

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

Lukas Heimes

Dmitriy Traytel

Joshua Schneider

March 19, 2025

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

1	Sliding Window Algorithm	1
2	Correctness	4
2.1	Correctness of the Slide Function	5
2.2	Correctness of the Sliding Window Algorithm	10
2.3	Summary of the Correctness Proof	10
3	Alternative Slide Interface and Additional Operations	11
3.1	Alternative Slide Interface	11
3.2	Updating all Values in the Tree	12
3.3	Updating the Rightmost Leaf of the Tree	13

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 ==> t ≠ Leaf
  by auto

primrec discharge :: 'a tree ⇒ '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
  fix a b c :: 'a option
  show a + b + c = a + (b + c)
  by (induct a b rule: plus_option.induct; cases c)
    (auto simp: algebra_simps)
qed

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: [t ≠ Leaf; u ≠ Leaf] ⇒ combine t u = Node (l t) (r u) (val t + val u)
(discharge t) u
proof -
  assume assms: t ≠ Leaf u ≠ Leaf
  from assms have combine t u = Node (l (Node (l t) (r t) (val t) (lchild t) (rchild t))) (r (Node (l u)
(r u) (val u) (lchild u) (rchild u))) (val (Node (l t) (r t) (val t) (lchild t) (rchild t)) + val (Node (l u) (r
u) (val u) (lchild u) (rchild u))) (discharge (Node (l t) (r t) (val t) (lchild t) (rchild t))) (Node (l u) (r u)
(val u) (lchild u) (rchild u)))
  by (metis (no_types) combine.simps(3) tree.exhaust_sel)
  with assms show ?thesis
    by simp
qed

lemma r_combine_non_Leaves: [t ≠ Leaf; u ≠ Leaf] ⇒ r (combine t u) = r u
by (simp add: combine_non_Leaves)

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)
  by auto

termination
  by (relation measure (λp. size (fst p)))
    (auto simp: lchild_def rchild_def split: tree.splits)

declare reusables.simps[simp del]

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

```

```

by (simp add: reusables.simps)

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]:
  [well_shaped t; l t < r t] ⇒ l (lchild t) = l t
  by (induct t) auto

lemma r_rchild_eq_r_if_well_shaped[simp]:
  [well_shaped t; l t < r t] ⇒ r (rchild t) = r t
  by (induct t) auto

lemma r_lchild_eq_l_rchild_if_well_shaped:
  [well_shaped t; l t < r t] ⇒ r (lchild t) = l (rchild t) − 1
  by (induct t) auto

lemma r_lchild_le_r: well_shaped t ⇒ r (lchild t) ≤ r t
proof (induct t)
  case (Node i j a t1 t2)
  then show ?case
  by auto
  (metis Suc_eq_plus1_left order.trans le_add2 tree.exhaust_sel well_shaped.simps(2))
qed simp

lemma well_shaped_lchild[simp]: well_shaped t ⇒ well_shaped (lchild t)
  by (induct t) auto

lemma well_shaped_rchild[simp]: well_shaped t ⇒ well_shaped (rchild t)
  by (induct t) auto

definition adjacent where
  adjacent w ts = (Leaf ∉ set ts ∧
    list_all2 (λt u. l t = Suc (r u)) (butlast ts) (tl ts) ∧
    (ts = [] ∨ (l (last ts) = fst w ∧ r (hd ts) = snd w)))

lemma adjacent_Nil[simp]: adjacent w []
  unfolding adjacent_def by simp

lemma adjacent_Cons: adjacent w (t # ts) =
  (t ≠ Leaf ∧ r t = snd w ∧ (case ts of [] ⇒ l t = fst w
    | u # us ⇒ adjacent (fst w, r u) ts ∧ l t = Suc (r u)))
  unfolding adjacent_def by (auto split: list.splits)

lemma adjacent_ConsI: [| t ≠ Leaf; r t = snd w;
  (case ts of [] ⇒ l t = fst w
    | u # us ⇒ adjacent (fst w, r u) ts ∧ l t = Suc (r u))|] ⇒
  adjacent w (t # ts)
  by (simp add: adjacent_Cons)

lemma adjacent_singleton: t ≠ Leaf ⇒ adjacent (l t, r t) [t]
  unfolding adjacent_def by simp

lemma append_Cons_eq_append_append: xs @ y # ys = xs @ [y] @ ys
  by simp

```

```

lemma list_all2_append_singletonI: [[list_all2 P xs ys; P x y] ==> list_all2 P (xs @ [x]) (ys @ [y])]
  by (simp add: list_all2_appendI)

lemma list_all2_Cons_append_singletonI: [[xs ≠ []; list_all2 P (x # butlast xs) ys; P (last xs) y] ==>
  list_all2 P (x # xs) (ys @ [y])
  using list_all2_append_singletonI by fastforce

lemma adjacent_appendI: [[0 < fst w; fst w ≤ snd w;
  (case us of [] ⇒ adjacent w ts
  | u # us' ⇒ adjacent (Suc (r u), snd w) ts ∧ adjacent (fst w, (case ts of [] ⇒ snd w | ts ⇒ r u)) (u
  # us'))] ==>
  adjacent w (ts @ us)
  unfolding adjacent_def butlast_append
  by (auto simp: intro: list_all2_Cons_append_singletonI
    list_all2_appendI[OF list_all2_Cons_append_singletonI, simplified] split: list.splits if_splits)

lemma adjacent_Cons_implies_adjacent: adjacent (a, b) (t # ts) ==> adjacent (a, l t - Suc 0) ts
  by (cases ts) (simp_all add: adjacent_def)

lemma (in semigroup_add) fold_add_add: fold (+) xs (x + y) = fold (+) xs x + y
  by (induct xs arbitrary: x) (auto simp: add_assoc[symmetric])

context
  fixes as :: 'a :: semigroup_add list
  and ws :: window list
begin

abbreviation atomic where
  atomic i ≡ Node i i (Some (nth as (i - 1))) Leaf Leaf

definition atomics :: nat ⇒ nat ⇒ 'a tree list where
  atomics i j ≡ map atomic (rev [i ..< Suc j])

definition slide :: 'a tree ⇒ window ⇒ '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 ⇒ window list ⇒ '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 ≡ fold (+) (rev (map (nth as) [i - 1 ..< j - 1])) (nth as (j - 1))

primrec well_valued0 :: 'a tree ⇒ bool where
  well_valued0 Leaf = True
  | well_valued0 (Node i j a t u) = (0 < i ∧ j ≤ length as ∧ (a ≠ None → a = Some (sum i j)) ∧
    well_valued0 t ∧ well_valued0 u ∧ (u = Leaf ∨ val u ≠ 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
  unfolding valid_def by auto

lemma add_sum:
  assumes i > 0 j  $\geq$  i k > j
  shows sum i j + sum (Suc j) k = sum i k
proof -
  have *: [i - 1 .. < k - 1] = [i - 1 .. < j] @ [j .. < k - 1]
  using assms upt_add_eq_append[of i - 1 j - 1 k - j]
  by (cases j) (auto simp: upt_conv_Cons)
  then show ?thesis using assms
  by (cases j) (auto simp: fold_add_add)
qed

lemma well_valued0_rchild_if_well_valued0[simp]: well_valued0 t  $\Longrightarrow$  well_valued0 (rchild t)
  by (induct t) auto

lemma well_valued0_lchild_if_well_valued0[simp]: well_valued0 t  $\Longrightarrow$  well_valued0 (lchild t)
  by (induct t) auto

lemma valid_rchild_if_valid: valid t  $\Longrightarrow$  valid (rchild t)
  by (metis tree.exhaust_sel tree.sel(9) valid_def well_shaped_rchild well_valued0.simps(2))

lemma val_eq_Some_sum_if_valid_neq_Leaf: [valid t; t  $\neq$  Leaf]  $\Longrightarrow$  val t = Some (sum (l t) (r t))
  by (auto simp: valid_def foldr_conv_fold)
  (metis One_nat_def option.distinct(1) option.inject tree.exhaust_sel well_valued0.simps(2))

```

2.1 Correctness of the Slide Function

```

lemma adjacent_atomics: adjacent (i, j) (atomics i j)
  unfolding adjacent_def atomics_def
  by (auto 0 1 simp: last_map last_rev nth_tl
    map_butlast[symmetric] list_all2_conv_all_nth nth_Cons' rev_nth nth_append)

lemma valid_atomics: [t  $\in$  set (atomics i j); 0 < i; j  $\leq$  length as]  $\Longrightarrow$  valid t
  unfolding atomics_def
  by (auto simp: valid_def)

lemma reusables_neq_Nil_if_well_shaped_and_overlapping:
  [well_shaped t; l t  $\leq$  fst w; r t  $\leq$  snd w; fst w  $\leq$  r t]  $\Longrightarrow$  reusables t w  $\neq$  []
  by (induction t w rule: reusables.induct) (simp add: reusables.simps Let_def)

lemma reusables_lchild_neq_Nil_under_some_conditions:
  [well_shaped t; l t  $\leq$  fst w; r t  $\leq$  snd w; fst w  $\neq$  l t; r t  $\geq$  fst w; l (rchild t) > fst w]  $\Longrightarrow$ 
  reusables (lchild t) w  $\neq$  []
  using r_lchild_eq_l_rchild_if_well_shaped[of t] r_lchild_le_r[of t]
  by (intro reusables_neq_Nil_if_well_shaped_and_overlapping) auto

lemma adjacent_reusables: [0 < fst w; well_shaped t; l t  $\leq$  fst w; r t  $\leq$  snd w]  $\Longrightarrow$ 
  adjacent (fst w, r t) (reusables t w)

```

```

proof (induction t w rule: reusables.induct)
  case (1 t w)
  show ?case
  proof (cases fst w > r t)
    case False
    with 1 show ?thesis
    proof (cases fst w = l t)
      case False
      then show ?thesis
      proof (cases fst w ≥ l (rchild t))
        case True
        with 1 <fst w ≠ l t> show ?thesis by (subst reusables.simps) auto
      next
      case False
      with 1 <fst w ≠ l t> <¬ fst w > r t> obtain x xs where *: reusables (lchild t) w = x # xs
        by (cases reusables (lchild t) w) (auto simp: reusables_lchild_neq_Nil_under_some_conditions)
      with 1(2–6) <fst w ≠ l t> <¬ fst w > r t> <¬ l (rchild t) ≤ fst w>
      have adjacent (fst w, r (lchild t)) (x # xs)
        by (simp add: adjacent_Cons_r_lchild_le_r_dual_order.trans)
      with 1 <fst w ≠ l t> <¬ fst w > r t> * show ?thesis
        by (subst reusables.simps)
          (auto simp add: Let_def adjacent_Cons_r_lchild_eq_l_rchild_if_well_shaped)
      qed
      qed (auto simp: reusables.simps intro!: adjacent_singleton)
      qed (auto simp: reusables.simps)
    qed
  lemma valid_rchild_if_well_valued0: [| well_shaped t; well_valued0 t |] ==> valid (rchild t)
    by (metis tree.exhaust_sel tree.sel(9) valid_def well_shaped_rchild well_valued0.simps(2))

  lemma valid_reusables_under_some_conditions:
    [| 0 < fst w; well_valued0 t; well_shaped t; l t < fst w; r t ≤ snd w |] ==>
    ∀ t' ∈ set (reusables t w). valid t'
  proof (induction t w rule: reusables.induct)
    case (1 t w)
    show ?case
    proof (cases fst w > r t)
      case False
      with 1 show ?thesis
      proof (cases fst w = l t)
        next
        case False
        then show ?thesis
        proof (cases fst w ≥ l (rchild t))
          case True
          with 1 <fst w ≠ l t> <¬ fst w > r t> have *: Ball (set (reusables (rchild t) w)) valid
            by (metis (no_types, lifting) ball_empty dual_order.strict_trans2 leI le_neq_implies_less
                list.set(1) r_rchild_eq_r_if_well_shaped reusables.simps set_ConsD valid_rchild_if_well_valued0 well_shaped_rchild
                well_valued0_rchild_if_well_valued0)
          from 1 <fst w ≠ l t> <¬ fst w > r t> True have reusables t w = reusables (rchild t) w
            by (subst reusables.simps)
              simp
          with 1 * show ?thesis
            by simp
        next
        case False
        with 1 <fst w ≠ l t> <¬ fst w > r t> have *: Ball (set (reusables (lchild t) w)) valid
          by (metis dual_order.strict_trans2 l_lchild_eq_l_if_well_shaped leI order_trans r_lchild_le_r

```

```

well_shaped_lchild well_valued0_lchild_if_well_valued0)
with 1 have valid_rchild: valid (rchild t)
  by (simp add: valid_rchild_if_well_valued0)
with 1 <fst w ≠ l t> ⊢ fst w > r t False have reusables t w = rchild t # reusables (lchild t) w
  by (subst reusables.simps) presburger
with 1 <fst w ≠ l t> ⊢ fst w > r t False * valid_rchild show ?thesis
  by (metis set_ConsD)
qed
qed (auto simp: reusables.simps)
qed (auto simp: reusables.simps)
qed

lemma valid_reusables:
assumes 0 < fst w valid t l t ≤ fst w r t ≤ snd w
shows ∀ t' ∈ set (reusables t w). valid t'
proof (cases l t < fst w)
  case True
  with assms show ?thesis
    using valid_def valid_reusables_under_some_conditions by blast
next
  case False
  with assms show ?thesis
    by (simp add: reusables.simps)
qed

lemma combine_valid_Nodes_aux:
assumes prems: 0 < l a a ≠ Leaf z ≠ Leaf l z = Suc (r a) well_shaped a well_shaped z
  well_valued0 a val a = Some va well_valued0 z val z = Some vz
shows va + vz = fold (+) (rev (map (!) as) [l a - Suc 0..<r z - Suc 0])) (as ! (r z - Suc 0))
proof -
  from prems have l a > 0 by simp
  moreover from prems have r a ≥ l a
    by (metis tree.collapse well_shaped.simps(2))
  moreover have r z > r a
    by (metis Suc_le_lessD prems(3) prems(4) prems(6) tree.collapse well_shaped.simps(2))
  ultimately have *: sum (l a) (r a) + sum (Suc (r a)) (r z) = sum (l a) (r z)
    by (rule add_sum)
  from prems have va = sum (l a) (r a)
    by (metis option.discI option.inject tree.collapse well_valued0.simps(2))
  moreover from prems have vz = sum (Suc (r a)) (r z)
    by (metis option.discI option.inject prems(2) prems(3) prems(8) prems(9) tree.collapse well_valued0.simps(2))
  moreover have fold (+) (rev (map (!) as) [l a - Suc 0..<r z - Suc 0])) (as ! (r z - Suc 0)) = sum
    (l a) (r z)
    by simp
  ultimately show ?thesis
    using * by (auto simp: add_sum)
qed

lemma discharge_is_Leaf[simp]: discharge a = Leaf ↔ a = Leaf
  by (cases a) auto

lemma well_shaped_discharge[simp]: well_shaped a ⇒ well_shaped (discharge a)
  by (cases a) auto

lemma well_valued0_discharge[simp]: well_valued0 a ⇒ well_valued0 (discharge a)
  by (cases a) auto

```

```

lemma l_discharge[simp]: l (discharge a) = l a
  by (cases a) auto

lemma r_discharge[simp]: r (discharge a) = r a
  by (cases a) auto

lemma well_shaped_lr: well_shaped a ==> l a ≤ r a
  by (cases a) auto

lemma well_valued0_r: well_valued0 a ==> a ≠ Leaf ==> r a ≤ length as
  by (cases a) auto

lemma valid_combine_if_valid: [|0 < l a; valid a; valid z; a ≠ Leaf; z ≠ Leaf; l z = Suc (r a)|] ==>
  valid (combine a z)
  by (force simp add: valid_def combine_non_Leaves combine_valid_Nodes_aux
    dest: well_shaped_lr well_valued0_r)

lemma combine_neq_Leaf_if_both_non_Leaf: [|a ≠ Leaf; z ≠ Leaf|] ==>
  combine a z ≠ Leaf
  by (simp add: combine_non_Leaves)

lemma valid_fold_combine: [|0 < fst w; ts = h # ts'; ∀ t ∈ set ts. valid t; adjacent (fst w, l h - 1) ts';
  valid z; z ≠ Leaf; l z = (case ts' of [] ⇒ fst w | t1 # ts'' ⇒ Suc (r t1)); r z = snd w|] ==>
  valid (fold combine ts' z) ∧
  l (fold combine ts' z) = fst w ∧ r (fold combine ts' z) = snd w
proof (induction ts' arbitrary: z ts h)
  case Nil
  then show ?case by simp
next
  case (Cons a ts')
  moreover from Cons(3–4) have valid a by simp
  moreover from Cons(5) have adjacent (fst w, l a - Suc 0) ts'
    unfolding adjacent_Cons by (auto split: list.splits)
  moreover from Cons(2–6,8) have valid (combine a z)
    unfolding adjacent_Cons
    by (intro valid_combine_if_valid) (auto split: list.splits)
  moreover from Cons(5,7) have combine a z ≠ Leaf
    by (intro combine_neq_Leaf_if_both_non_Leaf) (simp_all add: adjacent_Cons)
  moreover from Cons(5,7) have l (combine a z) = (case ts' of [] ⇒ fst w | t1 # ts'' ⇒ Suc (r t1))
    by (auto simp add: adjacent_def combine_non_Leaves split: list.splits)
  moreover from Cons(5,7,9) have r (combine a z) = snd w
    by (subst r_combine_non_Leaves) (auto simp add: adjacent_def)
  ultimately show ?case
    by simp
qed

lemma valid_fold_combine_Leaf:
  assumes 0 < fst w ts = h # ts' ∀ t ∈ set ts. valid t adjacent w ts
  shows valid (fold combine ts Leaf) ∧
  l (fold combine ts Leaf) = fst w ∧ r (fold combine ts Leaf) = snd w
proof -
  from assms(2) have fold_combine ts Leaf = fold_combine ts' h
    by simp
  moreover have valid (fold combine ts' h) ∧
    l (fold combine ts' h) = fst w ∧ r (fold combine ts' h) = snd w
  proof (rule valid_fold_combine)

```

```

from assms show 0 < fst w ts = h # ts' ∀ t ∈ set ts. valid t valid h h ≠ Leaf r h = snd w
  l h = (case ts' of [] ⇒ fst w | t1 # ts'' ⇒ Suc (r t1))
  by (simp_all add: adjacent_Cons list.case_eq_if)
from assms show adjacent (fst w, l h - 1) ts'
  by (metis One_nat_def adjacent_Cons_implies_adjacent prod.collapse)
qed
ultimately show ?thesis by simp
qed

lemma adjacent_atomics_nonempty_reusables:
fixes x :: 'a tree and xs :: 'a tree list
assumes a1: 0 < fst w
  and a2: l t ≤ fst w
  and a3: r t ≤ snd w
  and a4: valid t
  and a5: reusables t w = x # xs
shows adjacent (Suc (r x), snd w) (atomics (max (fst w) (Suc (r t))) (snd w))
proof -
  have f6: ∀ p ts. adjacent p ts = ((Leaf::'a tree) ∉ set ts ∧ list_all2 (λt ta. l t = Suc (r ta)) (butlast ts)
  (tl ts) ∧ (ts = [] ∨ l (last ts) = fst p ∧ r (hd ts) = snd p))
    using adjacent_def by blast
  then have f7: Leaf ∉ set (atomics (max (fst w) (Suc (r t))) (snd w)) ∧ list_all2 (λt ta. l t = Suc (r ta)) (butlast (atomics (max (fst w) (Suc (r t))) (snd w)))
  (tl (atomics (max (fst w) (Suc (r t))) (snd w))) ∧ (atomics (max (fst w) (Suc (r t))) (snd w)) = [] ∨ l (last (atomics (max (fst w) (Suc (r t))) (snd w)))
  = fst (max (fst w) (Suc (r t)), snd w) ∧ r (hd (atomics (max (fst w) (Suc (r t))) (snd w))) = snd (max (fst w) (Suc (r t)), snd w))
    using adjacent_atomics by presburger
  have adjacent (fst w, r t) (reusables t w)
    using a4 a3 a2 a1 by (simp add: adjacent_reusables valid_def)
  then have f8: r x = r t
    using a5 by (simp add: adjacent_Cons)
  have max (fst w) (Suc (r t)) = Suc (r t)
    using a5 by (metis (no_types) Suc_n_not_le_n list.simps(3) max.bounded_iff max_def_raw not_le_imp_less
  reusables.simps)
  then show ?thesis
    using f8 f7 f6 by presburger
qed

lemma adjacent_Cons_r: adjacent (a, r t) (x # xs) ⇒ adjacent (a, r x) (x # xs)
by (simp add: adjacent_Cons)

lemma adjacent_Cons_r2:
adjacent (fst w, r t) (x # xs) ⇒ 0 < fst w ⇒ fst w ≤ snd w ⇒ r t ≤ snd w ⇒
atomics (max (fst w) (Suc (r t))) (snd w) = [] ⇒
adjacent w (x # xs)
by (metis (no_types, lifting) atomics_def adjacent_def append_is_Nil_conv diff_Suc_Suc_diff_zero
le_Suc_eq list.simps(3) map_is_Nil_conv max_Suc_Suc max_def_raw prod.sel(1) prod.sel(2) rev_is_Nil_conv
upt.simps(2))

lemma adjacent_append_atomics_reusables:
[| 0 < fst w; fst w ≤ snd w; valid t; l t ≤ fst w; r t ≤ snd w |] ⇒
adjacent w (atomics (max (fst w) (Suc (r t))) (snd w) @ reusables t w)
using adjacent_atomics_nonempty_reusables[of w t] reusables_neq Nil_if_well_shaped_and_overlapping[of
t w]
  adjacent_atomics[of fst w snd w] adjacent_reusables[of w t]
by (intro adjacent_appendI) (auto simp: valid_def max.absorb1 atomize_not
elim: adjacent_Cons_r adjacent_Cons_r2 split: list.splits nat.splits)

```

```

lemma valid_append_atomics_reusables: [|0 < fst w; valid t; l t ≤ fst w; r t ≤ snd w; snd w ≤ length as|] ==>
  ∀ t ∈ set (atomics (max (fst w) (Suc (r t))) (snd w) @ reusables t w). valid t
  by (auto simp only: set_append valid_reusables dest: valid_atomics split: if_splits)

lemma append_atomics_reusables_neq_Nil: [|0 < fst w; fst w ≤ snd w; valid t; l t ≤ fst w; r t ≤ snd w|] ==>
  atomics (max (fst w) (Suc (r t))) (snd w) @ reusables t w ≠ []
  by (simp add: reusables_neq_Nil_if_well_shaped_and_overlapping valid_def atomics_def)

lemma valid_slide:
  assumes 0 < fst w fst w ≤ snd w valid t l t ≤ fst w r t ≤ snd w snd w ≤ length as
  shows valid (slide t w) ∧ l (slide t w) = fst w ∧ r (slide t w) = snd w
proof -
  from assms have non_empty: atomics (max (fst w) (Suc (r t))) (snd w) @ reusables t w ≠ []
    using append_atomics_reusables_neq_Nil by blast
  from assms have adjacent: adjacent w (atomics (max (fst w) (Suc (r t))) (snd w) @ reusables t w)
    using adjacent_append_atomics_reusables by blast
  from assms have valid: ∀ t ∈ set (atomics (max (fst w) (Suc (r t))) (snd w) @ reusables t w). valid t
    using valid_append_atomics_reusables by blast
  have *: slide t w = fold_combine (atomics (max (fst w) (Suc (r t))) (snd w) @ reusables t w) Leaf
    by (simp add: slide_def)
  from assms(1) non_empty adjacent valid show valid (slide t w) ∧ l (slide t w) = fst w ∧ r (slide t w)
  = snd w
    unfolding * neq_Nil_conv using valid_fold_combine_Leaf by blast
qed

```

2.2 Correctness of the Sliding Window Algorithm

```

lemma iterate_eq_map_sum: [|valid t; windows as xs; (case xs of [] ⇒ True | x # xs' ⇒ l t ≤ fst x ∧ r t ≤ snd x)|] ==>
  iterate t xs = map (λw. sum (fst w) (snd w)) xs
  by (induction xs arbitrary: t)
    (auto simp: valid_slide windows_def window_def val_eq_Some_sum_if_valid_neq_Leaf_neq_Leaf_if_l_gt0
    split: list.split)

```

```

theorem correctness: windows as ws ==> sliding_window = map (λw. sum (fst w) (snd w)) ws
  by (auto simp only: sliding_window_def intro!: iterate_eq_map_sum)
    (auto simp: valid_def split: list.split)

```

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**

atomic' as *b idx* \equiv Node *b b* (*Some (nth as idx)*) Leaf Leaf

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

atomics' as *i j sidx* \equiv map (λb . *atomic'* as *b* (*b - sidx*)) (*rev [i ..< 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 (cases *r t = snd w*)

case True

with assms have *: *atomics' (as @ as') (max (fst w) (Suc (r t))) (snd w) 1 = []*

by simp

from True assms have *atomics' as' (max (fst w) (Suc (r t))) (snd w) (Suc (r t)) = []*

by simp

with * show ?thesis

unfolding *slide_def slide'_def atomics_def* by simp

next

case False

with assms have *r t < snd w*

by simp

with assms have *atomic' (as @ as') (snd w) (snd w - Suc 0) = atomic' as' (snd w) (snd w - Suc (length as))*

by (simp add: leD nth_append)

with assms have *: $\forall i. i \in set ([max (fst w) (Suc (length as))..< snd w]) \longrightarrow atomic' (as @ as') i (i - Suc 0) = atomic' as' i (i - Suc (length as))$

by (auto simp: nth_append)

then have map ($\lambda i. atomic' (as @ as') i (i - Suc 0)$) (*rev [max (fst w) (Suc (length as))..< snd w]*) = map ($\lambda b. atomic' as' b (b - Suc (r t))$) (*rev [max (fst w) (Suc (r t))..< snd w]*)

by (simp add: assms(4))

with assms have *atomics' (as @ as') (max (fst w) (Suc (r t))) (snd w) 1 = atomics' as' (max (fst w) (Suc (r t))) (snd w) (Suc (r t))*

by (auto simp: nth_append)

then show ?thesis

unfolding *slide_def slide'_def atomics_def*

by presburger

qed

lemma *sum_eq_sum_append*: $\llbracket 0 < i; i \leq j; j \leq length as \rrbracket \implies sum as i j = sum (as @ as') i j$

proof –

assume assms: $0 < i$ $i \leq j$ $j \leq length as$

then have *: *rev (map (!) as) [i - Suc 0..< j - Suc 0] = rev (map (!) (as @ as')) [i - Suc 0..< j -*

```

Suc 0])
  by (auto simp: nth_append)
from assms have as ! (j - Suc 0) = (as @ as') ! (j - Suc 0)
  by (auto simp: nth_append)
then show ?thesis
  by (simp add: *)
qed

lemma well_valued0_append: [|well_shaped t; well_valued0 as t|] ==> well_valued0 (as @ as') t
proof (induction t)
  case (Node i j a t1 t2)
  then show ?case
    using sum_eq_sum_append
    by (auto 4 0)
qed simp

lemma valid_append: valid as t ==> valid (as @ as') t
unfolding valid_def
by (auto intro: well_valued0_append)

lemma valid_slide_append: [|0 < fst w; fst w ≤ snd w; valid as t; l t ≤ fst w; r t ≤ snd w; snd w ≤ length as + length as'|] ==>
  valid (as @ as') (slide (as @ as') t w) ∧ l (slide (as @ as') t w) = fst w ∧ r (slide (as @ as') t w) = snd w
  by (auto simp: valid_append valid_slide)

theorem valid_slide':
assumes 0 < fst w fst w ≤ snd w valid as t length as = r t length as' ≥ snd w - r t l t ≤ fst w r t ≤ snd w
shows valid (as @ as') (slide' as' t w) ∧ l (slide' as' t w) = fst w ∧ r (slide' as' t w) = snd w
proof -
  from assms have valid (as @ as') (slide (as @ as') t w) ∧ l (slide (as @ as') t w) = fst w ∧ r (slide (as @ as') t w) = snd w
    using valid_slide_append by (metis add_le_cancel_left le_add_diff_inverse)
  with assms show ?thesis
    using slide_eq_slide' by (metis add_le_cancel_left le_add_diff_inverse length_append)
qed

```

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 ↔ t = Leaf
  by simp

lemma l_map_tree_eq_l[simp]: l (map_tree f t) = l t
  by (cases t)
    (auto split: option.splits)

lemma r_map_tree_eq_r[simp]: r (map_tree f t) = r t
  by (cases t)
    (auto split: option.splits)

lemma val_map_tree_neq_None: val t ≠ None ==> val (map_tree f t) ≠ None
  by (cases t) auto

```

```

lemma well_shaped_map_tree: well_shaped t ==> well_shaped (map_tree f t)
  by (induction t)
    (auto split: option.split)

lemma fold_distr: (∀ x y. f (x + y) = f x + f y) ==> f (fold (+) list e) = fold (+) (map f list) (f e)
  by (induction list arbitrary: e) auto

lemma map_rev_map_nth_eq: ∀ x ∈ set xs. x < length as ==> map f (rev (map ((!) as) xs)) = rev (map
((!) (map f as)) xs)
  by (simp add: rev_map)

lemma f_nth_eq_map_f_nth: [|as ≠ []; length as ≥ n|] ==> f (as ! (n - Suc 0)) = map f as ! (n - Suc
0)
  by (cases n = length as) auto

lemma well_valued0_map_map_tree:
  [|∀ x y. f (x + y) = f x + f y; well_shaped t; well_valued0 as t; r t ≤ length as; as ≠ []|] ==>
  well_shaped (map_tree f t) ∧ well_valued0 (map f as) (map_tree f t)
proof (rule conjI[OF well_shaped_map_tree], assumption, induction t)
  case (Node i j a t1 t2)
  then have map f (rev (map ((!) as) [|l t1 - Suc 0.. < r t2 - Suc 0|])) =
    rev (map ((!) (map f as)) [|l t1 - Suc 0.. < r t2 - Suc 0|])
    by (subst map_rev_map_nth_eq) auto
  moreover from Node have r t1 ≤ length as
    by (cases t1) auto
  ultimately show ?case using Node(3–7)
    by (auto simp: fold_distr[of f] val_map_tree_neq_None
      well_shaped_map_tree intro!: Node(1,2))
qed simp

lemma valid_map_map_tree:
  assumes ∀ x y. f (x + y) = f x + f y valid as t r t ≤ length as
  shows valid (map f as) (map_tree f t)
proof (cases as ≠ [])
  case True
  with assms show ?thesis
    by (metis map_tree_eq_Leaf_iff val_map_tree_neq_None valid_def well_valued0_map_map_tree)
next
  case False
  with assms show ?thesis
    by (metis le_zero_eq list.size(3) tree.exhaust_sel map_tree_eq_Leaf_iff valid_Leaf valid_def well_shaped.simps(2)
      well_valued0.simps(2))
qed

lemma valid_Nil_iff: valid [] t ↔ t = Leaf
  unfolding valid_def
proof
  assume well_shaped t ∧ well_valued [] t
  then show t = Leaf
    by (metis le_neq_implies_less list.size(3) not_less_zero tree.collapse well_shaped.simps(2) well_valued0.simps(2))
qed simp

```

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 ⇒ 'a) ⇒ 'a tree ⇒ 'a 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 ↔ t = Leaf
  by (cases t)
    (auto split: option.splits)

lemma l_update_rightmost_eq_l[simp]: l (update_rightmost f t) = l t
  by (cases t)
    (auto split: option.splits)

lemma r_update_rightmost_eq_r[simp]: r (update_rightmost f t) = r t
  by (cases t)
    (auto split: option.splits)

lemma val_update_rightmost_neq_None: val t ≠ None ⇒ val (update_rightmost f t) ≠ None
  by (cases t) auto

lemma well_shaped_update_rightmost: well_shaped t ⇒ well_shaped (update_rightmost f t)
  by (induction t)
    (auto simp: update_rightmost_eq_Leaf_iff split: option.split)

lemma sum_eq_sum_prepend: [|0 < i; i ≤ j; length xs < i; length ys = length xs|] ⇒ sum (xs @ as) i
  j = sum (ys @ as) i j
  proof –
    assume assms: 0 < i i ≤ j length xs < i length ys = length xs
    then have *: rev (map ((!) (xs @ as)) [i - Suc 0 .. < j - Suc 0]) = rev (map ((!) (ys @ as)) [i - Suc 0 .. < j - Suc 0])
      by (auto simp: nth_append)
    from assms have (xs @ as) ! (j - Suc 0) = (ys @ as) ! (j - Suc 0)
      by (auto simp: nth_append)
    then show ?thesis
      by (simp add: *)
  qed

lemma well_valued0_prepend: [|length xs ≤ l t - 1; length ys = length xs; well_shaped t; well_valued0 (xs @ as) t|] ⇒ well_valued0 (ys @ as) t
  proof (induction t)
    case (Node i j a t1 t2)
    then have well_valued0_t1: well_valued0 (ys @ as) t1
      by auto
    from Node have well_valued0 (ys @ as) t2
    proof (cases a)
      case None
      with Node show ?thesis
        by auto
        (metis One_nat_def Suc_pred diff_Suc_1 le_Suc_eq le_Suc_ex trans_le_add1 tree.exhaust_sel well_shaped.simps(2))
    next
      case (Some a')
      with Node show ?thesis
        by auto
        (metis One_nat_def diff_Suc_1 diff_le_self le_trans tree.exhaust_sel well_shaped.simps(2))
    qed
    with Node well_valued0_t1 show ?case
    proof –

```

```

have f1:  $0 < i \wedge j \leq \text{length } (\text{xs} @ \text{as}) \wedge (a = \text{None} \vee a = \text{Some } (\text{SWA.sum } (\text{xs} @ \text{as}) \ i \ j)) \wedge$ 
well_valued0 (xs @ as) t1  $\wedge$  well_valued0 (xs @ as) t2  $\wedge$  (t2 = Leaf  $\vee$  val t2 ≠ None)
  by (meson well_valued0 (xs @ as) (Node i j a t1 t2) well_valued0.simps(2))
then have f2:  $j \leq \text{length } (\text{ys} @ \text{as})$ 
  by (simp add: length ys = length xs)
have a = None  $\vee$  a = Some (SWA.sum (ys @ as) i j)
  using f1 by (metis Node.simps(1) Node.simps(2) Node.simps(3) One_nat_def Suc_pred le_imp_less_Suc
sum_eq_sum_prepend tree.sel(2) well_shaped.simps(2))
then show ?thesis
  using f2 f1 well_valued0 (ys @ as) t2 well_valued0.simps(2) well_valued0_t1 by blast
qed
qed simp

lemma valid_prepend:  $[\text{length } \text{xs} \leq l \ t - 1; \ \text{length } \text{ys} = \text{length } \text{xs}; \ \text{valid } (\text{xs} @ \text{as}) \ t] \implies \text{valid } (\text{ys} @ \text{as}) \ t$ 
proof -
  unfolding valid_def
  by (auto intro: well_valued0_prepend)

lemma take_eq_append_take_take_drop:  $m \leq n \implies \text{take } n \ \text{xs} = \text{take } m \ \text{xs} @ \text{take } (n-m) \ (\text{drop } m \ \text{xs})$ 
proof (induction xs)
  case (Cons a xs)
  then show ?case
    by (metis le_add_diff_inverse take_add)
qed simp

lemma well_valued0_take_r:  $[\text{well_shaped } t; \ \text{well_valued0 as } t] \implies \text{well_valued0 } (\text{take } (r \ t) \ \text{as}) \ t$ 
proof (induction t)
  case (Node i j a t1 t2)
  from Node have well_valued0_t1: well_valued0 (take j as) t1
  proof -
    from Node have r t1  $\leq j$ 
      using r_lchild_le_r by (metis tree.sel(4) tree.sel(8))
    with Node have take j as = take (r t1) as @ take (j - r t1) (drop (r t1) as)
      using take_eq_append_take_take_drop[of r t1 j as] by simp
    with Node show ?thesis
      by (auto intro!: well_valued0_append)
  qed
  from Node have well_valued0_t2: well_valued0 (take j as) t2
  by auto
  from Node have sum_eq: fold (+) (rev (map ((!) as) [l t1 - Suc 0..<r t2 - Suc 0])) (as ! (r t2 - Suc 0)) =
    fold (+) (rev (map ((!) (take (r t2) as)) [l t1 - Suc 0..<r t2 - Suc 0])) (as ! (r t2 - Suc 0))
    by (intro arg_cong[where f=λxs. fold _ xs _]) auto
  from Node well_valued0_t1 well_valued0_t2 sum_eq show ?case
  by auto
qed simp

lemma valid_take_r:  $\text{valid as } t \implies \text{valid } (\text{take } (r \ t) \ \text{as}) \ t$ 
proof -
  unfolding valid_def
  by (auto intro: well_valued0_take_r)

lemma well_valued0_butlast:  $[\text{well_shaped } t; \ \text{well_valued0 as } t; \ r \ t < \text{length as}] \implies \text{well_valued0 } (\text{butlast as}) \ t$ 
proof (induction t)
  case (Node i j a t1 t2)
  then have r_le_length:  $j \leq \text{length } (\text{butlast as})$ 
    by simp
  from Node have i > 0

```

```

    by simp
with r_le_length Node have sum_eq: sum as i j = sum (butlast as) i j
    by (metis append_butlast_last_id le_zero_eq less_imp_le_nat list.size(3) neq_iff
        sum_eq_sum_append well_shaped.simps(2))
from Node have r t1 < length as
    by (metis dual_order.strict_trans2 r_lchild_le_r tree.sel(8))
with Node r_le_length show ?case
    by (metis (no_types, lifting) dual_order.strict_iff_order sum_eq tree.sel(10) tree.sel(4)
        well_shaped.simps(2) well_shaped_rchild well_valued0.simps(2))
qed simp

lemma well_valued0_append_butlast_lchild: [well_shaped t; well_valued0 as t] ==>
    well_valued0 (butlast as @ [last as + x]) (lchild t)
proof (cases t)
    case (Node i j a t1 t2)
    assume assms: well_shaped t well_valued0 as t
    from Node assms have r t1 < length as
        by (fastforce dest: well_shaped_lr)
    with Node assms show ?thesis
        by (auto simp: well_valued0_butlast well_valued0_append)
qed simp

lemma sum_update_rightmost: [|0 < i; i ≤ j; length as = j|] ==>
    sum as i j + x = sum (butlast as @ [last as + x]) i j
by (cases as)
    (auto simp: nth_append[abs_def] fold_add_add last_conv_nth nth_cons' nth_butlast
    intro!: arg_cong2[where f=(+)] arg_cong[where f=λxs. fold _ xs _])

lemma well_valued0_update_rightmost: [well_shaped t; well_valued0 as t; length as = r t] ==>
    well_valued0 (butlast as @ [last as + x]) (update_rightmost (λa. a + x) t)
proof (induction t)
    case Leaf
    then show ?case
        by (auto simp add: valid_Leaf)
next
    case (Node i j a t1 t2)
    moreover
    from Node have well_valued0_t1: well_valued0 (butlast as @ [last as + x]) t1
        using well_valued0_append_butlast_lchild by (metis tree.sel(8))
    moreover
    from Node have well_valued0_t2:
        well_valued0 (butlast as @ [last as + x]) (update_rightmost ((λa. a + x)) t2)
        by (metis ‹well_valued0 (butlast as @ [last as + x]) t1› dual_order.strict_iff_order
            tree.sel(4) update_rightmost.simps(1) well_shaped.simps(2) well_valued0.simps(2))
    ultimately show ?case
        using sum_update_rightmost
        by (cases a) (auto simp: val_update_rightmost_neq_None)
qed

lemma valid_update_rightmost: [|valid as t; length as = r t|] ==>
    valid (butlast as @ [last as + x]) (update_rightmost (λa. a + x) t)
unfolding valid_def
using well_valued0_update_rightmost
by (metis update_rightmost.simps(1) val_update_rightmost_neq_None well_shaped_update_rightmost)

```

References

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