ u)) \rrbracket \implies$   
 $\text{adjacent } w\ (t \# ts)$   
 ⟨proof⟩

**lemma** *adjacent\_singleton*:  $t \neq \text{Leaf} \implies \text{adjacent } (l\ t, r\ t)\ [t]$   
 ⟨proof⟩

**lemma** *append\_Cons\_eq\_append\_append*:  $xs\ @\ y \# ys = xs\ @\ [y]\ @\ ys$   
 ⟨proof⟩

**lemma** *list\_all2\_append\_singletonI*:  $\llbracket \text{list\_all2 } P\ xs\ ys; P\ x\ y \rrbracket \implies \text{list\_all2 } P\ (xs\ @\ [x])\ (ys\ @\ [y])$   
 ⟨proof⟩

**lemma** *list\_all2\_Cons\_append\_singletonI*:  $\llbracket xs \neq []; \text{list\_all2 } P\ (x \# \text{butlast } xs)\ ys; P\ (\text{last } xs)\ y \rrbracket \implies$   
 $\text{list\_all2 } P\ (x \# xs)\ (ys\ @\ [y])$   
 ⟨proof⟩

**lemma** *adjacent\_appendI*:  $\llbracket 0 < \text{fst } w; \text{fst } w \leq \text{snd } w;$   
 $(\text{case } us\ \text{of } [] \Rightarrow \text{adjacent } w\ ts$   
 $| u \# us' \Rightarrow \text{adjacent } (\text{Suc } (r\ u), \text{snd } w)\ ts \wedge \text{adjacent } (\text{fst } w, (\text{case } ts\ \text{of } [] \Rightarrow \text{snd } w | ts \Rightarrow r\ u))\ (u$   
 $\# us')) \rrbracket \implies$   
 $\text{adjacent } w\ (ts\ @\ us)$   
 ⟨proof⟩

**lemma** *adjacent\_Cons\_implies\_adjacent*:  $\text{adjacent } (a, b)\ (t \# ts) \implies \text{adjacent } (a, l\ t - \text{Suc } 0)\ ts$   
 ⟨proof⟩

**lemma** (in *semigroup\_add*) *fold\_add\_add*:  $\text{fold } (+)\ xs\ (x + y) = \text{fold } (+)\ xs\ x + y$   
 ⟨proof⟩

**context**

**fixes**  $as :: 'a :: semigroup\_add\ list$

**and**  $ws :: window\ list$

**begin**

**abbreviation** *atomic* **where**

$atomic\ i \equiv Node\ i\ i\ (Some\ (nth\ as\ (i - 1)))\ Leaf\ Leaf$

**definition** *atomics*  $:: nat \Rightarrow nat \Rightarrow 'a\ tree\ list$  **where**

$atomics\ i\ j \equiv map\ atomic\ (rev\ [i ..< Suc\ j])$

**definition** *slide*  $:: 'a\ tree \Rightarrow window \Rightarrow 'a\ tree$  **where**

$slide\ t\ w =$

(*let*

$ts = atomics\ (max\ (fst\ w)\ (Suc\ (r\ t)))\ (snd\ w);$

$ts' = reusables\ t\ w$

*in fold combine (ts @ ts') Leaf*)

**primrec** *iterate*  $:: 'a\ tree \Rightarrow window\ list \Rightarrow 'a\ list$  **where**

$iterate\ t\ [] = []$

|  $iterate\ t\ (w \# xs) = (let\ t' = slide\ t\ w\ in\ the\ (val\ t')\ \# iterate\ t'\ xs)$

**definition** *sliding\_window*  $:: 'a\ list$  **where**

$sliding\_window = iterate\ Leaf\ ws$

## 2 Correctness

**abbreviation** *sum* **where**

$sum\ i\ j \equiv fold\ (+)\ (rev\ (map\ (nth\ as)\ [i - 1 ..< j - 1]))\ (nth\ as\ (j - 1))$

**primrec** *well\_valued0*  $:: 'a\ tree \Rightarrow bool$  **where**

$well\_valued0\ Leaf = True$

|  $well\_valued0\ (Node\ i\ j\ a\ t\ u) = (0 < i \wedge j \leq length\ as \wedge (a \neq None \longrightarrow a = Some\ (sum\ i\ j)) \wedge well\_valued0\ t \wedge well\_valued0\ u \wedge (u = Leaf \vee val\ u \neq None))$

**abbreviation** *well\_valued*  $:: 'a\ tree \Rightarrow bool$  **where**

$well\_valued\ t \equiv (well\_valued0\ t \wedge (t \neq Leaf \longrightarrow val\ t \neq None))$

**definition** *valid*  $:: 'a\ tree \Rightarrow bool$  **where**

$valid\ t = (well\_shaped\ t \wedge well\_valued\ t)$

**lemma** *valid\_Leaf*:  $valid\ Leaf$

*<proof>*

**lemma** *add\_sum*:

**assumes**  $i > 0\ j \geq i\ k > j$

**shows**  $sum\ i\ j + sum\ (Suc\ j)\ k = sum\ i\ k$

*<proof>*

**lemma** *well\_valued0\_rchild\_if\_well\_valued0[simp]*:  $well\_valued0\ t \Longrightarrow well\_valued0\ (rchild\ t)$

*<proof>*

**lemma** *well\_valued0\_lchild\_if\_well\_valued0[simp]*:  $well\_valued0\ t \Longrightarrow well\_valued0\ (lchild\ t)$

*<proof>*

**lemma** *valid\_rchild\_if\_valid*:  $valid\ t \Longrightarrow valid\ (rchild\ t)$

*<proof>*

**lemma** *val\_eq\_Some\_sum\_if\_valid\_neq\_Leaf*:  $\llbracket \text{valid } t; t \neq \text{Leaf} \rrbracket \implies \text{val } t = \text{Some } (\text{sum } (l \ t) \ (r \ t))$   
 <proof>

## 2.1 Correctness of the Slide Function

**lemma** *adjacent\_atomics*:  $\text{adjacent } (i, j) \ (\text{atomics } i \ j)$   
 <proof>

**lemma** *valid\_atomics*:  $\llbracket t \in \text{set } (\text{atomics } i \ j); 0 < i; j \leq \text{length } \text{as} \rrbracket \implies \text{valid } t$   
 <proof>

**lemma** *reusables\_neq\_Nil\_if\_well\_shaped\_and\_overlapping*:  
 $\llbracket \text{well\_shaped } t; l \ t \leq \text{fst } w; r \ t \leq \text{snd } w; \text{fst } w \leq r \ t \rrbracket \implies \text{reusables } t \ w \neq []$   
 <proof>

**lemma** *reusables\_lchild\_neq\_Nil\_under\_some\_conditions*:  
 $\llbracket \text{well\_shaped } t; l \ t \leq \text{fst } w; r \ t \leq \text{snd } w; \text{fst } w \neq l \ t; r \ t \geq \text{fst } w; l \ (\text{rchild } t) > \text{fst } w \rrbracket \implies$   
 $\text{reusables } (\text{lchild } t) \ w \neq []$   
 <proof>

**lemma** *adjacent\_reusables*:  $\llbracket 0 < \text{fst } w; \text{well\_shaped } t; l \ t \leq \text{fst } w; r \ t \leq \text{snd } w \rrbracket \implies$   
 $\text{adjacent } (\text{fst } w, r \ t) \ (\text{reusables } t \ w)$   
 <proof>

**lemma** *valid\_rchild\_if\_well\_valued0*:  $\llbracket \text{well\_shaped } t; \text{well\_valued0 } t \rrbracket \implies \text{valid } (\text{rchild } t)$   
 <proof>

**lemma** *valid\_reusables\_under\_some\_conditions*:  
 $\llbracket 0 < \text{fst } w; \text{well\_valued0 } t; \text{well\_shaped } t; l \ t < \text{fst } w; r \ t \leq \text{snd } w \rrbracket \implies$   
 $\forall t' \in \text{set } (\text{reusables } t \ w). \text{valid } t'$   
 <proof>

**lemma** *valid\_reusables*:  
**assumes**  $0 < \text{fst } w \ \text{valid } t \ l \ t \leq \text{fst } w \ r \ t \leq \text{snd } w$   
**shows**  $\forall t' \in \text{set } (\text{reusables } t \ w). \text{valid } t'$   
 <proof>

**lemma** *combine\_valid\_Nodes\_aux*:  
**assumes** *prems*:  $0 < l \ a \ a \neq \text{Leaf} \ z \neq \text{Leaf} \ l \ z = \text{Suc } (r \ a) \ \text{well\_shaped } a \ \text{well\_shaped } z$   
 $\text{well\_valued0 } a \ \text{val } a = \text{Some } va \ \text{well\_valued0 } z \ \text{val } z = \text{Some } vz$   
**shows**  $va + vz = \text{fold } (+) \ (\text{rev } (\text{map } (!) \ \text{as}) \ [l \ a - \text{Suc } 0..<r \ z - \text{Suc } 0]) \ (\text{as} \ ! \ (r \ z - \text{Suc } 0))$   
 <proof>

**lemma** *discharge\_is\_Leaf[simp]*:  $\text{discharge } a = \text{Leaf} \longleftrightarrow a = \text{Leaf}$   
 <proof>

**lemma** *well\_shaped\_discharge[simp]*:  $\text{well\_shaped } a \implies \text{well\_shaped } (\text{discharge } a)$   
 <proof>

**lemma** *well\_valued0\_discharge[simp]*:  $\text{well\_valued0 } a \implies \text{well\_valued0 } (\text{discharge } a)$   
 <proof>

**lemma** *l\_discharge[simp]*:  $l \ (\text{discharge } a) = l \ a$   
 <proof>

**lemma** *r\_discharge[simp]*:  $r (\text{discharge } a) = r a$   
 ⟨proof⟩

**lemma** *well\_shaped\_lr*:  $\text{well\_shaped } a \implies l a \leq r a$   
 ⟨proof⟩

**lemma** *well\_valued0\_r*:  $\text{well\_valued0 } a \implies a \neq \text{Leaf} \implies r a \leq \text{length } a s$   
 ⟨proof⟩

**lemma** *valid\_combine\_if\_valid*:  $\llbracket 0 < l a; \text{valid } a; \text{valid } z; a \neq \text{Leaf}; z \neq \text{Leaf}; l z = \text{Suc } (r a) \rrbracket \implies$   
 $\text{valid } (\text{combine } a z)$   
 ⟨proof⟩

**lemma** *combine\_neq\_Leaf\_if\_both\_non\_Leaf*:  $\llbracket a \neq \text{Leaf}; z \neq \text{Leaf} \rrbracket \implies$   
 $\text{combine } a z \neq \text{Leaf}$   
 ⟨proof⟩

**lemma** *valid\_fold\_combine*:  $\llbracket 0 < \text{fst } w; ts = h \# ts'; \forall t \in \text{set } ts. \text{valid } t; \text{adjacent } (\text{fst } w, l h - 1) ts';$   
 $\text{valid } z; z \neq \text{Leaf}; l z = (\text{case } ts' \text{ of } [] \Rightarrow \text{fst } w \mid t_1 \# ts'' \Rightarrow \text{Suc } (r t_1)); r z = \text{snd } w \rrbracket \implies$   
 $\text{valid } (\text{fold combine } ts' z) \wedge$   
 $l (\text{fold combine } ts' z) = \text{fst } w \wedge r (\text{fold combine } ts' z) = \text{snd } w$   
 ⟨proof⟩

**lemma** *valid\_fold\_combine\_Leaf*:  
**assumes**  $0 < \text{fst } w$   $ts = h \# ts' \forall t \in \text{set } ts. \text{valid } t$   $\text{adjacent } w ts$   
**shows**  $\text{valid } (\text{fold combine } ts \text{ Leaf}) \wedge$   
 $l (\text{fold combine } ts \text{ Leaf}) = \text{fst } w \wedge r (\text{fold combine } ts \text{ Leaf}) = \text{snd } w$   
 ⟨proof⟩

**lemma** *adjacent\_atomics\_nonempty\_reusables*:  
**fixes**  $x :: 'a \text{ tree}$  **and**  $xs :: 'a \text{ tree list}$   
**assumes**  $a1: 0 < \text{fst } w$   
**and**  $a2: l t \leq \text{fst } w$   
**and**  $a3: r t \leq \text{snd } w$   
**and**  $a4: \text{valid } t$   
**and**  $a5: \text{reusables } t w = x \# xs$   
**shows**  $\text{adjacent } (\text{Suc } (r x), \text{snd } w) (\text{atomics } (\text{max } (\text{fst } w) (\text{Suc } (r t))) (\text{snd } w))$   
 ⟨proof⟩

**lemma** *adjacent\_Cons\_r*:  $\text{adjacent } (a, r t) (x \# xs) \implies \text{adjacent } (a, r x) (x \# xs)$   
 ⟨proof⟩

**lemma** *adjacent\_Cons\_r2*:  
 $\text{adjacent } (\text{fst } w, r t) (x \# xs) \implies 0 < \text{fst } w \implies \text{fst } w \leq \text{snd } w \implies r t \leq \text{snd } w \implies$   
 $\text{atomics } (\text{max } (\text{fst } w) (\text{Suc } (r t))) (\text{snd } w) = [] \implies$   
 $\text{adjacent } w (x \# xs)$   
 ⟨proof⟩

**lemma** *adjacent\_append\_atomics\_reusables*:  
 $\llbracket 0 < \text{fst } w; \text{fst } w \leq \text{snd } w; \text{valid } t; l t \leq \text{fst } w; r t \leq \text{snd } w \rrbracket \implies$   
 $\text{adjacent } w (\text{atomics } (\text{max } (\text{fst } w) (\text{Suc } (r t))) (\text{snd } w) @ \text{reusables } t w)$   
 ⟨proof⟩

**lemma** *valid\_append\_atomics\_reusables*:  $\llbracket 0 < \text{fst } w; \text{valid } t; l t \leq \text{fst } w; r t \leq \text{snd } w; \text{snd } w \leq \text{length } a s \rrbracket \implies$

$\forall t \in \text{set } (\text{atomics } (\text{max } (\text{fst } w) (\text{Suc } (r \ t))) (\text{snd } w) \ @ \ \text{reusables } t \ w). \ \text{valid } t$   
 $\langle \text{proof} \rangle$

**lemma** *append\_atomics\_reusables\_neq\_Nil*:  $\llbracket 0 < \text{fst } w; \text{fst } w \leq \text{snd } w; \text{valid } t; l \ t \leq \text{fst } w; r \ t \leq \text{snd } w \rrbracket \implies$   
 $\text{atomics } (\text{max } (\text{fst } w) (\text{Suc } (r \ t))) (\text{snd } w) \ @ \ \text{reusables } t \ w \neq []$   
 $\langle \text{proof} \rangle$

**lemma** *valid\_slide*:

**assumes**  $0 < \text{fst } w \ \text{fst } w \leq \text{snd } w \ \text{valid } t \ l \ t \leq \text{fst } w \ r \ t \leq \text{snd } w \ \text{snd } w \leq \text{length } as$

**shows**  $\text{valid } (\text{slide } t \ w) \wedge l \ (\text{slide } t \ w) = \text{fst } w \wedge r \ (\text{slide } t \ w) = \text{snd } w$

$\langle \text{proof} \rangle$

## 2.2 Correctness of the Sliding Window Algorithm

**lemma** *iterate\_eq\_map\_sum*:  $\llbracket \text{valid } t; \text{windows } as \ xs; (\text{case } xs \ \text{of } [] \Rightarrow \text{True} \mid x \ \# \ xs' \Rightarrow l \ t \leq \text{fst } x \wedge r \ t \leq \text{snd } x) \rrbracket \implies$

$\text{iterate } t \ xs = \text{map } (\lambda w. \ \text{sum } (\text{fst } w) (\text{snd } w)) \ xs$

$\langle \text{proof} \rangle$

**theorem** *correctness*:  $\text{windows } as \ ws \implies \text{sliding\_window} = \text{map } (\lambda w. \ \text{sum } (\text{fst } w) (\text{snd } w)) \ ws$   
 $\langle \text{proof} \rangle$

end

## 2.3 Summary of the Correctness Proof

We closely follow Basin et al.'s proof outline [1].

1. Lemma 1, the correctness result about the function *slide*, is formalized by *SWA.valid\_slide*. It follows from the following auxiliary facts:
  - Fact (a) is formalized by *SWA.adjacent\_reusables* and *SWA.valid\_reusables*.
  - Fact (b) is formalized by *SWA.adjacent\_atomics* and *SWA.valid\_atomics*.
  - Fact (c) is formalized by *SWA.valid\_fold\_combine\_Leaf*.
2. Theorem 2, the correctness result about the function *sliding\_window*, is formalized by *SWA.correctness*.

# 3 Alternative Slide Interface and Additional Operations

## 3.1 Alternative Slide Interface

The slide operation above takes the *entire* input sequence as a parameter. This is often impractical. We provide an alternative interface to the slide operation that takes only the *new* elements as a parameter.

**abbreviation** *atomic' where*

$\text{atomic}' \ as \ b \ idx \equiv \text{Node } b \ b \ (\text{Some } (nth \ as \ idx)) \ \text{Leaf } \ \text{Leaf}$

**abbreviation** *atomics' :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  'a tree list where*

$\text{atomics}' \ as \ i \ j \ sidx \equiv \text{map } (\lambda b. \ \text{atomic}' \ as \ b \ (b - sidx)) \ (\text{rev } [i \ ..< \ \text{Suc } j])$

**definition**  $slide' :: 'a :: semigroup\_add\ list \Rightarrow 'a\ tree \Rightarrow window \Rightarrow 'a\ tree$  **where**  
 $slide'\ as\ t\ w =$

(let  
 $ts = atomics'\ as\ (max\ (fst\ w)\ (Suc\ (r\ t)))\ (snd\ w)\ (Suc\ (r\ t));$   
 $ts' = reusables\ t\ w$   
in fold combine (ts @ ts') Leaf)

**lemma**  $slide\_eq\_slide'$ :

**assumes**  $0 < fst\ w\ fst\ w \leq snd\ w\ valid\ as\ t\ r\ t = length\ as\ l\ t \leq fst\ w\ r\ t \leq snd\ w\ snd\ w \leq length\ (as$   
@ as')

**shows**  $slide\ (as\ @\ as')\ t\ w = slide'\ as'\ t\ w$

*<proof>*

**lemma**  $sum\_eq\_sum\_append$ :  $\llbracket 0 < i; i \leq j; j \leq length\ as \rrbracket \Longrightarrow sum\ as\ i\ j = sum\ (as\ @\ as')\ i\ j$

*<proof>*

**lemma**  $well\_valued0\_append$ :  $\llbracket well\_shaped\ t; well\_valued0\ as\ t \rrbracket \Longrightarrow well\_valued0\ (as\ @\ as')\ t$

*<proof>*

**lemma**  $valid\_append$ :  $valid\ as\ t \Longrightarrow valid\ (as\ @\ as')\ t$

*<proof>*

**lemma**  $valid\_slide\_append$ :  $\llbracket 0 < fst\ w; fst\ w \leq snd\ w; valid\ as\ t; l\ t \leq fst\ w; r\ t \leq snd\ w; snd\ w \leq$   
 $length\ as + length\ as' \rrbracket \Longrightarrow$

$valid\ (as\ @\ as')\ (slide\ (as\ @\ as')\ t\ w) \wedge l\ (slide\ (as\ @\ as')\ t\ w) = fst\ w \wedge r\ (slide\ (as\ @\ as')\ t\ w) =$   
 $snd\ w$

*<proof>*

**theorem**  $valid\_slide'$ :

**assumes**  $0 < fst\ w\ fst\ w \leq snd\ w\ valid\ as\ t\ length\ as = r\ t\ length\ as' \geq snd\ w - r\ t\ l\ t \leq fst\ w\ r\ t \leq$   
 $snd\ w$

**shows**  $valid\ (as\ @\ as')\ (slide'\ as'\ t\ w) \wedge l\ (slide'\ as'\ t\ w) = fst\ w \wedge r\ (slide'\ as'\ t\ w) = snd\ w$

*<proof>*

## 3.2 Updating all Values in the Tree

So far, we have assumed that the sequence is fixed. However, under certain conditions, SWA can be applied even if the sequence changes. In particular, if a function that distributes over the associative operation is mapped onto the sequence, validity of the tree can be preserved by mapping the same function onto the tree using  $map\_tree$ .

**lemma**  $map\_tree\_eq\_Leaf\_iff$ :  $map\_tree\ f\ t = Leaf \longleftrightarrow t = Leaf$

*<proof>*

**lemma**  $l\_map\_tree\_eq\_l[simp]$ :  $l\ (map\_tree\ f\ t) = l\ t$

*<proof>*

**lemma**  $r\_map\_tree\_eq\_r[simp]$ :  $r\ (map\_tree\ f\ t) = r\ t$

*<proof>*

**lemma**  $val\_map\_tree\_neq\_None$ :  $val\ t \neq None \Longrightarrow val\ (map\_tree\ f\ t) \neq None$

*<proof>*

**lemma**  $well\_shaped\_map\_tree$ :  $well\_shaped\ t \Longrightarrow well\_shaped\ (map\_tree\ f\ t)$

*<proof>*

**lemma**  $fold\_distr$ :  $(\forall x\ y. f\ (x + y) = f\ x + f\ y) \Longrightarrow f\ (fold\ (+)\ list\ e) = fold\ (+)\ (map\ f\ list)\ (f\ e)$

*<proof>*

**lemma** *map\_rev\_map\_nth\_eq*:  $\forall x \in \text{set } xs. x < \text{length } as \implies \text{map } f (\text{rev } (\text{map } (!) \text{ as}) xs) = \text{rev } (\text{map } (!) (\text{map } f \text{ as}) xs)$

*<proof>*

**lemma** *f\_nth\_eq\_map\_f\_nth*:  $\llbracket as \neq []; \text{length } as \geq n \rrbracket \implies f (as ! (n - \text{Suc } 0)) = \text{map } f \text{ as} ! (n - \text{Suc } 0)$

*<proof>*

**lemma** *well\_valued0\_map\_map\_tree*:

$\llbracket \forall x y. f (x + y) = f x + f y; \text{well\_shaped } t; \text{well\_valued0 } as \text{ } t; r \text{ } t \leq \text{length } as; as \neq [] \rrbracket \implies \text{well\_shaped } (\text{map\_tree } f \text{ } t) \wedge \text{well\_valued0 } (\text{map } f \text{ as}) (\text{map\_tree } f \text{ } t)$

*<proof>*

**lemma** *valid\_map\_map\_tree*:

**assumes**  $\forall x y. f (x + y) = f x + f y$  *valid as t r t*  $\leq \text{length } as$

**shows** *valid (map f as) (map\_tree f t)*

*<proof>*

**lemma** *valid\_Nil\_iff*: *valid [] t*  $\longleftrightarrow t = \text{Leaf}$

*<proof>*

### 3.3 Updating the Rightmost Leaf of the Tree

We provide a function to update the rightmost leaf of the tree. This may be used in an online setting where the input sequence is not known in advance to update the latest observed element using the same associative operation used in SWA. We show that validity of the tree is preserved in this case.

**fun** *update\_rightmost* ::  $('a \Rightarrow 'a) \Rightarrow 'a \text{ tree} \Rightarrow 'a \text{ tree}$  **where**

*update\_rightmost Leaf = Leaf*

| *update\_rightmost f (Node i j a t u) = Node i j (map\_option f a) t (update\_rightmost f u)*

**lemma** *update\_rightmost\_eq\_Leaf\_iff*: *update\_rightmost f t = Leaf*  $\longleftrightarrow t = \text{Leaf}$

*<proof>*

**lemma** *l\_update\_rightmost\_eq\_l[simp]*:  $l (\text{update\_rightmost } f \text{ } t) = l \text{ } t$

*<proof>*

**lemma** *r\_update\_rightmost\_eq\_r[simp]*:  $r (\text{update\_rightmost } f \text{ } t) = r \text{ } t$

*<proof>*

**lemma** *val\_update\_rightmost\_neq\_None*:  $\text{val } t \neq \text{None} \implies \text{val } (\text{update\_rightmost } f \text{ } t) \neq \text{None}$

*<proof>*

**lemma** *well\_shaped\_update\_rightmost*:  $\text{well\_shaped } t \implies \text{well\_shaped } (\text{update\_rightmost } f \text{ } t)$

*<proof>*

**lemma** *sum\_eq\_sum\_prepend*:  $\llbracket 0 < i; i \leq j; \text{length } xs < i; \text{length } ys = \text{length } xs \rrbracket \implies \text{sum } (xs @ as) \text{ } i = \text{sum } (ys @ as) \text{ } i$

*<proof>*

**lemma** *well\_valued0\_prepend*:  $\llbracket \text{length } xs \leq l \text{ } t - 1; \text{length } ys = \text{length } xs; \text{well\_shaped } t; \text{well\_valued0 } (xs @ as) \text{ } t \rrbracket \implies \text{well\_valued0 } (ys @ as) \text{ } t$

*<proof>*

**lemma** *valid\_prepend*:  $\llbracket \text{length } xs \leq l \text{ } t - 1; \text{length } ys = \text{length } xs; \text{valid } (xs @ as) \text{ } t \rrbracket \implies \text{valid } (ys @ as) \text{ } t$

*<proof>*

**lemma** *take\_eq\_append\_take\_take\_drop*:  $m \leq n \implies \text{take } n \text{ } xs = \text{take } m \text{ } xs @ \text{take } (n-m) \text{ } (\text{drop } m \text{ } xs)$   
*<proof>*

**lemma** *well\_valued0\_take\_r*:  $\llbracket \text{well\_shaped } t; \text{well\_valued0 } as \text{ } t \rrbracket \implies \text{well\_valued0 } (\text{take } (r \ t) \ as) \ t$   
*<proof>*

**lemma** *valid\_take\_r*:  $\text{valid } as \ t \implies \text{valid } (\text{take } (r \ t) \ as) \ t$   
*<proof>*

**lemma** *well\_valued0\_butlast*:  $\llbracket \text{well\_shaped } t; \text{well\_valued0 } as \ t; r \ t < \text{length } as \rrbracket \implies \text{well\_valued0 } (\text{butlast } as) \ t$   
*<proof>*

**lemma** *well\_valued0\_append\_butlast\_lchild*:  $\llbracket \text{well\_shaped } t; \text{well\_valued0 } as \ t \rrbracket \implies \text{well\_valued0 } (\text{butlast } as @ [\text{last } as + x]) \ (\text{lchild } t)$   
*<proof>*

**lemma** *sum\_update\_rightmost*:  $\llbracket 0 < i; i \leq j; \text{length } as = j \rrbracket \implies \text{sum } as \ i \ j + x = \text{sum } (\text{butlast } as @ [\text{last } as + x]) \ i \ j$   
*<proof>*

**lemma** *well\_valued0\_update\_rightmost*:  $\llbracket \text{well\_shaped } t; \text{well\_valued0 } as \ t; \text{length } as = r \ t \rrbracket \implies \text{well\_valued0 } (\text{butlast } as @ [\text{last } as + x]) \ (\text{update\_rightmost } (\lambda a. a + x) \ t)$   
*<proof>*

**lemma** *valid\_update\_rightmost*:  $\llbracket \text{valid } as \ t; \text{length } as = r \ t \rrbracket \implies \text{valid } (\text{butlast } as @ [\text{last } as + x]) \ (\text{update\_rightmost } (\lambda a. a + x) \ t)$   
*<proof>*

## References

- [1] D. Basin, F. Klaedtke, and E. Zălinescu. Greedily computing associative aggregations on sliding windows. *Information Processing Letters*, 115(2):186 – 192, 2015.