Much Ado about Two

By Sascha Böhme

March 17, 2025

Abstract

This article is an Isabelle formalisation of a paper with the same. In a similar way as Knuth's 0-1-principle for sorting algorithms, that paper develops a "0-1-2-principle" for parallel prefix computations.

Contents

1 Much Ado about Two

Due to Donald E. Knuth, it is known for some time that certain sorting functions for lists of arbitrary types can be proved correct by only showing that they are correct for boolean lists ([3], see also [2]). This reduction idea, i.e. reducing a proof for an arbitrary type to a proof for a fixed type with a fixed number of values, has also instances in other fields. Recently, in [5], a similar result as Knuth's 0-1-principle is explained for the problem of parallel prefix computation [1]. That is the task to compute, for given $x_1, \ldots, x_n$ and an associative operation $\oplus$, the values x_1, $x_1 \oplus x_2$, …, $x_1 \oplus x_2 \oplus \cdots \oplus x_n$. There are several solutions which optimise this computation, and an obvious question is to ask whether these solutions are correct. One way to answer this question is given in [5]. There, a "0-1-2-principle" is proved which relates an unspecified solution of the parallel prefix computation, expressed as a function *candidate*, with *scanl1*, a functional representation of the parallel prefix computation. The essence proved in the mentioned paper is as follows: If *candidate* and *scanl1* behave identical on all lists over a type which has three elements, then *candidate* is semantically equivalent to *scanl1*, that is, *candidate* is a correct solution of the parallel prefix computation.

Although it seems that nearly nothing is known about the function *candidate*, it turns out that the type of *candidate* already suffices for the proof of the paper's result. The key is relational parametricity [4] in the form of a free theorem [6]. This, some rewriting and a few properties about list-processing functions thrown in allow to proof the "0-1-2-principle".

The paper first shows some simple properties and derives a specialisation of the free theorem. The proof of the main theorem itself is split up in two parts. The first, and considerably more complicated part relates lists over a type with three values to lists of integer lists. Here, the paper uses several figures to demonstrate and shorten several proofs. The second part then relates lists of integer list with lists over arbitrary types, and consists of applying the free theorem and some rewriting. The combination of these two parts then yields the theorem.

Th article at hand formalises the proofs given in [5], which is called here "the original paper". Compared to that paper, there are several differences in this article. The major differences are listed below. A more detailed collection follows thereafter.

- The original paper requires lists to be non-empty. Eventhough lists in Isabelle may also be empty, we stick to Isabelle's list datatype instead of declaring a new datatype, due to the huge, already existing theory about lists in Isabelle. As a consequence, however, several modifications become necessary.
- The figure-based proofs of the original paper are replaced by formal proofs. This forms a major part of this article (see Section 6).
- Instead of integers, we restrict ourselves to natural numbers. Thus, several conditions can be simplified since every natural number is greater than or equal to *0*. This decision has no further influence on the proofs because they never consider negative integers.

- Mainly due to differences between Haskell and Isabelle, certain notations are different here compared to the original paper. List concatenation is denoted by @ instead of ++, and in writing down intervals, we use $[\mathit{0}..{<}k + \mathit{1}]$ instead of $[\mathit{0}..k]$. Moreover, we write f instead of $\oplus$ and g instead of $\otimes$. Functions mapping an element of the three-valued type to an arbitrary type are denoted by h.

Whenever we use lemmas from already existing Isabelle theories, we qualify them by their theory name. For example, instead of *map-map*, we write *List.map-map* to point out that this lemma is taken from Isabelle's list theory.

The following comparison shows all differences of this article compared to the original paper. The items below follow the structure of the original paper (and also this article's structure). They also highlight the challenges which needed to be solved in formalising the original paper.

- Introductions of several list functions (e.g. *length*, *map*, *take*) are dropped. They exist already in Isabelle's list theory and are be considered familiar to the reader.

- The free theorem given in Lemma 1 of the original paper is not sufficient for later proofs, because the assumption is not appropriate in the context of Isabelle's lists, which may also be empty. Thus, here, Lemma 1 is a derived version of the free theorem given as Lemma 1 in the original paper, and some additional proof-work is done.

- Before proceeding in the original paper's way, we state and proof additional lemmas, which are not part of Isabelle's libraries. These lemmas are not specific to this article and may also be used in other theories.

- Laws 1 to 8 and Lemma 2 of the original paper are explicitly proved. Most of the proofs follow directly from existing results of Isabelle's list theory. To proof Law 7, Law 8 and Lemma 2, more work was necessary, especially for Law 8.

- Lemma 3 and its proof are nearly the same here as in the original paper. Only the additional assumptions of Lemma 1, due to Isabelle's list datatype, have to be shown.

- Lemma 4 is split up in several smaller lemmas, and the order of them tries to follow the structure of the original paper's Lemma 4.

  For every figure of the original paper, there is now one separate proof. These proofs constitute the major difference in the structure of this article compared to the original paper.

  The proof of Lemma 4 in the original paper concludes by combining the results of the figure-based proofs to a non-trivial permutation property.

These three sentences given in the original paper are split up in five separate lemmas and according proofs, and therefore, they as well form a major difference to the original paper.

- Lemma 5 is mostly identical to the version in the original paper. It has one additional assumption required by Lemma 4. Moreover, the proof is slightly more structured, and some steps needed a bit more argumentation than in the original paper.

- In principle, Proposition 1 is identical to the according proposition in the original paper. However, to fulfill the additional requirement of Lemma 5, an additional lemma was proved. This, however, is only necessary, because we use Isabelle's list type which allows lists to be empty.

- Proposition 2 contains one non-trivial step, which is proved as a seperate lemma. Note that this is not due to any decisions of using special datatypes, but inherent in the proof itself. Apart from that, the proof is identical to the original paper's proof of Proposition 2.

- The final theorem is, as in the original paper, just a combination of Proposition 1 and Proposition 2. Only the assumptions are extended due to Isabelle's list datatype.

2 Basic definitions

fun $\mathit{foldl1} :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ \mathit{list} \Rightarrow 'a$
where
$\mathit{foldl1}\ f\ (x\ \#\ xs) = \mathit{foldl}\ f\ x\ xs$

fun $\mathit{scanl1} :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$
where
$\mathit{scanl1}\ f\ xs = \mathit{map}\ (\lambda k.\ \mathit{foldl1}\ f\ (\mathit{take}\ k\ xs))$
$[1..{<}\mathit{length}\ xs + 1]$

The original paper further relies on associative functions. Thus, we define another predicate to be able to express this condition:

definition
$\mathit{associative}\ f \equiv (\forall\, x\ y\ z.\ f\ x\ (f\ y\ z) = f\ (f\ x\ y)\ z)$

The following constant symbols represents our unspecified function. We want to show that this function is semantically equivalent to *scanl1*, provided that the first argument is an associative function.

consts
$\mathit{candidate} :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$

With the final theorem, it suffices to show that *candidate* behaves like *scanl1* on all lists of the following type, to conclude that *canditate* is semantically equivalent to *scanl1*.

datatype *three* = *Zero* | *One* | *Two*

Although most of the functions mentioned in the original paper already exist in Isabelle's list theory, we still need to define two more functions:

fun *wrap* :: $'a \Rightarrow 'a$ *list*
where
  wrap $x = [x]$

fun *ups* :: *nat* $\Rightarrow$ *nat list list*
where
  ups n = *map* $(\lambda k.\ [0..<k + 1])\ [0..<n + 1]$

3 A Free Theorem

The key to proof the final theorem is the following free theorem [4, 6] of *candidate*. Since there is no proof possible without specifying the underlying (functional) language (which would be beyond the scope of this work), this lemma is expected to hold. As a consequence, all following lemmas and also the final theorem only hold under this provision.

axiomatization where
  candidate-free-theorem:
  $\bigwedge x\ y.\ h\ (f\ x\ y) = g\ (h\ x)\ (h\ y) \Longrightarrow map\ h\ (candidate\ f\ zs) = candidate\ g\ (map\ h\ zs)$

In what follows in this section, the previous lemma is specialised to a lemma for non-empty lists. More precisely, we want to restrict the above assumption to be applicable for non-empty lists. This is already possible without modifications when having a list datatype which does not allow for empty lists. However, before being able to also use Isabelle's list datatype, further conditions on f and zs are necessary.

To prove the derived lemma, we first introduce a datatype for non-empty lists, and we furthermore define conversion functions to map the new datatype on Isabelle lists and back.

datatype *'a nel*
  = *NE-One 'a*
  | *NE-Cons 'a 'a nel*

fun *n2l* :: *'a nel* $\Rightarrow$ *'a list*
where
  n2l (*NE-One x*)   = $[x]$
| *n2l* (*NE-Cons x xs*) = *x* # *n2l xs*

fun *l2n* :: *'a list* $\Rightarrow$ *'a nel*

where
  l2n [x]   = *NE-One* x
| *l2n* (x # *xs*) = (*case xs of* []   ⇒ *NE-One* x
                                      | (- # -) ⇒ *NE-Cons* x (*l2n xs*))

The following results relate Isabelle lists and non-empty lists:

lemma *non-empty-n2l*: *n2l xs* $\neq$ []
by (*cases xs*, *auto*)

lemma *n2l-l2n-id*: $x \neq [] \Longrightarrow$ *n2l* (*l2n* x) = x
proof (*induct* x)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons x xs*) **thus** *?case* **by** (*cases xs*, *auto*)
qed

lemma *n2l-l2n-map-id*:
  assumes $\bigwedge x.\ x \in$ *set zs* $\Longrightarrow x \neq []$
  shows *map* (*n2l* ∘ *l2n*) *zs* = *zs*
using *assms*
proof (*induct zs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons z zs*)
  hence $\bigwedge x.\ x \in$ *set zs* $\Longrightarrow x \neq []$ **using** *List.set-subset-Cons* **by** *auto*
  with *Cons* **have** *IH*: *map* (*n2l* ∘ *l2n*) *zs* = *zs* **by** *blast*

  have
  map (*n2l* ∘ *l2n*) (z # *zs*)
   = (*n2l* ∘ *l2n*) z # *map* (*n2l* ∘ *l2n*) *zs* **by** *simp*
  also have
  ... = z # *map* (*n2l* ∘ *l2n*) *zs* **using** *Cons* **and** *n2l-l2n-id* **by** *auto*
  also have
  ... = z # *zs* **using** *IH* **by** *simp*
  finally show *?case* **.**
qed

Based on the previous lemmas, we can state and proof a specialised version of *candidate*'s free theorem, suitable for our setting as explained before.

lemma *Lemma-1*:
  assumes *A1*: $\bigwedge(x::'a\ list)\ (y::'a\ list).$
                $x \neq [] \Longrightarrow y \neq [] \Longrightarrow h\ (f\ x\ y) = g\ (h\ x)\ (h\ y)$
    and *A2*: $\bigwedge x\ y.\ x \neq [] \Longrightarrow y \neq [] \Longrightarrow f\ x\ y \neq []$
    and *A3*: $\bigwedge x.\ x \in$ *set zs* $\Longrightarrow x \neq []$
  shows *map h* (*candidate f zs*) = *candidate g* (*map h zs*)
proof −

```
— We define two functions, fn :: 'a nel ⇒ 'a nel ⇒ 'a nel and
— hn :: 'a nel ⇒ b, which wrap f and h in the
— setting of non-empty lists.
let ?fn = λx y. l2n (f (n2l x) (n2l y))
let ?hn = h ∘ n2l

— Our new functions fulfill the preconditions of candidate's
— free theorem:
have ⋀(x::'a nel) (y::'a nel). ?hn (?fn x y) = g (?hn x) (?hn y)
  proof –
    fix x y
    let ?xl = n2l (x :: 'a nel)
    let ?yl = n2l (y :: 'a nel)
    have
    ?hn (?fn x y)
     = h (n2l (l2n (f (n2l x) (n2l y)))) by simp
    also have
    . . . = h (f ?xl ?yl)
        using A2 [where x=?xl and y=?yl]
        and n2l-l2n-id [where x=f (n2l x) (n2l y)]
        and non-empty-n2l [where xs=x]
        and non-empty-n2l [where xs=y] by simp
    also have
    . . . = g (h ?xl) (h ?yl)
        using A1 and non-empty-n2l and non-empty-n2l by auto
    also have
    . . . = g (?hn x) (?hn y) by simp
    finally show ?hn (?fn x y) = g (?hn x) (?hn y) .
  qed
with candidate-free-theorem [where f=?fn and h=?hn and g = g]
have ne-free-theorem:
map ?hn (candidate ?fn (map l2n zs)) = candidate g (map ?hn (map l2n zs))
    by auto

— We use candidate's free theorem again to show the following
— property:
have n2l-candidate:
  ⋀zs. map n2l (candidate ?fn zs) = candidate f (map n2l zs)
  proof –
    fix zs
    have ⋀x y. n2l (?fn x y) = f (n2l x) (n2l y)
      proof –
        fix x y
        show n2l (?fn x y) = f (n2l x) (n2l y)
          using n2l-l2n-id [where x=f (n2l x) (n2l y)]
            and A2 [where x=n2l x and y=n2l y]
            and non-empty-n2l [where xs=x] and non-empty-n2l [where xs=y]
          by simp
      qed
```

with *candidate-free-theorem* [**where** *h=n2l* **and** *f=?fn* **and** *g=f*]
show *map n2l (candidate ?fn zs) = candidate f (map n2l zs)* **by** *simp*
qed

— Now, with the previous preparations, we conclude the thesis by the
— following rewriting:
have
map h (candidate f zs)
= map h (candidate f (map (n2l ∘ l2n) zs))
using *n2l-l2n-map-id* [**where** *zs=zs*] **and** *A3* **by** *simp*
also have
... = map h (candidate f (map n2l (map l2n zs)))
using *List.map-map* [**where** *f=n2l* **and** *g=l2n* **and** *xs=zs*] **by** *simp*
also have
...= map h (map n2l (candidate ?fn (map l2n zs)))
using *n2l-candidate* **by** *auto*
also have
... = map ?hn (candidate ?fn (map l2n zs))
using *List.map-map* **by** *auto*
also have
... = candidate g (map ?hn (map l2n zs))
using *ne-free-theorem* **by** *simp*
also have
... = candidate g (map ((h ∘ n2l) ∘ l2n) zs)
using *List.map-map* [**where** *f=h ∘ n2l* **and** *g=l2n*] **by** *simp*
also have
... = candidate g (map (h ∘ (n2l ∘ l2n)) zs)
using *Fun.o-assoc* [*symmetric*, **where** *f=h* **and** *g=n2l* **and** *h=l2n*] **by** *simp*
also have
... = candidate g (map h (map (n2l ∘ l2n) zs))
using *List.map-map* [**where** *f=h* **and** *g=n2l ∘ l2n*] **by** *simp*
also have
... = candidate g (map h zs)
using *n2l-l2n-map-id* [**where** *zs=zs*] **and** *A3* **by** *auto*
finally show *?thesis* **.**
qed

4 Useful lemmas

In this section, we state and proof several lemmas, which neither occur in the original paper nor in Isabelle's libraries.

lemma *upt-map-Suc*:
$k > 0 \Longrightarrow [0..<k + 1] = 0 \# map\ Suc\ [0..<k]$
using *List.upt-conv-Cons* **and** *List.map-Suc-upt* **by** *simp*

lemma *divide-and-conquer-induct*:
assumes *A1*: *P* []

```
      and A2: ⋀x. P [x]
      and A3: ⋀xs ys. ⟦ xs ≠ [] ; ys ≠ [] ; P xs ; P ys ⟧ ⟹ P (xs @ ys)
  shows P zs
proof (induct zs)
  case Nil with A1 show ?case by simp
next
  case (Cons z zs)
  hence IH: P zs by simp
  show ?case
  proof (cases zs)
    case Nil with A2 show ?thesis by simp
  next
    case Cons with IH and A2 and A3 [where xs=[z] and ys=zs]
    show ?thesis by auto
  qed
qed

lemmas divide-and-conquer
  = divide-and-conquer-induct [case-names Nil One Partition]


lemma all-set-inter-empty-distinct:
  assumes ⋀xs ys. js = xs @ ys ⟹ set xs ∩ set ys = {}
  shows distinct js
using assms proof (induct js rule: divide-and-conquer)
  case Nil thus ?case by simp
next
  case One thus ?case by simp
next
  case (Partition xs ys)
  hence P: ⋀as bs. xs @ ys = as @ bs ⟹ set as ∩ set bs = {} by simp

  have ⋀xs1 xs2. xs = xs1 @ xs2 ⟹ set xs1 ∩ set xs2 = {}
    proof −
      fix xs1 xs2
      assume xs = xs1 @ xs2
      hence set xs1 ∩ set (xs2 @ ys) = {}
        using P [where as=xs1 and bs=xs2 @ ys] by simp
      thus set xs1 ∩ set xs2 = {}
        using List.set-append and Set.Int-Un-distrib by auto
    qed
  with Partition have distinct-xs: distinct xs by simp
  have ⋀ys1 ys2. ys = ys1 @ ys2 ⟹ set ys1 ∩ set ys2 = {}
    proof −
      fix ys1 ys2
      assume ys = ys1 @ ys2
      hence set (xs @ ys1) ∩ set ys2 = {}
        using P [where as=xs @ ys1 and bs=ys2] by simp
      thus set ys1 ∩ set ys2 = {}
```

```
      using List.set-append and Set.Int-Un-distrib by auto
    qed
  with Partition have distinct-ys: distinct ys by simp
  from Partition and distinct-xs and distinct-ys show ?case by simp
qed



lemma partitions-sorted:
  assumes ⋀xs ys x y. ⟦ js = xs @ ys ; x ∈ set xs ; y ∈ set ys ⟧ ⟹ x ≤ y
  shows sorted js
using assms proof (induct js rule: divide-and-conquer)
  case Nil thus ?case by simp
next
  case One thus ?case by simp
next
  case (Partition xs ys)
  hence P: ⋀as bs x y. ⟦ xs @ ys = as @ bs ; x ∈ set as ; y ∈ set bs⟧ ⟹ x ≤ y
    by simp

  have ⋀xs1 xs2 x y. ⟦ xs = xs1 @ xs2 ; x ∈ set xs1 ; y ∈ set xs2 ⟧ ⟹ x ≤ y
    proof −
      fix xs1 xs2
      assume xs = xs1 @ xs2
      hence ⋀x y. ⟦ x ∈ set xs1 ; y ∈ set (xs2 @ ys) ⟧ ⟹ x ≤ y
        using P [where as=xs1 and bs=xs2 @ ys] by simp
      thus ⋀x y. ⟦ x ∈ set xs1 ; y ∈ set xs2 ⟧ ⟹ x ≤ y
        using List.set-append by auto
    qed
  with Partition have sorted-xs: sorted xs by simp
  have ⋀ys1 ys2 x y. ⟦ ys = ys1 @ ys2 ; x ∈ set ys1 ; y ∈ set ys2 ⟧ ⟹ x ≤ y
    proof −
      fix ys1 ys2
      assume ys = ys1 @ ys2
      hence ⋀x y. ⟦ x ∈ set (xs @ ys1) ; y ∈ set ys2 ⟧ ⟹ x ≤ y
        using P [where as=xs @ ys1 and bs=ys2] by simp
      thus ⋀x y. ⟦ x ∈ set ys1 ; y ∈ set ys2 ⟧ ⟹ x ≤ y
        using List.set-append by auto
    qed
  with Partition have sorted-ys: sorted ys by simp

  have ∀x ∈ set xs. ∀y ∈ set ys. x ≤ y
    using P [where as=xs and bs=ys] by simp
  with sorted-xs and sorted-ys show ?case using List.sorted-append by auto
qed
```

5 Preparatory Material

In the original paper, the following lemmas L1 to L8 are given without a proof, although it is hinted there that most of them follow from parametricity properties [4, 6]. Alternatively, most of them can be shown by induction over lists. However, since we are using Isabelle's list datatype, we rely on already existing results.

lemma *L1*: *map g* (*map f xs*) = *map* (*g* ∘ *f*) *xs*
using *List.map-map* **by** *auto*

lemma *L2*: *length* (*map f xs*) = *length xs*
using *List.length-map* **by** *simp*

lemma *L3*: *take k* (*map f xs*) = *map f* (*take k xs*)
using *List.take-map* **by** *auto*

lemma *L4*: *map f* ∘ *wrap* = *wrap* ∘ *f*
by (*simp add*: *fun-eq-iff*)

lemma *L5*: *map f* (*xs* @ *ys*) = (*map f xs*) @ (*map f ys*)
using *List.map-append* **by** *simp*

lemma *L6*: *k* < *length xs* ⟹ (*map f xs*) ! *k* = *f* (*xs* ! *k*)
using *List.nth-map* **by** *simp*

lemma *L7*: ⋀*k*. *k* < *length xs* ⟹ *map* (*nth xs*) [*0*..<*k* + *1*] = *take* (*k* + *1*) *xs*
proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons x xs*)
  thus *?case*
  proof (*cases k*)
    case *0* **thus** *?thesis* **by** *simp*
  next
    case (*Suc k′*)
    hence *k* > *0* **by** *simp*
    hence *map* (*nth* (*x* # *xs*)) [*0*..<*k* + *1*]
          = *map* (*nth* (*x* # *xs*)) (*0* # *map Suc* [*0*..<*k*])
      using *upt-map-Suc* **by** *simp*
    also have ... = ((*x* # *xs*) ! *0*) # (*map* (*nth* (*x* # *xs*) ∘ *Suc*) [*0*..<*k*])
      using *L1* **by** *simp*
    also have ... = *x* # *map* (*nth xs*) [*0*..<*k*] **by** *simp*
    also have ... = *x* # *map* (*nth xs*) [*0*..<*k′* + *1*] **using** *Suc* **by** *simp*
    also have ... = *x* # *take* (*k′* + *1*) *xs* **using** *Cons* **and** *Suc* **by** *simp*
    also have ... = *take* (*k* + *1*) (*x* # *xs*) **using** *Suc* **by** *simp*
    finally show *?thesis* **.**
  qed

qed

In Isabelle's list theory, a similar result for *foldl* already exists. Therefore, it is easy to prove the following lemma for *foldl1*. Note that this lemma does not occur in the original paper.

lemma *foldl1-append*:
  assumes $xs \neq []$
  shows $foldl1\ f\ (xs\ @\ ys) = foldl1\ f\ (foldl1\ f\ xs\ \#\ ys)$
proof –
  have *non-empty-list*: $xs \neq [] \Longrightarrow \exists\, y\ ys.\ xs = y\ \#\ ys$ **by** (*cases xs*, *auto*)
  with *assms* **obtain** $x\ xs'$ **where** *x-xs-def*: $xs = x\ \#\ xs'$ **by** *auto*

  have $foldl1\ f\ (xs\ @\ ys) = foldl\ f\ x\ (xs'\ @\ ys)$ **using** *x-xs-def* **by** *simp*
  also have $\ldots = foldl\ f\ (foldl\ f\ x\ xs')\ ys$ **using** *List.foldl-append* **by** *simp*
  also have $\ldots = foldl\ f\ (foldl1\ f\ (x\ \#\ xs'))\ ys$ **by** *simp*
  also have $\ldots = foldl1\ f\ (foldl1\ f\ xs\ \#\ ys)$ **using** *x-xs-def* **by** *simp*
  finally show *?thesis* **.**
qed

This is a special induction scheme suitable for proving L8. It is not mentioned in the original paper.

lemma *foldl1-induct'*:
  assumes $\bigwedge f\ x.\ P\ f\ [x]$
    and $\bigwedge f\ x\ y.\ P\ f\ [x,\ y]$
    and $\bigwedge f\ x\ y\ z\ zs.\ P\ f\ (f\ x\ y\ \#\ z\ \#\ zs) \Longrightarrow P\ f\ (x\ \#\ y\ \#\ z\ \#\ zs)$
    and $\bigwedge f.\ P\ f\ []$
  shows $P\ f\ xs$
proof (*induct xs rule*: *List.length-induct*)
  fix *xs*
  assume A: $\forall\, ys{::}'a\ list.\ length\ ys < length\ (xs{::}'a\ list) \longrightarrow P\ f\ ys$
  thus $P\ f\ xs$
  proof (*cases xs*)
    case *Nil* **with** *assms* **show** *?thesis* **by** *simp*
  next
    case (*Cons x1 xs1*)
    hence *xs1*: $xs = x1\ \#\ xs1$ **by** *simp*
    thus *?thesis*
    proof (*cases xs1*)
      case *Nil* **with** *assms* **and** *xs1* **show** *?thesis* **by** *simp*
    next
      case (*Cons x2 xs2*)
      hence *xs2*: $xs1 = x2\ \#\ xs2$ **by** *simp*
      thus *?thesis*
      proof (*cases xs2*)
        case *Nil* **with** *assms* **and** *xs1* **and** *xs2* **show** *?thesis* **by** *simp*
      next
        case (*Cons x3 xs3*)
        hence $xs2 = x3\ \#\ xs3$ **by** *simp*
        with *assms* **and** *xs1 xs2* **and** *A* **show** *?thesis* **by** *simp*

qed
qed
qed
qed

lemmas *foldl1-induct* = *foldl1-induct′* [*case-names One Two More Nil*]

lemma *L8*:
assumes *associative f*
and *xs* ≠ []
and *ys* ≠ []
shows *foldl1 f* (*xs* @ *ys*) = *f* (*foldl1 f xs*) (*foldl1 f ys*)
using *assms* **proof** (*induct f ys rule*: *foldl1-induct*)
case (*One f y*)
have
foldl1 f (*xs* @ [*y*])
= *foldl1 f* (*foldl1 f xs* # [*y*])
using *foldl1-append* [**where** *xs*=*xs*] **and** *One* **by** *simp*
also have
. . . = *f* (*foldl1 f xs*) *y* **by** *simp*
also have
. . . = *f* (*foldl1 f xs*) (*foldl1 f* [*y*]) **by** *simp*
finally show *?case* **.**
next
case (*Two f x y*)
have
foldl1 f (*xs* @ [*x*, *y*])
= *foldl1 f* (*foldl1 f xs* # [*x*, *y*])
using *foldl1-append* [**where** *xs*=*xs*] **and** *Two* **by** *simp*
also have
. . . = *foldl1 f* (*f* (*foldl1 f xs*) *x* # [*y*]) **by** *simp*
also have
. . . = *f* (*f* (*foldl1 f xs*) *x*) *y* **by** *simp*
also have
. . . = *f* (*foldl1 f xs*) (*f x y*) **using** *Two*
unfolding *associative-def* **by** *simp*
also have
. . . = *f* (*foldl1 f xs*) (*foldl1 f* [*x*, *y*]) **by** *simp*
finally show *?case* **.**
next
case (*More f x y z zs*)
hence *IH*: *foldl1 f* (*xs* @ *f x y* # *z* # *zs*)
= *f* (*foldl1 f xs*) (*foldl1 f* (*f x y* # *z* # *zs*)) **by** *simp*

have
foldl1 f (*xs* @ *x* # *y* # *z* # *zs*)
= *foldl1 f* (*foldl1 f xs* # *x* # *y* # *z* # *zs*)
using *foldl1-append* [**where** *xs*=*xs*] **and** *More* **by** *simp*

also have
$\ldots = \mathit{foldl1}\ f\ (f\ (\mathit{foldl1}\ f\ \mathit{xs})\ x\ \#\ y\ \#\ z\ \#\ \mathit{zs})$ **by** *simp*
also have
$\ldots = \mathit{foldl1}\ f\ (f\ (f\ (\mathit{foldl1}\ f\ \mathit{xs})\ x)\ y\ \#\ z\ \#\ \mathit{zs})$ **by** *simp*
also have
$\ldots = \mathit{foldl1}\ f\ (f\ (\mathit{foldl1}\ f\ \mathit{xs})\ (f\ x\ y)\ \#\ z\ \#\ \mathit{zs})$
  using *More* **unfolding** *associative-def* **by** *simp*
also have
$\ldots = \mathit{foldl1}\ f\ (\mathit{foldl1}\ f\ \mathit{xs}\ \#\ f\ x\ y\ \#\ z\ \#\ \mathit{zs})$ **by** *simp*
also have
$\ldots = \mathit{foldl1}\ f\ (\mathit{xs}\ @\ f\ x\ y\ \#\ z\ \#\ \mathit{zs})$
  using *foldl1-append* [**where** *xs=xs*] **and** *More* **by** *simp*
also have
$\ldots = f\ (\mathit{foldl1}\ f\ \mathit{xs})\ (\mathit{foldl1}\ f\ (x\ \#\ y\ \#\ z\ \#\ \mathit{zs}))$
  using *IH* **by** *simp*
finally show *?case* **.**
next
case *Nil* **thus** *?case* **by** *simp*
qed

The next lemma is applied in several following proofs whenever the equivalence of two lists is shown.

lemma *Lemma-2*:
assumes $\mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys}$
  and $\bigwedge k.\ k < \mathit{length}\ \mathit{xs} \Longrightarrow \mathit{xs}\ !\ k = \mathit{ys}\ !\ k$
shows $\mathit{xs} = \mathit{ys}$
using *assms* **by** (*auto simp*: *List.list-eq-iff-nth-eq*)

In the original paper, this lemma and its proof appear inside of Lemma 3. However, this property will be useful also in later proofs and is thus separated.

lemma *foldl1-map*:
assumes *associative f*
  and $\mathit{xs} \neq []$
  and $\mathit{ys} \neq []$
shows $\mathit{foldl1}\ f\ (\mathit{map}\ h\ (\mathit{xs}\ @\ \mathit{ys}))$
  $= f\ (\mathit{foldl1}\ f\ (\mathit{map}\ h\ \mathit{xs}))\ (\mathit{foldl1}\ f\ (\mathit{map}\ h\ \mathit{ys}))$
proof −
have
$\mathit{foldl1}\ f\ (\mathit{map}\ h\ (\mathit{xs}\ @\ \mathit{ys}))$
$= \mathit{foldl1}\ f\ (\mathit{map}\ h\ \mathit{xs}\ @\ \mathit{map}\ h\ \mathit{ys})$
  using *L5* **by** *simp*
also have
$\ldots = f\ (\mathit{foldl1}\ f\ (\mathit{map}\ h\ \mathit{xs}))\ (\mathit{foldl1}\ f\ (\mathit{map}\ h\ \mathit{ys}))$
  using *assms* **and** *L8* [**where** *f=f*] **by** *auto*
finally show *?thesis* **.**
qed

lemma *Lemma-3*:
  fixes f :: $'a \Rightarrow 'a \Rightarrow 'a$
    and h :: $nat \Rightarrow 'a$
  assumes *associative f*
  shows $map\ (foldl1\ f \circ map\ h)\ (candidate\ (@)\ (map\ wrap\ [0..<n+1]))$
    $= candidate\ f\ (map\ h\ [0..<n+1])$
proof –
  — The following three properties *P1*, *P2* and *P3*
  — are preconditions of Lemma 1.
  have *P1*:
  $\bigwedge x\ y.\ [\![\ x \neq []\ ;\ y \neq []\ ]\!]$
  $\Longrightarrow foldl1\ f\ (map\ h\ (x\ @\ y)) = f\ (foldl1\ f\ (map\ h\ x))\ (foldl1\ f\ (map\ h\ y))$
    using *assms* **and** *foldl1-map* **by** *blast*

  have *P2*: $\bigwedge x\ y.\ x \neq [] \Longrightarrow y \neq [] \Longrightarrow x\ @\ y\ \neq []$ **by** *auto*

  have *P3*: $\bigwedge x.\ x \in set\ (map\ wrap\ [0..<n+1]) \Longrightarrow x \neq []$ **by** *auto*

  — The proof for the thesis is now equal to the one of the original paper:
  from *Lemma-1* [**where** $h{=}foldl1\ f \circ map\ h$ **and** $zs{=}map\ wrap\ [0..<n+1]$
    and $f{=}(@)$] **and** *P1 P2 P3*
  have
  $map\ (foldl1\ f \circ map\ h)\ (candidate\ (@)\ (map\ wrap\ [0..<n+1]))$
  $= candidate\ f\ (map\ (foldl1\ f \circ map\ h)\ (map\ wrap\ [0..<n+1]))$
    by *auto*
  also have
  $\ldots = candidate\ f\ (map\ (foldl1\ f \circ map\ h \circ wrap)\ [0..<n+1])$
    by (*simp add*: *L1*)
  also have
  $\ldots = candidate\ f\ (map\ (foldl1\ f \circ wrap \circ h)\ [0..<n+1])$
    using *L4* **by** (*simp add*: *Fun.o-def*)
  also have
  $\ldots = candidate\ f\ (map\ h\ [0..<n+1])$
    by (*simp add*: *Fun.o-def*)
  finally show *?thesis* **.**
qed

6 Proving Proposition 1

6.1 Definitions of Lemma 4

In the same way as in the original paper, the following two functions are defined:

fun *f1* :: $three \Rightarrow three \Rightarrow three$
where
  f1 x Zero = *x*
| *f1 Zero One* = *One*
| *f1 x y* = *Two*

```
fun f2 :: three ⇒ three ⇒ three
where
  f2 x Zero = x
| f2 x One  = One
| f2 x Two  = Two
```

Both functions are associative as is proved by case analysis:

```
lemma f1-assoc: associative f1
unfolding associative-def proof auto
  fix x y z
  show f1 x (f1 y z) = f1 (f1 x y) z
  proof (cases z)
    case Zero thus ?thesis by simp
  next
    case One
    hence z-One: z = One by simp
    thus ?thesis by (cases y, simp-all, cases x, simp-all)
  next
    case Two thus ?thesis by simp
  qed
qed

lemma f2-assoc: associative f2
unfolding associative-def proof auto
  fix x y z
  show f2 x (f2 y z) = f2 (f2 x y) z by (cases z, auto)
qed
```

Next, we define two other functions, again according to the original paper. Note that *h1* has an extra parameter *k* which is only implicit in the original paper.

```
fun h1 :: nat ⇒ nat ⇒ nat ⇒ three
where
  h1 k i j = (if i = j then One
              else if j ≤ k then Zero
              else Two)

fun h2 :: nat ⇒ nat ⇒ three
where
  h2 i j = (if i = j then One
            else if i + 1 = j then Two
            else Zero)
```

6.2 Figures and Proofs

In the original paper, this lemma is depicted in (and proved by) Figure 2. Therefore, it carries this unusual name here.

```
lemma Figure-2:
```

```
  assumes i ≤ k
  shows foldl1 f1 (map (h1 k i) [0..<k + 1]) = One
proof −
  let ?mr = replicate i Zero @ [One] @ replicate (k − i) Zero

  have P1: map (h1 k i) [0..<k + 1] = ?mr
    proof −
      have Q1: length (map (h1 k i) [0..<k + 1]) = length ?mr
        using assms by simp

      have Q2: ⋀j. j < length (map (h1 k i) [0..<k + 1])
                  ⟹ (map (h1 k i) [0..<k + 1]) ! j = ?mr ! j
        proof −
          fix j
          assume j < length (map (h1 k i) [0..<k + 1])
          hence j-k: j < k + 1 by simp
          have M1: (map (h1 k i) [0..<k + 1]) ! i = One
            using L6 [where f=h1 k i and xs=[0..<k + 1]] and assms
              and List.nth-upt [where i=0 and k=i and j=k + 1] by simp
          have M2: j ≠ i ⟹ (map (h1 k i) [0..<k + 1]) ! j = Zero
            using L6 [where f=h1 k i and xs=[0..<k + 1]] and j-k
              and List.nth-upt [where i=0 and j=k + 1] by simp
          have R1: ?mr ! i = One
            using List.nth-append [where xs=replicate i Zero] by simp
          have R2: j < i ⟹ ?mr ! j = Zero
            using List.nth-append [where xs=replicate i Zero] by simp
          have R3: j > i ⟹ ?mr ! j = Zero
            using List.nth-append [where xs=replicate i Zero @ [One]]
              and j-k by simp

          show (map (h1 k i) [0..<k + 1]) ! j = ?mr ! j
          proof (cases i = j)
            assume i = j
            with M1 and R1 show ?thesis by simp
          next
            assume i-ne-j: i ≠ j
            thus ?thesis
            proof (cases i < j)
              assume i < j
              with M2 and R3 show ?thesis by simp
            next
              assume ¬(i < j)
              with i-ne-j have i > j by simp
              with M2 and R2 show ?thesis by simp
            qed
          qed
        qed

      from Q1 Q2 and Lemma-2 show ?thesis by blast
```

```
    qed

  have P2: ⋀j. j > 0 ⟹ foldl1 f1 (replicate j Zero) = Zero
    proof −
      fix j
      assume (j::nat) > 0
      thus foldl1 f1 (replicate j Zero) = Zero
      proof (induct j)
        case 0 thus ?case by simp
      next
        case (Suc j) thus ?case by (cases j, auto)
      qed
    qed

  have P3: ⋀j. foldl1 f1 ([One] @ replicate j Zero) = One
    proof −
      fix j
      show foldl1 f1 ([One] @ replicate j Zero) = One
        using L8 [where f=f1 and xs=[One] and ys=replicate (Suc j) Zero]
          and f1-assoc and P2 [where j=Suc j] by simp
    qed

  have foldl1 f1 ?mr = One
    proof (cases i)
      case 0
      thus ?thesis using P3 by simp
    next
      case (Suc i)
      hence
      foldl1 f1 (replicate (Suc i) Zero @ [One] @ replicate (k − Suc i) Zero)
       = f1 (foldl1 f1 (replicate (Suc i) Zero))
            (foldl1 f1 ([One] @ replicate (k − Suc i) Zero))
          using L8 [where xs=replicate (Suc i) Zero
                    and ys=[One] @ replicate (k − Suc i) Zero]
          and f1-assoc by simp
      also have
      . . . = One
          using P2 [where j=Suc i] and P3 [where j=k − Suc i] by simp
      finally show ?thesis using Suc by simp
    qed
  with P1 show ?thesis by simp
qed
```

In the original paper, this lemma is depicted in (and proved by) Figure 3. Therefore, it carries this unusual name here.

```
lemma Figure-3:
  assumes i < k
  shows foldl1 f2 (map (h2 i) [0..<k + 1]) = Two
proof −
```

```
let ?mr = replicate i Zero @ [One, Two] @ replicate (k − i − 1) Zero

have P1: map (h2 i) [0..<k + 1] = ?mr
  proof −
    have Q1: length (map (h2 i) [0..<k + 1]) = length ?mr
      using assms by simp

    have Q2: ⋀j. j < length (map (h2 i) [0..<k + 1])
              ⟹ (map (h2 i) [0..<k + 1]) ! j = ?mr ! j
      proof −
        fix j
        assume j < length (map (h2 i) [0..<k + 1])
        hence j-k: j < k + 1 by simp
        have M1: (map (h2 i) [0..<k + 1]) ! i = One
          using L6 [where xs=[0..<k + 1] and f=h2 i and k=i] and assms
          and List.nth-upt [where i=0 and k=i and j=k + 1] by simp
        have M2: (map (h2 i) [0..<k + 1]) ! (i + 1) = Two
          using L6 [where xs=[0..<k + 1] and f=h2 i and k=i + 1]
          and assms and List.nth-upt [where i=0 and k=i + 1 and j=k + 1]
          by simp
        have M3: j < i ∨ j > i + 1 ⟹ (map (h2 i) [0..<k + 1]) ! j = Zero
          using L6 [where xs=[0..<k + 1] and f=h2 i and k=j]
          and assms and List.nth-upt [where i=0 and k=j and j=k + 1]
          and j-k by auto
        have R1: j < i ⟹ ?mr ! j = Zero
          using List.nth-append [where xs=replicate i Zero] by simp
        have R2: ?mr ! i = One
          using List.nth-append [where xs=replicate i Zero] by simp
        have R3: ?mr ! (i + 1) = Two
          using List.nth-append [where xs=replicate i Zero @ [One]] by simp
        have R4: j > i + 1 ⟹ ?mr ! j = Zero
          using List.nth-append [where xs=replicate i Zero @ [One,Two]]
          and j-k by simp
        show (map (h2 i) [0..<k + 1]) ! j = ?mr ! j
        proof (cases j < i)
          assume j < i with M3 and R1 show ?thesis by simp
        next
          assume ¬(j < i)
          hence j-ge-i: j ≥ i by simp
          thus ?thesis
          proof (cases j = i)
            assume j = i with M1 and R2 show ?thesis by simp
          next
            assume ¬(j = i)
            with j-ge-i have j-gt-i: j > i by simp
            thus ?thesis
            proof (cases j = i + 1)
              assume j = i + 1 with M2 and R3 show ?thesis by simp
            next
```

```
          assume ¬(j = i + 1)
          with j-gt-i have j > i + 1 by simp
          with M3 and R4 show ?thesis by simp
         qed
        qed
       qed
      qed
    from Q1 Q2 and Lemma-2 show ?thesis by blast
  qed

have P2: ⋀j. j > 0 ⟹ foldl1 f2 (replicate j Zero) = Zero
  proof −
    fix j
    assume j-0: (j::nat) > 0
    show foldl1 f2 (replicate j Zero) = Zero
    using j-0 proof (induct j)
      case 0 thus ?case by simp
    next
      case (Suc j) thus ?case by (cases j, auto)
    qed
  qed

have P3: ⋀j. foldl1 f2 ([One, Two] @ replicate j Zero) = Two
  proof −
    fix j
    show foldl1 f2 ([One, Two] @ replicate j Zero) = Two
      using L8 [where f=f2 and xs=[One,Two]
      and ys=replicate (Suc j) Zero] and f2-assoc and P2 [where j=Suc j]
      by simp
  qed

have foldl1 f2 ?mr = Two
  proof (cases i)
    case 0 thus ?thesis using P3 by simp
  next
    case (Suc i)
    hence
    foldl1 f2 (replicate (Suc i) Zero @ [One, Two]
                                   @ replicate (k − Suc i − 1) Zero)
     = f2 (foldl1 f2 (replicate (Suc i) Zero))
          (foldl1 f2 ([One, Two] @  replicate (k − Suc i − 1) Zero))
       using L8 [where f=f2 and xs=replicate (Suc i) Zero
               and ys=[One, Two] @ replicate (k − Suc i − 1) Zero]
       and f2-assoc by simp
    also have
    . . . = Two
       using P2 [where j=Suc i] and P3 [where j=k − Suc i − 1] by simp
    finally show ?thesis using Suc by simp
  qed
```

with *P1* **show** *?thesis* **by** *simp*
qed

Counterparts of the following two lemmas are shown in the proof of Lemma 4 in the original paper. Since here, the proof of Lemma 4 is seperated in several smaller lemmas, also these two properties are given separately.

lemma *L9*:
  assumes $\bigwedge$ (f :: *three* $\Rightarrow$ *three* $\Rightarrow$ *three*) h. *associative* f
    $\Longrightarrow$ *foldl1* f (*map* h *js*) = *foldl1* f (*map* h [*0*..<k + *1*])
    and $i \leq k$
  shows *foldl1 f1* (*map* (*h1* k i) *js*) = *One*
using *assms* **and** *f1-assoc* **and** *Figure-2* **by** *auto*

lemma *L10*:
  assumes $\bigwedge$ (f :: *three* $\Rightarrow$ *three* $\Rightarrow$ *three*) h. *associative* f
    $\Longrightarrow$ *foldl1* f (*map* h *js*) = *foldl1* f (*map* h [*0*..<k + *1*])
    and $i < k$
  shows *foldl1 f2* (*map* (*h2* i) *js*) = *Two*
using *assms* **and** *f2-assoc* **and** *Figure-3* **by** *auto*

In the original paper, this lemma is depicted in (and proved by) Figure 4. Therefore, it carries this unusual name here. This lemma expresses that every $i \leq k$ is contained in *js* at least once.

lemma *Figure-4*:
  assumes *foldl1 f1* (*map* (*h1* k i) *js*) = *One*
    and *js* $\neq$ []
  shows $i \in$ *set js*
proof (*rule ccontr*)
  assume *i-not-in-js*: $i \notin$ *set js*

  have *One-not-in-map-js*: *One* $\notin$ *set* (*map* (*h1* k i) *js*)
    proof
      assume *One* $\in$ *set* (*map* (*h1* k i) *js*)
      hence *One* $\in$ (*h1* k i) ‘ (*set js*) **by** *simp*
      then obtain j **where** *j-def*: $j \in$ *set js* $\wedge$ *One* = *h1* k i j
        using *Set.image-iff* [**where** f=*h1* k i] **by** *auto*
      hence $i = j$ **by** (*simp split*: *if-splits*)
      with *i-not-in-js* **and** *j-def* **show** *False* **by** *simp*
    qed

  have *f1-One*: $\bigwedge x\ y.\ x \neq$ *One* $\wedge\ y \neq$ *One* $\Longrightarrow$ *f1* x y $\neq$ *One*
    proof −
      fix x y
      assume $x \neq$ *One* $\wedge\ y \neq$ *One*
      thus *f1* x y $\neq$ *One* **by** (*cases* y, *cases* x, *auto*)
    qed

  have $\bigwedge$*xs*. ⟦ *xs* $\neq$ [] ; *One* $\notin$ *set xs* ⟧ $\Longrightarrow$ *foldl1 f1 xs* $\neq$ *One*

```
  proof -
    fix xs
    assume A: (xs :: three list) ≠ []
    thus One ∉ set xs ⟹ foldl1 f1 xs ≠ One
    proof (induct xs rule: divide-and-conquer)
      case Nil thus ?case by simp
    next
      case (One x)
      thus foldl1 f1 [x] ≠ One by simp
    next
      case (Partition xs ys)
      hence One ∉ set xs ∧ One ∉ set ys by simp
      with Partition have foldl1 f1 xs ≠ One ∧ foldl1 f1 ys ≠ One by simp
      with f1-One have f1 (foldl1 f1 xs) (foldl1 f1 ys) ≠ One by simp
      with L8 [symmetric, where f=f1] and f1-assoc and Partition
      show foldl1 f1 (xs @ ys) ≠ One by auto
    qed
  qed
 with One-not-in-map-js and assms show False by auto
qed
```

In the original paper, this lemma is depicted in (and proved by) Figure 5. Therefore, it carries this unusual name here. This lemma expresses that every $i \leq k$ is contained in *js* at most once.

```
lemma Figure-5:
 assumes foldl1 f1 (map (h1 k i) js) = One
   and js = xs @ ys
 shows ¬(i ∈ set xs ∧ i ∈ set ys)
proof (rule ccontr)
 assume ¬¬(i ∈ set xs ∧ i ∈ set ys)
 hence i-xs-ys: i ∈ set xs ∧ i ∈ set ys by simp

 from i-xs-ys have xs-not-empty: xs ≠ [] by auto
 from i-xs-ys have ys-not-empty: ys ≠ [] by auto

 have f1-Zero: ⋀x y. x ≠ Zero ∨ y ≠ Zero ⟹ f1 x y ≠ Zero
  proof -
    fix x y
    show x ≠ Zero ∨ y ≠ Zero ⟹ f1 x y ≠ Zero
    by (cases y, simp-all, cases x, simp-all)
  qed

 have One-foldl1: ⋀xs. One ∈ set xs ⟹ foldl1 f1 xs ≠ Zero
  proof -
    fix xs
    assume One-xs: One ∈ set xs
    thus foldl1 f1 xs ≠ Zero
    proof (induct xs rule: divide-and-conquer)
      case Nil thus ?case by simp
```

```
    next
      case One thus ?case by simp
    next
      case (Partition xs ys)
      hence One ∈ set xs ∨ One ∈ set ys by simp
      with Partition have foldl1 f1 xs ≠ Zero ∨ foldl1 f1 ys ≠ Zero by auto
      with f1-Zero have f1 (foldl1 f1 xs) (foldl1 f1 ys) ≠ Zero by simp
     thus ?case using L8 [symmetric, where f=f1] and f1-assoc and Partition
        by auto
    qed
  qed

  have f1-Two: ⋀x y. x ≠ Zero ∧ y ≠ Zero ⟹ f1 x y = Two
    proof −
      fix x y
      show x ≠ Zero ∧ y ≠ Zero ⟹ f1 x y = Two
      by (cases y, simp-all, cases x, simp-all)
    qed

  from i-xs-ys
  have One ∈ set (map (h1 k i) xs) ∧ One ∈ set (map (h1 k i) ys) by simp
  hence foldl1 f1 (map (h1 k i) xs) ≠ Zero
        ∧ foldl1 f1 (map (h1 k i) ys) ≠ Zero
    using One-foldl1 by simp
  hence f1 (foldl1 f1 (map (h1 k i) xs)) (foldl1 f1 (map (h1 k i) ys)) = Two
    using f1-Two by simp
  hence foldl1 f1 (map (h1 k i) (xs @ ys)) = Two
    using foldl1-map [symmetric, where h=h1 k i] and f1-assoc
      and xs-not-empty and ys-not-empty by auto
  with assms show False by simp
qed
```

In the original paper, this lemma is depicted in (and proved by) Figure 6. Therefore, it carries this unusual name here. This lemma expresses that *js* contains only elements of $[0..{<}k + 1]$.

```
lemma Figure-6:
  assumes ⋀i. i ≤ k ⟹ foldl1 f1 (map (h1 k i) js) = One
      and i > k
  shows i ∉ set js
proof
  assume i-in-js: i ∈ set js

  have Two-map: Two ∈ set (map (h1 k 0) js)
    proof −
      have Two = h1 k 0 i using assms by simp
      with i-in-js show ?thesis using IntI by (auto split: if-splits)
    qed

  have f1-Two: ⋀x y. x = Two ∨ y = Two ⟹ f1 x y = Two
```

```
    proof -
      fix x y
      show x = Two ∨ y = Two ⟹ f1 x y = Two by (cases y, auto)
    qed

  have ⋀xs. Two ∈ set xs ⟹ foldl1 f1 xs = Two
    proof -
      fix xs
      assume Two-xs: Two ∈ set xs
      thus foldl1 f1 xs = Two using Two-xs
      proof (induct xs rule: divide-and-conquer)
        case Nil thus ?case by simp
      next
        case One thus ?case by simp
      next
        case (Partition xs ys)
        hence Two ∈ set xs ∨ Two ∈ set ys by simp
        hence foldl1 f1 xs = Two ∨ foldl1 f1 ys = Two using Partition by auto
        with f1-Two have f1 (foldl1 f1 xs) (foldl1 f1 ys) = Two by simp
        thus foldl1 f1 (xs @ ys) = Two
          using L8 [symmetric, where f=f1] and f1-assoc and Partition by auto
      qed
    qed

  with Two-map have foldl1 f1 (map (h1 k 0) js) = Two by simp
  with assms show False by auto
qed
```

In the original paper, this lemma is depicted in (and proved by) Figure 7. Therefore, it carries this unusual name here. This lemma expresses that every $i \leq k$ in *js* is eventually followed by $i + 1$.

```
lemma Figure-7:
  assumes foldl1 f2 (map (h2 i) js) = Two
      and js = xs @ ys
      and xs ≠ []
      and i = last xs
  shows (i + 1) ∈ set ys
proof (rule ccontr)
  assume Suc-i-not-in-ys: (i + 1) ∉ set ys

  have last-map-One: last (map (h2 i) xs) = One
    proof -
      have
      last (map (h2 i) xs)
        = (map (h2 i) xs) ! (length (map (h2 i) xs) - 1)
          using List.last-conv-nth [where xs=map (h2 i) xs] and assms by simp
      also have
      ... = (map (h2 i) xs) ! (length xs - 1) using L2 by simp
      also have
```

. . . = *h2 i* (*xs* ! (*length xs* − *1*))
    using *L6* **and** *assms* **by** *simp*
  also have
  . . . = *h2 i* (*last xs*)
    using *List.last-conv-nth* [*symmetric*,**where** *xs*=*xs*] **and** *assms* **by** *simp*
  also have
  . . . = *One* **using** *assms* **by** *simp*
  finally show *?thesis* **.**
qed

have ⋀*xs*. ⟦ *xs* ≠ [] ; *last xs* = *One* ⟧ ⟹ *foldl1 f2 xs* = *One*
proof −
  fix *xs*
  assume *last-xs-One*: *last xs* = *One*
  assume *xs-not-empty*: *xs* ≠ []
  hence *xs-partition*: *xs* = *butlast xs* @ [*last xs*] **by** *simp*
  show *foldl1 f2 xs* = *One*
  proof (*cases butlast xs*)
    case *Nil* **with** *xs-partition* **and** *last-xs-One* **show** *?thesis* **by** *simp*
  next
    case *Cons*
    hence *butlast-not-empty*: *butlast xs* ≠ [] **by** *simp*

    have
    foldl1 f2 xs = *foldl1 f2* (*butlast xs* @ [*last xs*])
      using *xs-partition* **by** *simp*
    also have
    . . . = *f2* (*foldl1 f2* (*butlast xs*)) (*foldl1 f2* [*last xs*])
      using *L8* [**where** *f*=*f2*] **and** *f2-assoc* **and** *butlast-not-empty* **by** *simp*
    also have
    . . . = *One* **using** *last-xs-One* **by** *simp*
    finally show *?thesis* **.**
  qed
qed
with *last-map-One* **have** *foldl1-map-xs*: *foldl1 f2* (*map* (*h2 i*) *xs*) = *One*
  using *assms* **by** *simp*

have *ys-not-empty*: *ys* ≠ [] **using** *foldl1-map-xs* **and** *assms* **by** *auto*

have *Two-map-ys*: *Two* ∉ *set* (*map* (*h2 i*) *ys*)
proof
  assume *Two* ∈ *set* (*map* (*h2 i*) *ys*)
  hence *Two* ∈ (*h2 i*) ‘ (*set ys*) **by** *simp*
  then obtain *j* **where** *j-def*: *j* ∈ *set ys* ∧ *Two* = *h2 i j*
    using *Set.image-iff* [**where** *f*=*h2 i*] **by** *auto*
  hence *i* + *1* = *j* **by** (*simp split*: *if-splits*)
  with *Suc-i-not-in-ys* **and** *j-def* **show** *False* **by** *simp*
qed

```
  have f2-One: ⋀x y. x ≠ Two ∧ y ≠ Two ⟹ f2 x y ≠ Two
    proof -
      fix x y
      show x ≠ Two ∧ y ≠ Two ⟹ f2 x y ≠ Two by (cases y, auto)
    qed

  have ⋀xs. ⟦ xs ≠ [] ; Two ∉ set xs ⟧ ⟹ foldl1 f2 xs ≠ Two
    proof -
      fix xs
      assume xs-not-empty: (xs :: three list) ≠ []
      thus Two ∉ set xs ⟹ foldl1 f2 xs ≠ Two
      proof (induct xs rule: divide-and-conquer)
        case Nil thus ?case by simp
      next
        case One thus ?case by simp
      next
        case (Partition xs ys)
        hence Two ∉ set xs ∧ Two ∉ set ys by simp
        hence foldl1 f2 xs ≠ Two ∧ foldl1 f2 ys ≠ Two using Partition by simp
        hence f2 (foldl1 f2 xs) (foldl1 f2 ys) ≠ Two using f2-One by simp
        thus foldl1 f2 (xs @ ys) ≠ Two
          using L8 [symmetric, where f=f2] and f2-assoc and Partition by simp
      qed
    qed
  with Two-map-ys have foldl1-map-ys: foldl1 f2 (map (h2 i) ys) ≠ Two
    using ys-not-empty by simp

  from f2-One
  have f2 (foldl1 f2 (map (h2 i) xs)) (foldl1 f2 (map (h2 i) ys)) ≠ Two
    using foldl1-map-xs and foldl1-map-ys by simp
  hence foldl1 f2 (map (h2 i) (xs @ ys)) ≠ Two
    using foldl1-map [symmetric, where h=h2 i and f=f2] and f2-assoc
      and assms and ys-not-empty by simp
  with assms show False by simp
qed
```

6.3 Permutations and Lemma 4

In the original paper, the argumentation goes as follows: From *Figure-4* and *Figure-5* we can show that *js* contains every $i \leq k$ exactly once, and from *Figure-6* we can furthermore show that *js* contains no other elements. Thus, *js* must be a permutation of $[0..<k + 1]$.

Here, however, the argumentation is different, because we want to use already existing results. Therefore, we show first, that the sets of *js* and $[0..<k + 1]$ are equal using the results of *Figure-4* and *Figure-6*. Second, we show that *js* is a distinct list, i.e. no element occurs twice in *js*. Since also $[0..<k + 1]$ is distinct, the multisets of *js* and $[0..<k + 1]$ are equal, and therefore, both lists are permutations.

lemma *js-is-a-permutation*:
  assumes *A1*: $\bigwedge$ (f :: *three* $\Rightarrow$ *three* $\Rightarrow$ *three*) h. *associative f*
    $\Longrightarrow$ *foldl1 f* (*map h js*) = *foldl1 f* (*map h* [*0*..<*k* + *1*])
    and *A2*: *js* $\neq$ []
  shows *mset js* = *mset* [*0*..<*k* + *1*]
proof −
  from *A1* **and** *L9* **have** *L9′*:
  $\bigwedge i.\ i \leq k \Longrightarrow$ *foldl1 f1* (*map* (*h1 k i*) *js*) = *One* **by** *auto*

  from *L9′* **and** *Figure-4* **and** *A2* **have** *P1*: $\bigwedge i.\ i \leq k \Longrightarrow i \in$ *set js* **by** *auto*
  from *L9′* **and** *Figure-5* **have** *P2*:
    $\bigwedge$*i xs ys*. ⟦ $i \leq k$; *js* = *xs* @ *ys* ⟧ $\Longrightarrow \neg$($i \in$ *set xs* $\wedge$ $i \in$ *set ys*) **by** *blast*
  from *L9′* **and** *Figure-6* **have** *P3*: $\bigwedge i.\ i > k \Longrightarrow i \notin$ *set js* **by** *auto*

  have *set-eq*: *set* [*0*..<*k* + *1*] = *set js*
    proof
      from *P1* **show** *set* [*0*..<*k* + *1*] $\subseteq$ *set js* **by** *auto*
    next
      show *set js* $\subseteq$ *set* [*0*..<*k* + *1*]
      proof
        fix *j*
        assume $j \in$ *set js*
        hence $\neg$($j \notin$ *set js*) **by** *simp*
        with *P3* **have** $\neg(j > k)$ **using** *HOL.contrapos-nn* **by** *auto*
        hence $j \leq k$ **by** *simp*
        thus $j \in$ *set* [*0*..<*k* + *1*] **by** *auto*
      qed
    qed

  have $\bigwedge$*xs ys*. *js* = *xs* @ *ys* $\Longrightarrow$ *set xs* $\cap$ *set ys* = {}
    proof −
      fix *xs ys*
      assume *js-xs-ys*: *js* = *xs* @ *ys*
      with *set-eq* **have** *i-xs-ys*: $\bigwedge i.\ i \in$ *set xs* $\vee$ $i \in$ *set ys* $\Longrightarrow i \leq k$ **by** *auto*
      have $\bigwedge i.\ i \leq k \Longrightarrow$ ($i \in$ *set xs*) = ($i \notin$ *set ys*)
        proof
          fix *i*
          assume $i \in$ *set xs*
          moreover assume $i \leq k$
          ultimately show $i \notin$ *set ys* **using** *js-xs-ys* **and** *P2* **by** *simp*
        next
          fix *i*
          assume $i \notin$ *set ys*
          moreover assume $i \leq k$
          ultimately show $i \in$ *set xs* **using** *js-xs-ys* **and** *P2* **and** *P1* **by** *auto*
        qed
      thus *set xs* $\cap$ *set ys* = {} **using** *i-xs-ys* **by** *auto*
    qed
  with *all-set-inter-empty-distinct* **have** *distinct js* **using** *A2* **by** *auto*

with *set-eq* **show** *mset js* = *mset* [*0..<k* + *1*]
  using *Multiset.set-eq-iff-mset-eq-distinct*
    [**where** *x=js* **and** *y=*[*0..<k* + *1*]] **by** *simp*
qed

The result of *Figure-7* is too specific. Instead of having that every i is eventually followed by $i + 1$, it more useful to know that every i is followed by all $i + r$, where $r > 0$. This result follows easily by induction from *Figure-7*.

```
lemma Figure-7-trans:
  assumes A1: ⋀i xs ys. ⟦ i < k ; js = xs @ ys ; xs ≠ [] ; i = last xs ⟧
                ⟹ (i + 1) ∈ set ys
    and A2: (r::nat) > 0
    and A3: i + r ≤ k
    and A4: js = xs @ ys
    and A5: xs ≠ []
    and A6: i = last xs
  shows (i + r) ∈ set ys
using A2 A3 proof (induct r)
  case 0 thus ?case by simp
next
  case (Suc r)
  hence IH: 0 < r ⟹ (i + r) ∈ set ys by simp
  from Suc have i-r-k: i + Suc r ≤ k by simp
  show ?case
  proof (cases r)
    case 0 thus ?thesis using A1 and i-r-k and A4 and A5 and A6 by auto
  next
    case Suc
    with IH have (i + r) ∈ set ys by simp
    then obtain p where p-def: p < length ys ∧ ys ! p = i + r
      using List.in-set-conv-nth [where x=i + r] by auto

    let ?xs = xs @ take (p + 1) ys
    let ?ys = drop (p + 1) ys

    have i + r < k using i-r-k by simp
    moreover have js = ?xs @ ?ys using A4 by simp
    moreover have ?xs ≠ [] using A5 by simp
    moreover have i + r = last ?xs
     using p-def and List.take-Suc-conv-app-nth [where i=p and xs=ys] by simp
    ultimately have (i + Suc r) ∈ set ?ys using A1 [where i=i + r] by auto
    thus (i + Suc r) ∈ set ys
      using List.set-drop-subset [where xs=ys] by auto
  qed
qed
```

Since we want to use Lemma *partitions-sorted* to show that *js* is sorted, we need yet another result which can be obtained using the previous lemma and some further argumentation:

lemma *js-partition-order*:
  assumes *A1*: *mset js* = *mset* [*0*..<*k* + *1*]
    and *A2*: $\bigwedge$*i xs ys*. ⟦ *i* < *k* ; *js* = *xs* @ *ys* ; *xs* ≠ [] ; *i* = *last xs* ⟧
        ⟹ (*i* + *1*) ∈ *set ys*
    and *A3*: *js* = *xs* @ *ys*
    and *A4*: *i* ∈ *set xs*
    and *A5*: *j* ∈ *set ys*
  shows *i* ≤ *j*
proof (*rule ccontr*)
  from *A1* **have** *A1'*: ‹*set js* = {..<*k* + *1*}›
    by (*metis atLeast-upt mset-eq-setD*)
  assume ¬(*i* ≤ *j*)
  hence *i-j*: *i* > *j* **by** *simp*

  from *A5* **obtain** *pj* **where** *pj-def*: *pj* < *length ys* ∧ *ys* ! *pj* = *j*
    using *List.in-set-conv-nth* [**where** *x*=*j*] **by** *auto*

  let *?xs* = *xs* @ *take* (*pj* + *1*) *ys*
  let *?ys* = *drop* (*pj* + *1*) *ys*

  let *?r* = *i* − *j*

  from *A1* **and** *A3* **have** *distinct* (*xs* @ *ys*)
    using *distinct-upt mset-eq-imp-distinct-iff* **by** *blast*

  hence *xs-ys-inter-empty*: *set xs* ∩ *set ys* = {} **by** *simp*

  from *A2* **and** *Figure-7-trans* **have**
  $\bigwedge$*i r xs ys*. ⟦ *r* > *0* ; *i* + *r* ≤ *k* ; *js* = *xs* @ *ys* ; *xs* ≠ [] ; *i* = *last xs* ⟧
      ⟹ (*i* + *r*) ∈ *set ys* **by** *blast*
  moreover from *i-j* **have** *?r* > *0* **by** *simp*
  moreover have *j* + *?r* ≤ *k*
    proof −
      have *i* ∈ *set js* **using** *A4* **and** *A3* **by** *simp*
      hence *i* ∈ *set* [*0*..<*k* + *1*]
        using *A1'* **by** (*auto simp add*: *less-Suc-eq*)
      hence *i* ≤ *k* **by** *auto*
      thus *?thesis* **using** *i-j* **by** *simp*
    qed
  moreover have *js* = *?xs* @ *?ys* **using** *A3* **by** *simp*
  moreover have *?xs* ≠ [] **using** *A4* **by** *auto*
  moreover have *j* = *last* (*?xs*)
   using *pj-def* **and** *List.take-Suc-conv-app-nth* [**where** *i*=*pj* **and** *xs*=*ys*] **by** *simp*
  ultimately have (*j* + *?r*) ∈ *set ?ys* **by** *blast*
  hence *i* ∈ *set ys* **using** *i-j* **and** *List.set-drop-subset* [**where** *xs*=*ys*] **by** *auto*
  with *A4* **and** *xs-ys-inter-empty* **show** *False* **by** *auto*
qed

With the help of the previous lemma, we show now that *js* equals [*0*..<*k* + *1*], if both lists are permutations and every *i* is eventually followed by *i* + *1* in *js*.

lemma *js-equals-upt-k*:
  assumes *A1*: *mset js* = *mset* [*0*..<*k* + *1*]
    and *A2*: $\bigwedge$*i xs ys*. ⟦ *i* < *k* ; *js* = *xs* @ *ys* ; *xs* ≠ [] ; *i* = *last xs* ⟧
      ⟹ (*i* + *1*) ∈ *set ys*
  shows *js* = [*0*..<*k* + *1*]
proof −
  from *A1* **and** *A2* **and** *js-partition-order*
  have $\bigwedge$*xs ys x y*. ⟦ *js* = *xs* @ *ys* ; *x* ∈ *set xs* ; *y* ∈ *set ys* ⟧ ⟹ *x* ≤ *y*
    by *blast*
  hence *sorted js* **using** *partitions-sorted* **by** *blast*
  moreover have *distinct js*
    using *A1 distinct-upt mset-eq-imp-distinct-iff* **by** *blast*
  moreover have *sorted* [*0*..<*k* + *1*]
    using *List.sorted-upt* **by** *blast*
  moreover have *distinct* [*0*..<*k* + *1*] **by** *simp*
  moreover have *set js* = *set* [*0*..<*k* + *1*]
    using *A1 mset-eq-setD* **by** *blast*
  ultimately show *js* = [*0*..<*k* + *1*] **using** *List.sorted-distinct-set-unique*
    by *blast*
qed

From all the work done before, we conclude now Lemma 4:

lemma *Lemma-4*:
  assumes $\bigwedge$(*f* :: *three* ⇒ *three* ⇒ *three*) *h*. *associative f*
      ⟹ *foldl1 f* (*map h js*) = *foldl1 f* (*map h* [*0*..<*k* + *1*])
    and *js* ≠ []
  shows *js* = [*0*..<*k* + *1*]
proof −
  from *assms* **and** *js-is-a-permutation* **have** *mset js* = *mset*[*0*..<*k* + *1*] **by** *auto*
  moreover
  from *assms* **and** *L10* **and** *Figure-7*
  have $\bigwedge$*i xs ys*. ⟦ *i* < *k* ; *js* = *xs* @ *ys* ; *xs* ≠ [] ; *i* = *last xs* ⟧
      ⟹ (*i* + *1*) ∈ *set ys* **by** *blast*
  ultimately show *?thesis* **using** *js-equals-upt-k* **by** *auto*
qed

6.4 Lemma 5

This lemma is a lifting of Lemma 4 to the overall computation of *scanl1*. Its proof follows closely the one given in the original paper.

lemma *Lemma-5*:
  assumes $\bigwedge$(*f* :: *three* ⇒ *three* ⇒ *three*) *h*. *associative f*
      ⟹ *map* (*foldl1 f* ∘ *map h*) *jss* = *scanl1 f* (*map h* [*0*..<*n* + *1*])
    and $\bigwedge$*js*. *js* ∈ *set jss* ⟹ *js* ≠ []
  shows *jss* = *ups n*

proof −
  have *P1*: *length jss* = *length* (*ups n*)
    proof −
      obtain *f* :: *three* ⇒ *three* ⇒ *three* **where** *f-assoc*: *associative f*
        using *f1-assoc* **by** *auto*

      fix *h*
      have
      length jss = *length* (*map* (*foldl1 f* ∘ *map h*) *jss*) **by** (*simp add*: *L2*)
      also have
      ... = *length* (*scanl1 f* (*map h* [*0*..<*n* + *1*]))
          using *assms* **and** *f-assoc* **by** *simp*
      also have
      ... = *length* (*map* (λ*k*. *foldl1 f* (*take* (*k* + *1*) (*map h* [*0*..<*n* + *1*])))
                    [*0*..<*length* (*map h* [*0*..<*n* + *1*])]) **by** *simp*
      also have
      ... = *length* [*0*..<*length* (*map h* [*0*..<*n* + *1*])] **by** (*simp add*: *L2*)
      also have
      ... = *length* [*0*..<*length* [*0*..<*n* + *1*]] **by** (*simp add*: *L2*)
      also have
      ... = *length* [*0*..<*n* + *1*] **by** *simp*
      also have
      ... = *length* (*map* (λ*k*. [*0*..<*k* + *1*]) [*0*..<*n* + *1*]) **by** (*simp add*: *L2*)
      also have
      ... = *length* (*ups n*) **by** *simp*
      finally show *?thesis* **.**
    qed

  have *P2*: ⋀*k*. *k* < *length jss* ⟹ *jss* ! *k* = (*ups n*) ! *k*
    proof −
      fix *k*
      assume *k-length-jss*: *k* < *length jss*
      hence *non-empty-jss-k*: *jss* ! *k* ≠ [] **using** *assms* **by** *simp*

      from *k-length-jss*
      have *k-length-length*: *k* < *length* [*1*..<*length* [*0*..<*n* + *1*] + *1*]
        using *P1* **by** *simp*
      hence *k-length*: *k* < *length* [*0*..<*n* + *1*]
        using *List.length-upt* [**where** *i*=*1* **and** *j*=*length* [*0*..<*n* + *1*] + *1*]
        by *simp*

      have ⋀(*f* :: *three* ⇒ *three* ⇒ *three*) *h*. *associative f*
              ⟹ *foldl1 f* (*map h* (*jss* ! *k*)) = *foldl1 f* (*map h* [*0*..<*k* + *1*])
        proof −
          fix *f h*
          assume *f-assoc*: *associative* (*f* :: *three* ⇒ *three* ⇒ *three*)
          have
          foldl1 f (*map h* (*jss* ! *k*))
           = (*map* (*foldl1 f* ∘ *map h*) *jss*) ! *k*

```
        using L6 and k-length-jss by auto
      also have
      ... = (scanl1 f (map h [0..<n + 1])) ! k
        using assms and f-assoc by simp
      also have
      ... = (map (λk. foldl1 f (take k (map h [0..<n + 1])))
               [1..<length (map h [0..<n + 1]) + 1]) ! k by simp
      also have
      ... = (map (λk. foldl1 f (take k (map h [0..<n + 1])))
               [1..<length [0..<n + 1] + 1]) ! k
        by (simp add: L2)
      also have
      ... = (λk. foldl1 f (take k (map h [0..<n + 1])))
             ([1..<length [0..<n + 1] + 1] ! k)
       using L6 [where xs=[1..<length [0..<n + 1] + 1]
        and f=(λk. foldl1 f (take k (map h [0..<n + 1])))]
        and k-length-length by auto
      also have
      ... = foldl1 f (take (k + 1) (map h [0..<n + 1]))
        proof -
          have [1..<length [0..<n + 1] + 1] ! k  = k + 1
            using List.nth-upt [where i=1 and j=length [0..<n + 1] + 1]
              and k-length by simp
          thus ?thesis by simp
        qed
      also have
      ... = foldl1 f (map h (take (k + 1) [0..<n + 1]))
        using L3 [where k=k + 1 and xs=[0..<n + 1] and f=h] by simp
      also have ... = foldl1 f (map h [0..<k + 1])
        using List.take-upt [where i=0 and m=k + 1 and n=n + 1]
          and k-length by simp
      finally show
      foldl1 f (map h (jss ! k)) = foldl1 f (map h [0..<k + 1]) .
    qed
  with Lemma-4 and non-empty-jss-k have P3: jss ! k = [0..<k + 1]
    by blast

  have
  (ups n) ! k
   = (map (λk. [0..<k + 1]) [0..<n + 1]) ! k by simp
  also have
  ... = (λk. [0..<k + 1]) ([0..<n + 1] ! k)
    using L6 [where xs=[0..<n + 1]] and k-length by auto
  also have
  ... = [0..<k + 1]
    using List.nth-upt [where i=0 and j=n + 1] and k-length by simp
  finally have (ups n) ! k = [0..<k + 1] .

  with P3 show jss ! k = (ups n) ! k by simp
```

qed

from *P1 P2* **and** *Lemma-2* **show** *jss* = *ups n* **by** *blast*
qed

6.5 Proposition 1

In the original paper, only non-empty lists where considered, whereas here, the used list datatype allows also for empty lists. Therefore, we need to exclude non-empty lists to have a similar setting as in the original paper.

In the case of Proposition 1, we need to show that every list contained in the result of *candidate* (@) (*map wrap* [*0..*<*n* + *1*]) is non-empty. The idea is to interpret empty lists by the value *Zero* and non-empty lists by the value *One*, and to apply the assumptions.

lemma *non-empty-candidate-results*:
  assumes $\bigwedge$ (*f* :: *three* ⇒ *three* ⇒ *three*) (*xs* :: *three list*).
    ⟦ *associative f* ; *xs* ≠ [] ⟧ ⟹ *candidate f xs* = *scanl1 f xs*
   and *js* ∈ *set* (*candidate* (@) (*map wrap* [*0..*<*n* + *1*]))
  shows *js* ≠ []
proof −
  — We define a mapping of lists to values of *three* as explained
  — above, and a function which behaves like @ on values of
  — *three*.
  let *?h* = λ*xs. case xs of* [] ⇒ *Zero* | (-#-) ⇒ *One*
  let *?g* = λ*x y. if* (*x* = *One* ∨ *y* = *One*) *then One else Zero*
  have *g-assoc*: *associative ?g* **unfolding** *associative-def* **by** *auto*

  — Our defined functions fulfill the requirements of the free theorem of
  — *candidate*, that is:
  have *req-free-theorem*: $\bigwedge$*xs ys. ?h* (*xs* @ *ys*) = *?g* (*?h xs*) (*?h ys*)
   proof −
    fix *xs ys*
    show *?h* (*xs* @ *ys*) = *?g* (*?h xs*) (*?h ys*)
     by (*cases xs*, *simp-all*, *cases ys*, *simp-all*)
   qed

  — Before applying the assumptions, we show that *candidate*'s
  — counterpart *scanl1*, applied to a non-empty list, returns only
  — a repetition of the value *One*.
  have *set-scanl1-is-One*:
   set (*scanl1 ?g* (*map ?h* (*map wrap* [*0..*<*n* + *1*]))) = {*One*}
   proof −
    have *const-One*: *map* (λ*x. One*) [*0..*<*n* + *1*] = *replicate* (*n* + *1*) *One*
     proof (*induct n*)
      case *0* **thus** *?case* **by** *simp*
     next
      case (*Suc n*)
      have

map (*λx. One*) [*0..<Suc n + 1*]
  = *map* (*λx. One*) ([*0..<Suc n*] @ [*Suc n*]) **by** *simp*
also have
... = *map* (*λx. One*) [*0..<Suc n*] @ *map* (*λx. One*) [*Suc n*]
    by *simp*
also have ... = *replicate* (*Suc n*) *One* @ *replicate 1 One*
    using *Suc* **by** *simp*
also have ... = *replicate* (*Suc n + 1*) *One*
    using *List.replicate-add*
        [*symmetric*, **where** *x=One* **and** *n=Suc n* **and** *m=1*]
    by *simp*
finally show *?case* **.**
qed

have *foldl1-One*: $\bigwedge$*xs. foldl1 ?g* (*One # xs*) = *One*
  proof −
    fix *xs*
    show *foldl1 ?g* (*One # xs*) = *One*
    proof (*induct xs rule*: *measure-induct* [**where** *f=length*])
      fix *x*
      assume ∀ *y. length y < length* (*x::three list*)
          ⟶ *foldl1 ?g* (*One # y*) = *One*
      thus *foldl1 ?g* (*One # x*) = *One* **by** (*cases x*, *auto*)
    qed
  qed

have
scanl1 ?g (*map ?h* (*map wrap* [*0..<n + 1*]))
 = *scanl1 ?g* (*map* (*?h ∘ wrap*) [*0..<n + 1*])
    using *L1* [**where** *g=?h* **and** *f=wrap* **and** *xs=*[*0..<n + 1*]] **by** *simp*
also have
... = *scanl1 ?g* (*map* (*λx. One*) [*0..<n + 1*]) **by** (*simp add*: *Fun.o-def*)
also have
... = *scanl1 ?g* (*replicate* (*n + 1*) *One*) **using** *const-One* **by** *auto*
also have
... = *map* (*λk. foldl1 ?g* (*take k* (*replicate* (*n + 1*) *One*)))
        [*1..<length* (*replicate* (*n + 1*) *One*) + *1*] **by** *simp*
also have
... = *map* (*λk. foldl1 ?g* (*take k* (*replicate* (*n + 1*) *One*)))
        (*map Suc* [*0..<length* (*replicate* (*n + 1*) *One*)])
    using *List.map-Suc-upt* **by** *simp*
also have
... = *map* ((*λk. foldl1 ?g* (*take k* (*replicate* (*n + 1*) *One*))) ∘ *Suc*)
        [*0..<length* (*replicate* (*n + 1*) *One*)]
    using *L1* **by** *simp*
also have
... = *map* (*λk. foldl1 ?g* (*replicate* (*min* (*k + 1*) (*n + 1*)) *One*))
        [*0..<n + 1*] **using** *Fun.o-def* **by** *simp*
also have

$\ldots = map\ (\lambda k.\ foldl1\ ?g\ (One\ \#\ replicate\ (min\ k\ n)\ One))$
$[0..<n + 1]$ **by** *simp*
also have
$\ldots = map\ (\lambda k.\ One)\ [0..<n + 1]$ **using** *foldl1-One* **by** *simp*
also have
$\ldots = replicate\ (n + 1)\ One$ **using** *const-One* **by** *simp*
finally show *?thesis* **using** *List.set-replicate* [**where** $n=n + 1$] **by** *simp*
qed

— Thus, with the assumptions and the free theorem of candidate, we show
— that results of *candidate*, after applying *h*, can only
— have the value *One*.
have
$scanl1\ ?g\ (map\ ?h\ (map\ wrap\ [0..<n + 1]))$
$= candidate\ ?g\ (map\ ?h\ (map\ wrap\ [0..<n + 1]))$
using *assms* **and** *g-assoc* **by** *auto*
also have
$\ldots = map\ ?h\ (candidate\ (@)\ (map\ wrap\ [0..<n + 1]))$
using *candidate-free-theorem* [*symmetric*, **where** $f=(@)$ **and** $g=?g$
and $h=?h$ **and** $zs=(map\ wrap\ [0..<n + 1])$] **and** *req-free-theorem* **by** *auto*
finally have *set-is-One*:
$\bigwedge x.\ x \in set\ (map\ ?h\ (candidate\ (@)\ (map\ wrap\ [0..<n + 1])))$
$\Longrightarrow x = One$
using *set-scanl1-is-One* **by** *auto*

— Now, it is easy to conclude the thesis.
from *assms*
have $?h\ js \in ?h\ `\ set\ (candidate\ (@)\ (map\ wrap\ [0..<n + 1]))$ **by** *auto*
with *set-is-One* **have** $?h\ js = One$ **by** *simp*
thus $js \neq []$ **by** *auto*
qed

Proposition 1 is very similar to the corresponding one shown in the original paper except of a slight modification due to the choice of using Isabelle's list datatype.

Strictly speaking, the requirement that *xs* must be non-empty in the assumptions of Proposition 1 is not necessary, because only non-empty lists are applied in the proof. However, the additional requirement eases the proof obligations of the final theorem, i.e. this additions allows more (or easier) applications of the final theorem.

lemma *Proposition-1*:
assumes $\bigwedge\ (f :: three \Rightarrow three \Rightarrow three)\ (xs :: three\ list).$
$[\![\ associative\ f\ ;\ xs \neq []\ ]\!] \Longrightarrow candidate\ f\ xs = scanl1\ f\ xs$
shows $candidate\ (@)\ (map\ wrap\ [0..<n + 1]) = ups\ n$
proof −
— This addition is necessary because we are using Isabelle's list datatype
— which allows for empty lists.
from *assms* **and** *non-empty-candidate-results* **have** *non-empty-candidate*:
$\bigwedge js.\ js \in set\ (candidate\ (@)\ (map\ wrap\ [0..<n + 1])) \Longrightarrow js \neq []$

by *auto*

have $\bigwedge(f{::}\ three \Rightarrow three \Rightarrow three)\ h.\ associative\ f$
$\Longrightarrow map\ (foldl1\ f \circ map\ h)\ (candidate\ (@)\ (map\ wrap\ [0..{<}n + 1]))$
$= scanl1\ f\ (map\ h\ [0..{<}n + 1])$
proof −
fix *f h*
assume *f-assoc*: $associative\ (f :: three \Rightarrow three \Rightarrow three)$
hence
$map\ (foldl1\ f \circ map\ h)\ (candidate\ (@)\ (map\ wrap\ [0..{<}n + 1]))$
$= candidate\ f\ (map\ h\ [0..{<}n + 1])$ **using** *Lemma-3* **by** *auto*
also have
$\ldots = scanl1\ f\ (map\ h\ [0..{<}n + 1])$
using *assms* [**where** $xs{=}map\ h\ [0..{<}n + 1]$] **and** *f-assoc* **by** *simp*
finally show
$map\ (foldl1\ f \circ map\ h)\ (candidate\ (@)\ (map\ wrap\ [0..{<}n + 1]))$
$= scanl1\ f\ (map\ h\ [0..{<}n + 1])$ **.**
qed
with *Lemma-5* **and** *non-empty-candidate* **show** *?thesis* **by** *auto*
qed

7 Proving Proposition 2

Before proving Proposition 2, a non-trivial step of that proof is shown first. In the original paper, the argumentation simply applies L7 and the definition of *map* and $[0..{<}k + 1]$. However, since, L7 requires that k must be less than $length\ [0..{<}length\ xs]$ and this does not simply follow for the bound occurrence of k, a more complicated proof is necessary. Here, it is shown based on Lemma 2.

lemma *Prop-2-step-L7*:
$map\ (\lambda k.\ foldl1\ g\ (map\ (nth\ xs)\ [0..{<}k + 1]))\ [0..{<}length\ xs]$
$= map\ (\lambda k.\ foldl1\ g\ (take\ (k + 1)\ xs))\ [0..{<}length\ xs]$
proof −
have *P1*:
$length\ (map\ (\lambda k.\ foldl1\ g\ (map\ (nth\ xs)\ [0..{<}k + 1]))\ [0..{<}length\ xs])$
$= length\ (map\ (\lambda k.\ foldl1\ g\ (take\ (k + 1)\ xs))\ [0..{<}length\ xs])$
by (*simp add*: *L2*)

have *P2*:
$\bigwedge k.\ k < length\ (map\ (\lambda k.\ foldl1\ g\ (map\ (nth\ xs)\ [0..{<}k + 1]))$
$[0..{<}length\ xs])$
$\Longrightarrow (map\ (\lambda k.\ foldl1\ g\ (map\ (nth\ xs)\ [0..{<}k + 1]))\ [0..{<}length\ xs])\ !\ k$
$= (map\ (\lambda k.\ foldl1\ g\ (take\ (k + 1)\ xs))\ [0..{<}length\ xs])\ !\ k$
proof −
fix *k*
assume *k-length*:
$k < length\ (map\ (\lambda k.\ foldl1\ g\ (map\ (nth\ xs)\ [0..{<}k + 1]))$
$[0..{<}length\ xs])$

hence *k-length′*: $k < length\ xs$ **by** (*simp add*: *L2*)

have
$(map\ (\lambda k.\ foldl1\ g\ (map\ (nth\ xs)\ [0..<k + 1]))\ [0..<length\ xs])\ !\ k$
$= (\lambda k.\ foldl1\ g\ (map\ (nth\ xs)\ [0..<k + 1]))\ ([0..<length\ xs]\ !\ k)$
using *L6* **and** *k-length* **by** (*simp add*: *L2*)
also have
$\ldots = foldl1\ g\ (map\ (nth\ xs)\ [0..<k + 1])$
using *k-length′* **by** (*auto simp*: *L2*)
also have
$\ldots = foldl1\ g\ (take\ (k + 1)\ xs)$
using *L7* [**where** *k=k* **and** *xs=xs*] **and** *k-length′* **by** *simp*
also have
$\ldots = (\lambda k.\ foldl1\ g\ (take\ (k + 1)\ xs))\ ([0..<length\ xs]\ !\ k)$
using *k-length′* **by** (*auto simp*: *L2*)
also have
$\ldots = (map\ (\lambda k.\ foldl1\ g\ (take\ (k + 1)\ xs))\ [0..<length\ xs])\ !\ k$
using *L6* [*symmetric*] **and** *k-length* **by** (*simp add*: *L2*)
finally show
$(map\ (\lambda k.\ foldl1\ g\ (map\ (nth\ xs)\ [0..<k + 1]))\ [0..<length\ xs])\ !\ k$
$= (map\ (\lambda k.\ foldl1\ g\ (take\ (k + 1)\ xs))\ [0..<length\ xs])\ !\ k$ **.**
qed

from *P1 P2* **and** *Lemma-2* **show** *?thesis* **by** *blast*
qed

Compared to the original paper, here, Proposition 2 has the additional assumption that *xs* is non-empty. The proof, however, is identical to the the one given in the original paper, except for the non-trivial step shown before.

lemma *Proposition-2*:
assumes *A1*: $\bigwedge n.\ candidate\ (@)\ (map\ wrap\ [0..<n + 1]) = ups\ n$
and *A2*: *associative g*
and *A3*: $xs \neq []$
shows $candidate\ g\ xs = scanl1\ g\ xs$
proof −
— First, based on Lemma 2, we show that
— $xs = map\ ((!)\ xs)\ [0..<length\ xs]$
— by the following facts P1 and P2.

have *P1*: $length\ xs = length\ (map\ (nth\ xs)\ [0..<length\ xs])$
proof −
have
$length\ xs$
$= length\ [0..<length\ xs]$ **by** *simp*
also have
$\ldots = length\ (map\ (nth\ xs)\ [0..<length\ xs])$
using *L2* [*symmetric*] **by** *auto*
finally show *?thesis* **.**
qed

have *P2*: $\bigwedge k.\ k < \mathit{length}\ \mathit{xs} \Longrightarrow \mathit{xs}\ !\ k = (\mathit{map}\ (\mathit{nth}\ \mathit{xs})\ [0..<\mathit{length}\ \mathit{xs}])\ !\ k$
  proof −
    fix *k*
    assume *k-length-xs*: $k < \mathit{length}\ \mathit{xs}$
    hence *k-length-xs'*: $k < \mathit{length}\ [0..<\mathit{length}\ \mathit{xs}]$ **by** *simp*
    have
    $\mathit{xs}\ !\ k = \mathit{nth}\ \mathit{xs}\ ([0..<\mathit{length}\ \mathit{xs}]\ !\ k)$
        using *k-length-xs* **by** *simp*
    also have
    $\ldots = (\mathit{map}\ (\mathit{nth}\ \mathit{xs})\ [0..<\mathit{length}\ \mathit{xs}])\ !\ k$
        using *L6* [*symmetric*] **and** *k-length-xs'* **by** *auto*
    finally show $\mathit{xs}\ !\ k = (\mathit{map}\ (\mathit{nth}\ \mathit{xs})\ [0..<\mathit{length}\ \mathit{xs}])\ !\ k$ **.**
  qed

from *P1 P2* **and** *Lemma-2* **have** $\mathit{xs} = \mathit{map}\ (\mathit{nth}\ \mathit{xs})\ [0..<\mathit{length}\ \mathit{xs}]$ **by** *blast*

— Thus, with some rewriting, we show the thesis.
hence
candidate g xs
$= \mathit{candidate}\ g\ (\mathit{map}\ (\mathit{nth}\ \mathit{xs})\ [0..<\mathit{length}\ \mathit{xs}])$ **by** *simp*
also have
$\ldots = \mathit{map}\ (\mathit{foldl1}\ g \circ \mathit{map}\ (\mathit{nth}\ \mathit{xs}))$
        $(\mathit{candidate}\ (@)\ (\mathit{map}\ \mathit{wrap}\ [0..<\mathit{length}\ \mathit{xs}]))$
    using *Lemma-3* [*symmetric*, **where** $h{=}\mathit{nth}\ \mathit{xs}$ **and** $n{=}\mathit{length}\ \mathit{xs} - 1$]
    and *A2* **and** *A3* **by** *auto*
also have
$\ldots = \mathit{map}\ (\mathit{foldl1}\ g \circ \mathit{map}\ (\mathit{nth}\ \mathit{xs}))\ (\mathit{ups}\ (\mathit{length}\ \mathit{xs} - 1))$
    using *A1* [**where** $n{=}\mathit{length}\ \mathit{xs} - 1$] **and** *A3* **by** *simp*
also have
$\ldots = \mathit{map}\ (\mathit{foldl1}\ g \circ \mathit{map}\ (\mathit{nth}\ \mathit{xs}))\ (\mathit{map}\ (\lambda k.\ [0..<k + 1])\ [0..<\mathit{length}\ \mathit{xs}])$
    using *A3* **by** *simp*
also have
$\ldots = \mathit{map}\ (\lambda k.\ \mathit{foldl1}\ g\ (\mathit{map}\ (\mathit{nth}\ \mathit{xs})\ [0..<k + 1]))\ [0..<\mathit{length}\ \mathit{xs}]$
    using *L1* [**where** $g{=}\mathit{foldl1}\ g \circ \mathit{map}\ (\mathit{nth}\ \mathit{xs})$ **and** $f{=}(\lambda k.\ [0..<k + 1])$]
    by (*simp add*: *Fun.o-def*)
also have
$\ldots = \mathit{map}\ (\lambda k.\ \mathit{foldl1}\ g\ (\mathit{take}\ (k + 1)\ \mathit{xs}))\ [0..<\mathit{length}\ \mathit{xs}]$
    using *Prop-2-step-L7* **by** *simp*
also have
$\ldots = \mathit{map}\ (\lambda k.\ \mathit{foldl1}\ g\ (\mathit{take}\ k\ \mathit{xs}))\ (\mathit{map}\ (\lambda k.\ k + 1)\ [0..<\mathit{length}\ \mathit{xs}])$
    by (*simp add*: *L1*)
also have
$\ldots = \mathit{map}\ (\lambda k.\ \mathit{foldl1}\ g\ (\mathit{take}\ k\ \mathit{xs}))\ [1..<\mathit{length}\ \mathit{xs} + 1]$
    using *List.map-Suc-upt* **by** *simp*
also have
$\ldots = \mathit{scanl1}\ g\ \mathit{xs}$ **by** *simp*
finally show *?thesis* **.**
qed

8 The Final Result

Finally, we the main result follows directly from Proposition 1 and Proposition 2.

theorem *The-0-1-2-Principle*:
  assumes $\bigwedge$ (f :: $three \Rightarrow three \Rightarrow three$) ($xs$ :: $three\ list$).
    $[\![$ *associative f* ; $xs \neq []$ $]\!] \Longrightarrow$ *candidate f xs* = *scanl1 f xs*
   and *associative g*
   and $ys \neq []$
  shows *candidate g ys* = *scanl1 g ys*
using *Proposition-1 Proposition-2* **and** *assms* **by** *blast*

Acknowledgments

I thank Janis Voigtländer for sharing a draft of his paper before its publication with me.

References

[1] Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*. Margona Kaufmann, 1993.

[2] N. A. Day, J. Launchbury, and J. Lewis. Logical Abstractions in Haskell. In *Proceedings of the 1999 Haskell Workshop*. Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28, October 1999.

[3] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[4] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[5] J. Voigtländer. Much Ado about Two – A Pearl on Parallel Prefix Computation. In *POPL'08*. ACM, Jan. 2008.

[6] P. Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.