# Much Ado about Two 

By Sascha Böhme

September 13, 2023


#### 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 ..... 1
2 Basic definitions ..... 4
3 A Free Theorem ..... 5
4 Useful lemmas ..... 6
5 Preparatory Material ..... 7
6 Proving Proposition 1 ..... 9
6.1 Definitions of Lemma 4 ..... 9
6.2 Figures and Proofs ..... 9
6.3 Permutations and Lemma 4 ..... 11
6.4 Lemma 5 ..... 12
6.5 Proposition 1 ..... 13
7 Proving Proposition 2 ..... 13
8 The Final Result ..... 14

## 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 $x_{1}, x_{1} \oplus x_{2}, \ldots, x_{1} \oplus x_{2} \oplus \cdots \oplus x_{n}$. 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 $[0 . .<k+1]$ instead of $[0 . . k]$. 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 foldl1 :: \(\left({ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a\right) \Rightarrow{ }^{\prime} a\) list \(\Rightarrow{ }^{\prime} a\)
where
    foldl1 \(f(x \# x s)=\) foldl \(f x\) xs
fun scanl1 :: \(\left({ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a\right) \Rightarrow{ }^{\prime}\) a list \(\Rightarrow{ }^{\prime} a\) list
where
    scanl1 \(f\) xs \(=\operatorname{map}(\lambda k\). foldl1 \(f(\) take \(k x s))\)
                        \([1 . .<\) length \(x s+1]\)
```

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

## definition

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

candidate $::\left({ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a\right) \Rightarrow{ }^{\prime} a$ list $\Rightarrow{ }^{\prime} a$ 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

$$
\text { wrap } x=[x]
$$

fun ups :: nat $\Rightarrow$ nat list list
where

$$
\text { ups } n=\operatorname{map}(\lambda k .[0 . .<k+1])[0 . .<n+1]
$$

## 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 $f z s)=$ candidate $g($ map $h z s)$

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 $z s$ are necessary.

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

```
datatype 'a nel
    \(=N E-O n e{ }^{\prime} a\)
    | NE-Cons 'a 'a nel
fun \(n 2 l::\) 'a nel \(\Rightarrow\) 'a list
where
    n2l (NE-One \(x)=[x]\)
\(\mid n 2 l(\) NE-Cons \(x x s)=x \#\) n2l \(x s\)
fun \(12 n::\) 'a list \(\Rightarrow\) 'a nel
```


## where

$$
\begin{aligned}
& 12 n[x]=\text { NE-One } x \\
& \mid l 2 n(x \# x s)=(\text { case xs of }[] \quad \Rightarrow \text { NE-One } x \\
& \mid(-\#-) \Rightarrow \text { NE-Cons } x(l 2 n x s))
\end{aligned}
$$

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

```
lemma non-empty-n2l: n2l xs \not=[]
<proof\rangle
```


## lemma n2l-l2n-id: $x \neq[] \Longrightarrow$ n2l $(l 2 n x)=x$

$\langle p r o o f\rangle$

```
lemma n2l-l2n-map-id:
    assumes \x. x set zs \Longrightarrowx\not=[]
    shows map (n2l \circ l2n) zs=zs
<proof\rangle
```

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: \(x::'a list) (y::'a list).
                        x\not=[]\Longrightarrowy\not=[]\Longrightarrowh(fxy)=g(hx)(hy)
    and A2: \x y. x\not=[]\Longrightarrowy\not=[]\Longrightarrowfxy\not=[]
    and A3: }\x.x\in set zs \Longrightarrowx\not=[
    shows maph(candidate fzs)= candidate g(maphzs)
<proof\rangle
```


## 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 \([0 . .<k]\)
\(\langle p r o o f\rangle\)
lemma divide-and-conquer-induct:
    assumes \(A 1\) : \(P[]\)
        and A2: \(\wedge x . P[x]\)
        and \(A 3: \bigwedge x s y s . \llbracket x s \neq[] ; y s \neq[] ; P x s ; P y s \rrbracket \Longrightarrow P(x s @ y s)\)
    shows \(P\) zs
\(\langle p r o o f\rangle\)
```


## 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 \Longrightarrow set xs \cap set ys = {}
    shows distinct js
<proof\rangle
lemma partitions-sorted:
assumes \(\bigwedge x s\) ys \(x y . \llbracket j s=x s @ y s ; x \in\) set \(x s ; y \in\) set \(y s \rrbracket \Longrightarrow x \leq y\)
shows sorted \(j s\)
\(\langle p r o o f\rangle\)
```


## 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 $g(\operatorname{map} f x s)=\operatorname{map}(g \circ f) x s$
$\langle p r o o f\rangle$
lemma L2: length $(\operatorname{map} f x s)=$ length $x s$
$\langle p r o o f\rangle$
lemma L3: take $k(\operatorname{map} f x s)=\operatorname{map} f($ take $k x s)$
$\langle p r o o f\rangle$
lemma L4: map $f \circ$ wrap $=$ wrap $\circ f$
$\langle p r o o f\rangle$
lemma $L 5: \operatorname{map} f(x s @ y s)=(\operatorname{map} f x s) @(\operatorname{map} f y s)$
$\langle p r o o f\rangle$
lemma L6: $k<$ length $x s \Longrightarrow(\operatorname{map} f x s)!k=f(x s!k)$
$\langle p r o o f\rangle$
lemma L7: $\wedge k . k<$ length $x s \Longrightarrow$ map (nth $x s)[0 . .<k+1]=$ take $(k+1) x s$〈proof〉

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 \(x s \neq[]\)
shows foldl1 \(f(x s\) @ ys) \(=\) foldl1 \(f(\) foldl1 \(f x s \# y s)\)
\(\langle p r o o f\rangle\)
```

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

```
lemma foldl1-induct':
    assumes \(\wedge f x . P f[x]\)
        and \(\bigwedge f x y . P f[x, y]\)
        and \(\bigwedge f x y z z s . P f(f x y \# z \# z s) \Longrightarrow P f(x \# y \# z \# z s)\)
        and \(\wedge f . P f[]\)
    shows \(P f x s\)
〈proof〉
```

lemmas foldl1-induct $=$ foldl1-induct ${ }^{\prime}$ [case-names One Two More Nil]
lemma L8:
assumes associative $f$
and $x s \neq[]$
and $y s \neq[$
shows foldl1 $f(x s$ @ ys) $=f($ foldl1 $f x s)($ foldl1 $f y s)$
$\langle p r o o f\rangle$

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

```
lemma Lemma-2:
    assumes length \(x s=\) length \(y s\)
        and \(\bigwedge k . k<\) length \(x s \Longrightarrow x s!k=y s!k\)
    shows \(x s=y s\)
\(\langle p r o o f\rangle\)
```

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 \(x s \neq[\)
    and \(y s \neq[\)
    shows foldl1 f (maph(xs@ys))
        \(=f(\) foldl1 \(f(\) map \(h x s))(\) foldl1 \(f(\) map \(h y s))\)
\(\langle p r o o f\rangle\)
lemma Lemma-3:
    fixes \(f::{ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a\)
        and \(h::\) nat \(\Rightarrow{ }^{\prime} a\)
    assumes associative \(f\)
    shows map (foldl1 \(f \circ\) map h) (candidate (@) (map wrap \([0 . .<n+1])\) )
```

```
    = candidate f (map h [0..<n+1])
\langleproof\rangle
```


## 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 $f 1$ :: three $\Rightarrow$ three $\Rightarrow$ three
where
f1 $x \quad$ Zero $=x$
|f1 Zero One = One
|f1 $x \quad y=$ Two
fun f2 :: three $\Rightarrow$ three $\Rightarrow$ three
where
f2 $x$ Zero $=x$
$\mid$ f2 $x$ One $=$ One
| f2 $x$ Two $=$ Two
Both functions are associative as is proved by case analysis:
lemma f1-assoc: associative f1
$\langle p r o o f\rangle$
lemma f2-assoc: associative f2
$\langle p r o o f\rangle$
Next, we define two other functions, again according to the original paper. Note that $h 1$ has an extra parameter $k$ which is only implicit in the original paper.

```
fun h1 :: nat }=>\mathrm{ nat }=>\mathrm{ nat }=>\mathrm{ three
where
    h1 kij=(if i=j then One
                            else if j\leqk then Zero
    else Two)
```

fun h2 :: nat $\Rightarrow$ nat $\Rightarrow$ three
where
h2 $i j=(i f i=j$ 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 \(i \leq k\)
shows foldl1 f1 \((\operatorname{map}(h 1 k i)[0 . .<k+1])=\) One
\(\langle p r o o f\rangle\)
```

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\rangle
```

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 \ (f :: three }=>\mathrm{ three }=>\mathrm{ three ) h. associative f
        \Longrightarrow \text { foldl1 f (map h js) = foldl1 f (map h[0..<k+1])}
        and}i\leq
    shows foldl1 f1 (map (h1 ki) js)=One
<proof\rangle
```

```
lemma L10:
    assumes \(\wedge(f::\) three \(\Rightarrow\) three \(\Rightarrow\) three \() h\). associative \(f\)
        \(\Longrightarrow\) foldl1 \(f(\) map \(h\) js \()=\) foldl1 \(f(\operatorname{map} h[0 . .<k+1])\)
        and \(i<k\)
    shows foldl1 f2 (map (h2 i) js) = Two
\(\langle p r o o f\rangle\)
```

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 $j s$ at least once.

```
lemma Figure-4:
    assumes foldl1 f1 (map (h1 k i) js) = One
    and js \not=[]
    shows i\in set js
<proof>
```

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 $j s$ at most once.

```
lemma Figure-5:
    assumes foldl1 f1 (map (h1 k i) js) = One
    and js = xs @ ys
    shows }\neg(i\in\mathrm{ set xs }\wedgei\in\mathrm{ set ys)
<proof>
```

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 $j s$
contains only elements of $[0 . .<k+1]$.

```
lemma Figure-6:
    assumes \(\bigwedge i . i \leq k \Longrightarrow\) foldl1 \(f 1(\operatorname{map}(h 1 k i) j s)=\) One
        and \(i>k\)
    shows \(i \notin\) set \(j s\)
\(\langle p r o o f\rangle\)
```

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 $j s$ is eventually followed by $i+1$.

```
lemma Figure-7:
    assumes foldl1 f2 (map (h2 i) js)= Two
    and js = xs @ys
    and }xs\not=[
    and i= last xs
    shows}(i+1)\in\mathrm{ set ys
\langleproof\rangle
```


### 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 $j s$ contains every $i \leq k$ exactly once, and from Figure- 6 we can furthermore show that $j s$ contains no other elements. Thus, $j s$ 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 $j s$ and $[0 . .<k+1]$ are equal using the results of Figure-4 and Figure-6. Second, we show that $j s$ is a distinct list, i.e. no element occurs twice in $j s$. Since also $[0 . .<k+1]$ is distinct, the multisets of $j s$ and $[0 . .<k+1]$ are equal, and therefore, both lists are permutations.
lemma js-is-a-permutation:

$$
\begin{aligned}
\text { assumes } & \text { A1: } \bigwedge(f:: \text { three } \Rightarrow \text { three } \Rightarrow \text { three }) h \text {.associative } f \\
& \Longrightarrow \text { foldl1 } f(\text { map } h \text { js })=\text { foldl1 } f(\text { map } h[0 . .<k+1])
\end{aligned}
$$

and A2: $j s \neq[]$
shows mset $j s=\operatorname{mset}[0 . .<k+1]$
$\langle p r o o f\rangle$
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: \ixs ys.\llbracketi<k;js=xs@ys;xs\not=[];i=last xs \rrbracket
    \Longrightarrow ( i + 1 ) \in \text { set ys}
    and A2: (r::nat)>0
    and A3: i}+r\leq
    and A4: js = xs @ ys
```

```
    and A5: xs \not=[]
    and A6:i = last xs
    shows}(i+r)\in\mathrm{ set ys
<proof\rangle
```

Since we want to use Lemma partitions-sorted to show that $j s$ 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 \(=\operatorname{mset}[0 . .<k+1]\)
    and A2: \(\bigwedge i x s y s . \llbracket i<k ; j s=x s @ y s ; x s \neq[] ; i=\) last \(x s \rrbracket\)
        \(\Longrightarrow(i+1) \in\) set ys
    and \(A 3: j s=x s @ y s\)
    and \(A 4: i \in\) set \(x s\)
    and \(A 5: j \in\) set \(y s\)
shows \(i \leq j\)
\(\langle p r o o f\rangle\)
```

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

```
lemma \(j s\)-equals-upt- \(k\) :
    assumes A1: mset js \(=\operatorname{mset}[0 . .<k+1]\)
    and A2: \(\bigwedge i x s y s . \llbracket i<k ; j s=x s @ y s ; x s \neq[] ; i=\) last \(x s \rrbracket\)
    \(\Longrightarrow(i+1) \in \operatorname{set} y s\)
    shows \(j s=[0 . .<k+1]\)
\(\langle p r o o f\rangle\)
```

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

```
lemma Lemma-4:
    assumes \(\bigwedge(f::\) three \(\Rightarrow\) three \(\Rightarrow\) three \() h\). associative \(f\)
        \(\Longrightarrow\) foldl1 \(f(\) map \(h j s)=\) foldl1 \(f(\operatorname{map} h[0 . .<k+1])\)
    and \(j s \neq[]\)
shows \(j s=[0 . .<k+1]\)
<proof〉
```


### 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 \(\Rightarrow\) three \(\Rightarrow\) three \()\) h. associative \(f\)
        \(\Longrightarrow \operatorname{map}(\) foldl1 \(f \circ\) map \(h) j s s=\operatorname{scanl1} f(\operatorname{map} h[0 . .<n+1])\)
    and \(\bigwedge j s . j s \in\) set \(j s s \Longrightarrow j s \neq[]\)
shows \(j s s=\) ups \(n\)
\(\langle\) proof \(\rangle\)
```


### 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 $O n e$, and to apply the assumptions.

```
lemma non-empty-candidate-results:
    assumes }\bigwedge(f:: three =>three => three) (xs :: three list)
```



```
        and js \in set (candidate (@) (map wrap [0..<n+1]))
    shows js \not= []
<proof\rangle
```

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 $x s$ 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 \ (f :: three => three }=>\mathrm{ three) (xs :: three list).
```



```
    shows candidate (@) (map wrap [0..<n+1])=ups n
<proof\rangle
```


## 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 $x s]$ 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 (\lambdak. foldl1 g (map (nth xs) [0..<k + 1])) [0..<length xs]
    =map}(\lambdak.foldl1 g (take (k+1) xs)) [0..<length xs
<proof\rangle
```

Compared to the original paper, here, Proposition 2 has the additional assumption that $x s$ 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: \n.candidate (@) (mapwrap \([0 . .<n+1])=\) ups \(n\)
        and A2: associative \(g\)
        and \(A 3: x s \neq[]\)
    shows candidate \(g\) xs \(=\) scanl1 \(g\) xs
\(\langle p r o o f\rangle\)
```


## 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 \()(x s\) :: three list \()\).
        \(\llbracket\) associative \(f ; x s \neq[] \rrbracket \Longrightarrow\) candidate \(f x s=\) scanl1 \(f x s\)
        and associative \(g\)
        and \(y s \neq[]\)
    shows candidate \(g\) ys \(=\) scanl1 \(g\) ys
\(\langle p r o o f\rangle\)
```


## 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.

