Much Ado about Two *(Pearl)*

A Pearl on Parallel Prefix Computation

Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany
voigt@tcs.inf.tu-dresden.de

Abstract

This pearl develops a statement about parallel prefix computation in the spirit of Knuth's 0-1-Principle for oblivious sorting algorithms. It turns out that 0-1 is not quite enough here. The perfect hammer for the nails we are going to drive in is relational parametricity.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Languages, Verification

Keywords 0-1-principle, free theorems, parallel prefix computation, relational parametricity

1. Introduction

Parallel prefix computation is the task to compute, given inputs $x_1, \ldots, x_n$ and an associative operation $\oplus$, the outputs $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n$. It has numerous applications in the hardware and algorithmics fields (Blelloch 1993). There is a wealth of solutions (Sklansky 1960; Brent and Kung 1980; Ladner and Fischer 1980), employing the associativity of $\oplus$ in different ways to realize different trade-offs between certain characteristics of the resulting "circuits", and more keep coming up (Lin and Hsiao 2004; Sheeran 2007). An obvious concern is that for correctness of such new, and increasingly complex, methods. One approach to address this concern is the derivation, using well-understood combinators, of new designs from basic building blocks (Hinze 2004). Another is explicit proof or at least systematic (possibly exhaustive) testing of new solution candidates. Of course, studies of the latter kind must be sufficiently generic, given that in the problem specification neither the type of the inputs $x_1, \ldots, x_n$, nor any specifics (apart from associativity) of $\oplus$ are fixed.

Here the 0-1-Principle of Knuth (1973) comes to mind. It states that if an oblivious sorting algorithm, i.e. one where the sequence of comparisons performed is the same for all input sequences of any given length, is correct on boolean valued input sequences, then it is correct on input sequences over any totally ordered value set. This greatly eases the analysis of such algorithms. Is something similar possible for parallel prefix computation? That is the question we address in this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'08, January 7–12, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-689-9/08/0001. . . $5.00
This is the author's version of the work. The definitive version is available from http://doi.acm.org/10.1145/1328438.1328445.

2. The Problem

We cast the problem and our analysis in the purely functional programming language Haskell (Peyton Jones 2003). Since it is universal, it allows us to precisely capture the notion of an "algorithm" (that may, or may not, be a correct solution to the parallel prefix computation task) in the most general way. It also provides the mathematical expressivity and reasoning techniques that we need. Since Haskell's notation is quite intuitive, we do not pause for a detailed introduction to the language, instead introducing concepts as we go.

The Haskell Prelude (the language's standard library) provides a function

$$\mathit{foldl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to \alpha$$

such that

$$\mathit{foldl1}\ (\oplus)\ [x_1, \ldots, x_n] \quad = \quad (\cdots (x_1 \oplus x_2) \oplus \cdots) \oplus x_n$$

for every binary operation and type-conforming input list. While Haskell allows empty, partially defined, and infinite lists as well, we from now on consider $[\alpha]$ to be the type of all non-empty, finite lists (with elements of appropriate type).[1] The variable α indicates that the function *foldl1* can be used at arbitrary type.

Now we can specify our computation task as follows:

$$\begin{array}{l} \mathit{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\ \mathit{scanl1}\ (\oplus)\ \mathit{xs} = \mathit{map}\ (\lambda k \to \mathit{foldl1}\ (\oplus)\ (\mathit{take}\ k\ \mathit{xs}))\ [1..\mathit{length}\ \mathit{xs}] \end{array}$$

by using Haskell's syntactic sugar $[a..b]$ for the list $[a, a+1, \ldots, b]$ (with $a, b :: \mathit{Int}$ and $a \leq b$) and some further Prelude functions:

$$\begin{array}{ll} \mathit{length} & :: [\alpha] \to \mathit{Int} \\ \mathit{take} & :: \mathit{Int} \to [\alpha] \to [\alpha] \\ \mathit{map} & :: (\alpha \to \beta) \to [\alpha] \to [\beta] \end{array}$$

Here $\mathit{length}\ \mathit{xs}$ gives the length of a list xs, $\mathit{take}\ k\ \mathit{xs}$ gives the first k elements of xs, and $\mathit{map}\ f\ \mathit{xs}$ gives the result of elementwise applying f to xs. Actually, *scanl1* itself is also a Prelude function, though usually with a more efficient implementation than shown above for specification purposes.

[1] See the discussion in Section 7.

The correctness issue we are interested in is whether a given function

$$candidate :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

is semantically equivalent to $scanl1$ for associative operations as first argument. The perfect analog to Knuth's 0-1-Principle would be if this were so provided one can establish that equivalence at least on the boolean type. Unfortunately, it does not hold. Exhaustive testing shows that $x_1 \oplus (x_1 \oplus (x_1 \oplus x_2)) = x_1 \oplus x_2$ for every $x_1, x_2 :: Bool$ and $(\oplus) :: Bool \to Bool \to Bool$ (associative or not). Thus, the function

$$\begin{array}{l} candidate :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\ candidate\ (\oplus)\ [x_1, x_2] = [x_1, x_1 \oplus (x_1 \oplus (x_1 \oplus x_2))] \\ candidate\ (\oplus)\ xs \quad\quad = scanl1\ (\oplus)\ xs \end{array}$$

is equivalent to $scanl1$ at type $Bool$, but clearly not in general (not even for associative $\oplus$).

In the context of systematic search for new solutions to the parallel prefix computation task, Sheeran (2007) observed that verifying a candidate at type $[Int]$, with a certain fixed operation on that type and a certain form of input lists of type $[[Int]]$, is enough to establish general correctness (for arbitrary associative operations). Essentially, this holds because the type $[Int]$ can be used to embed freely generated semigroups over arbitrarily large finite generator sets. Sheeran's observation already has a feel similar to Knuth's principle, except that the type $[Int]$ is much bigger than the boolean one; in particular, it is infinite. Here we want to improve on this. With the boolean type ruled out as above, the best thing we can hope for is a three-valued type as discriminator between good and bad candidates.

3. The Claim

We are going to show that parallel prefix computation enjoys a 0-1-2-Principle. To do so, we use the following Haskell datatype:

$$\mathbf{data}\ Three = Zero \mid One \mid Two$$

Now, for the remainder of the paper, we fix some Haskell function

$$candidate :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

Note that we do not put any restriction on the actual definition of $candidate$, just on its type. Nevertheless, we can prove the following theorem.

THEOREM 1. *If for every associative* $(\oplus) :: Three \to Three \to Three$ *and* $xs :: [Three]$, $candidate\ (\oplus)\ xs$ *and* $scanl1\ (\oplus)\ xs$ *give equal results, then the same holds for every type* τ*, associative* $(\oplus) :: \tau \to \tau \to \tau$*, and* $xs :: [\tau]$.

4. The Proof

4.1 Outline

Rather than relating correctness at type $Three$ to correctness at arbitrary type directly, we perform an indirection via the type $[Int]$. In fact, we go through Sheeran's statement, and prove that one as well. To formulate the indirection, we need another Haskell Prelude function

$$(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha]$$

that concatenates lists of equal type, as well as a function

$$\begin{array}{l} wrap :: \alpha \to [\alpha] \\ wrap\ x = [x] \end{array}$$

that wraps an element into a singleton list, and the following function:

$$\begin{array}{l} ups :: Int \to [[Int]] \\ ups\ n = map\ (\lambda k \to [0..k])\ [0..n] \end{array}$$

Then, we prove two propositions. Theorem 1 arises by combining these.

PROPOSITION 1. *If for every associative* $(\oplus) :: Three \to Three \to Three$ *and* $xs :: [Three]$, $candidate\ (\oplus)\ xs$ *and* $scanl1\ (\oplus)\ xs$ *give equal results, then for every* $n :: Int$ *with* $n \geq 0$,

$$candidate\ (+\!\!+)\ (map\ wrap\ [0..n]) \quad = \quad ups\ n$$

PROPOSITION 2. *If for every* $n :: Int$ *with* $n \geq 0$,

$$candidate\ (+\!\!+)\ (map\ wrap\ [0..n]) \quad = \quad ups\ n$$

then for every type τ*, associative* $(\oplus) :: \tau \to \tau \to \tau$*, and* $xs :: [\tau]$, $candidate\ (\oplus)\ xs$ *and* $scanl1\ (\oplus)\ xs$ *give equal results.*

4.2 A Free Theorem

The key to connecting the behavior of $candidate$ at different types, as required for both propositions we want to prove, is relational parametricity (Reynolds 1983), in the form of free theorems (Wadler 1989). It allows us to derive the following lemma just from the polymorphic type of $candidate$.

LEMMA 1. *For every choice of types* τ_1, τ_2 *and functions* $f :: \tau_1 \to \tau_2$, $(\otimes) :: \tau_1 \to \tau_1 \to \tau_1$, *and* $(\oplus) :: \tau_2 \to \tau_2 \to \tau_2$, *if for every* $x, y :: \tau_1$,

$$f\ (x \otimes y) \quad = \quad (f\ x) \oplus (f\ y)$$

then for every $z :: [\tau_1]$,

$$map\ f\ (candidate\ (\otimes)\ z) \quad = \quad candidate\ (\oplus)\ (map\ f\ z)$$

There is no need to do any proof for this lemma. It can be obtained from the general methodology of free theorems as the result of a completely automated process, with just the type of $candidate$ as input (Böhme 2007a). The crucial point now is to cleverly instantiate types and functions quantified over in the above lemma in such a way that the instantiated versions lend themselves to proving our desired propositions. This is unlikely to be achievable by machine, as it requires conceptual insight.

4.3 Preparatory Material

Figure 1 gives some simple laws about Haskell functions that we are going to use in calculations. The laws are to be read as universally quantified over appropriately typed f, g, k, xs, and ys, with the proviso that $0 \leq k < length\ xs$ in (6) and (7). The Haskell Prelude function

$$(!!) :: [\alpha] \to Int \to \alpha$$

extracts an element at a certain position from a list, counting from position 0 (so that $[0..n]!!k = k$ for every $0 \leq k \leq n$). The notation $(xs!!)$ gives a partial application of $(!!)$ to xs as first argument, i.e., a function that expects an argument of type Int and will return the element at the corresponding position in xs. Interestingly, all laws in Figure 1 except for the last one can be obtained automatically by using the machinery of free theorems. The correctness of (7) should also be clear, just as that of the following lemma of similar spirit.

LEMMA 2. *For every type* τ *and* $xs, ys :: [\tau]$, *if* $length\ xs = length\ ys$ *and for every* $k :: Int$ *with* $0 \leq k < length\ xs$, $xs!!k = ys!!k$, *then* $xs = ys$.

One further statement we need is that for every type τ, associative $(\oplus) :: \tau \to \tau \to \tau$, and $xs, ys :: [\tau]$,

$$foldl1\ (\oplus)\ (xs +\!\!+ ys) = (foldl1\ (\oplus)\ xs) \oplus (foldl1\ (\oplus)\ ys) \quad (8)$$

In fact, for the purposes of this paper the above can be seen as the very definition of associativity.

$$
\begin{array}{rcll}
map\ g\ (map\ f\ xs) & = & map\ (g \cdot f)\ xs & (1)\\
length\ (map\ f\ xs) & = & length\ xs & (2)\\
take\ k\ (map\ f\ xs) & = & map\ f\ (take\ k\ xs) & (3)\\
map\ f \cdot wrap & = & wrap \cdot f & (4)\\
map\ f\ (xs +\!\!+ ys) & = & (map\ f\ xs) +\!\!+ (map\ f\ ys) & (5)\\
(map\ f\ xs)!!k & = & f\ (xs!!k) & (6)\\
map\ (xs!!)\ [0..k] & = & take\ (k+1)\ xs & (7)
\end{array}
$$

Figure 1. Some required laws.

Now we can usefully instantiate Lemma 1 in a way that will benefit our proofs of both Proposition 1 and 2. With some further calculation put in, we obtain the following lemma.

LEMMA 3. *For every type τ, functions $h :: Int \to \tau$ and associative $(\oplus) :: \tau \to \tau \to \tau$, and $n :: Int$ with $n \geq 0$,*

$$
\begin{array}{c}
map\ (foldl1\ (\oplus) \cdot map\ h)\ (candidate\ (+\!\!+)\ (map\ wrap\ [0..n])) \\
= \\
candidate\ (\oplus)\ (map\ h\ [0..n])
\end{array}
$$

PROOF: For every $x, y :: [Int]$,

$$
\begin{array}{l}
foldl1\ (\oplus)\ (map\ h\ (x +\!\!+ y)) \\
\quad = \text{ by (5)} \\
foldl1\ (\oplus)\ ((map\ h\ x) +\!\!+ (map\ h\ y)) \\
\quad = \text{ by (8)} \\
(foldl1\ (\oplus)\ (map\ h\ x)) \oplus (foldl1\ (\oplus)\ (map\ h\ y))
\end{array}
$$

Thus,

$$
\begin{array}{l}
map\ (foldl1\ (\oplus) \cdot map\ h)\ (candidate\ (+\!\!+)\ (map\ wrap\ [0..n])) \\
\quad = \text{ by Lemma 1 with } \tau_1 = [Int],\ \tau_2 = \tau, \\
\qquad\qquad\qquad f = foldl1\ (\oplus) \cdot map\ h,\ \otimes = +\!\!+, \text{ and} \\
\qquad\qquad\qquad z = map\ wrap\ [0..n] \\
candidate\ (\oplus)\ (map\ (foldl1\ (\oplus) \cdot map\ h)\ (map\ wrap\ [0..n])) \\
\quad = \text{ by (1)} \\
candidate\ (\oplus)\ (map\ (foldl1\ (\oplus) \cdot map\ h \cdot wrap)\ [0..n]) \\
\quad = \text{ by (4)} \\
candidate\ (\oplus)\ (map\ (foldl1\ (\oplus) \cdot wrap \cdot h)\ [0..n]) \\
\quad = \text{ by the definitions of } foldl1, wrap, \text{ and function composition} \\
candidate\ (\oplus)\ (map\ h\ [0..n])
\end{array}
$$

4.4 Proving Proposition 1

The key insights on why a three-valued type suffices to distinguish good from bad candidates for parallel prefix computation are encapsulated in the following lemma (respectively, its proof and the illustrations given therein; see also Section 5). Quite nicely, it can be formulated in terms of *foldl1*, i.e., essentially in terms of just one element of the output list of *scanl1* at a time. The statement's lifting to the overall computation of *scanl1* is then the subject of the lemma immediately following it.

LEMMA 4. *Let $js :: [Int]$ and $k :: Int$ with $k \geq 0$. If for every $h :: Int \to Three$ and associative $(\oplus) :: Three \to Three \to Three$,*

$$
foldl1\ (\oplus)\ (map\ h\ js) \quad = \quad foldl1\ (\oplus)\ (map\ h\ [0..k])
$$

then

$$
js \quad = \quad [0..k]
$$

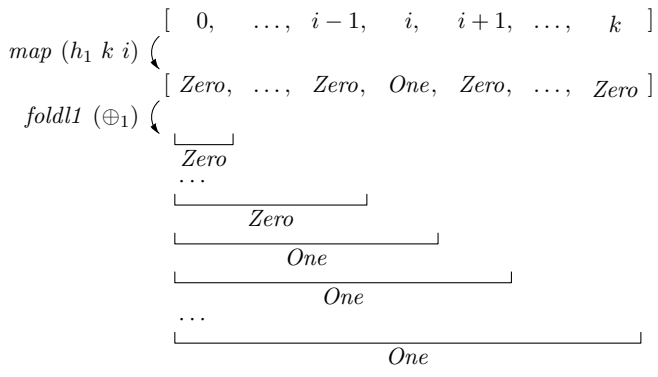

Figure 2. $\forall 0 \leq i \leq k.\ foldl1\ (\oplus_1)\ (map\ (h_1\ k\ i)\ [0..k]) = One$

PROOF: Consider the following two functions:

$$
\begin{array}{l}
(\oplus_1) :: Three \to Three \to Three \\
x \quad \oplus_1\ Zero = x \\
Zero \oplus_1\ One\ = One \\
x \quad \oplus_1\ y \quad = Two
\end{array}
$$

$$
\begin{array}{l}
(\oplus_2) :: Three \to Three \to Three \\
x \oplus_2\ Zero = x \\
x \oplus_2\ One\ = One \\
x \oplus_2\ Two = Two
\end{array}
$$

It is easy to check that both are associative. Further, consider the following two functions:[2]

$$
\begin{array}{l}
h_1 :: Int \to Int \to Int \to Three \\
h_1\ k\ i\ j \mid i \equiv j \qquad\qquad = One \\
\qquad\quad \mid 0 \leq j\ \&\&\ j \leq k = Zero \\
\qquad\quad \mid otherwise \qquad\quad = Two
\end{array}
$$

$$
\begin{array}{l}
h_2 :: Int \to Int \to Three \\
h_2\ i\ j \mid i \equiv j \quad = One \\
\qquad\quad \mid i \equiv j - 1\ = Two \\
\qquad\quad \mid otherwise = Zero
\end{array}
$$

Obviously, for every $i :: Int$ with $0 \leq i \leq k$,

$$
foldl1\ (\oplus_1)\ (map\ (h_1\ k\ i)\ [0..k]) \quad = \quad One
$$

and for every $i :: Int$ with $0 \leq i < k$,

$$
foldl1\ (\oplus_2)\ (map\ (h_2\ i)\ [0..k]) \quad = \quad Two
$$

(see Figures 2 and 3, respectively).

Thus, by the precondition, for every $i :: Int$ with $0 \leq i \leq k$,

$$
foldl1\ (\oplus_1)\ (map\ (h_1\ k\ i)\ js) \quad = \quad One \qquad (9)
$$

and for every $i :: Int$ with $0 \leq i < k$,

$$
foldl1\ (\oplus_2)\ (map\ (h_2\ i)\ js) \quad = \quad Two \qquad (10)
$$

as well.

Then, by (9), we know that js contains every $i :: Int$ with $0 \leq i \leq k$ exactly once (see Figures 4 and 5), and contains no other elements (see Figure 6); i.e., js is a permutation of $[0..k]$. Further, by (10), we know that for every $i :: Int$ with $0 \leq i < k$, every occurrence of i in js is subsequently followed by (but not necessarily directly adjacent to) an occurrence of $i + 1$ (see Figure 7). The only permutation of $[0..k]$ with this property is $[0..k]$ itself.

[2] Here the equality test is written $\equiv$ (in Haskell ==) to distinguish it from the $=$ used for function definition.

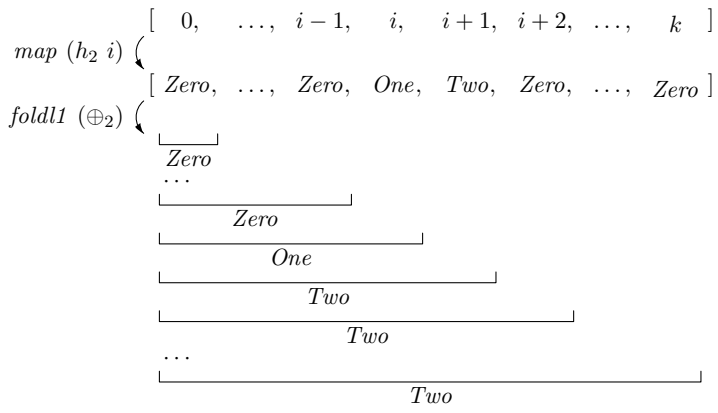

Figure 3. $\forall 0 \leq i < k.\ \mathit{foldl1}\ (\oplus_2)\ (\mathit{map}\ (h_2\ i)\ [0..k]) = \mathit{Two}$

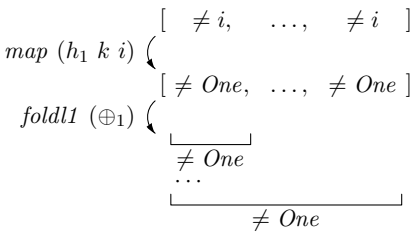

Figure 4. $\mathit{foldl1}\ (\oplus_1)\ (\mathit{map}\ (h_1\ k\ i)\ \mathit{js}) = \mathit{One}$
$\Rightarrow$ "js contains i"

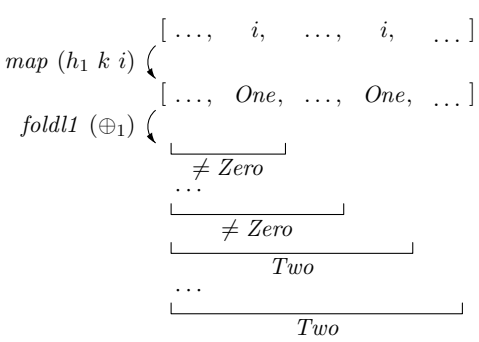

Figure 5. $\mathit{foldl1}\ (\oplus_1)\ (\mathit{map}\ (h_1\ k\ i)\ \mathit{js}) = \mathit{One}$
$\Rightarrow$ "js contains i at most once"

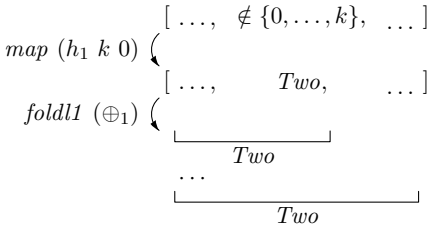

Figure 6. $\mathit{foldl1}\ (\oplus_1)\ (\mathit{map}\ (h_1\ k\ 0)\ \mathit{js}) = \mathit{One}$
$\Rightarrow$ "js contains only $0, \ldots, k$"

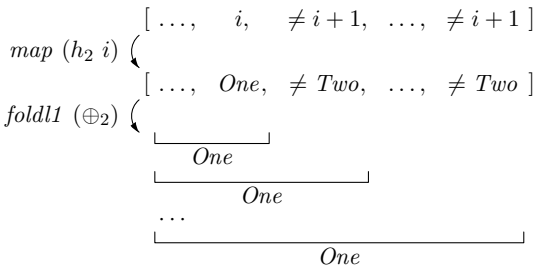

Figure 7. $\mathit{foldl1}\ (\oplus_2)\ (\mathit{map}\ (h_2\ i)\ \mathit{js}) = \mathit{Two}$
$\Rightarrow$ "every i in js is eventually followed by an $i + 1$"

LEMMA 5. *Let* $\mathit{jss} :: [[\mathit{Int}]]$ *and* $n :: \mathit{Int}$ *with* $n \geq 0$. *If for every* $h :: \mathit{Int} \to \mathit{Three}$ *and associative* $(\oplus) :: \mathit{Three} \to \mathit{Three} \to \mathit{Three}$,

$$\mathit{map}\ (\mathit{foldl1}\ (\oplus) \cdot \mathit{map}\ h)\ \mathit{jss} \quad = \quad \mathit{scanl1}\ (\oplus)\ (\mathit{map}\ h\ [0..n])$$

then

$$\mathit{jss} \quad = \quad \mathit{ups}\ n$$

PROOF: We have:

$\mathit{length}\ \mathit{jss}$
$=$ by (2)
$\mathit{length}\ (\mathit{map}\ (\mathit{foldl1}\ (\oplus) \cdot \mathit{map}\ h)\ \mathit{jss})$
$=$ by the precondition
$\mathit{length}\ (\mathit{scanl1}\ (\oplus)\ (\mathit{map}\ h\ [0..n]))$
$=$ by the definition of $\mathit{scanl1}$
$\mathit{length}\ (\mathit{map}\ (\lambda k \to \mathit{foldl1}\ (\oplus)\ (\mathit{take}\ k\ (\mathit{map}\ h\ [0..n])))$
$\quad [1..\mathit{length}\ (\mathit{map}\ h\ [0..n])])$
$=$ by (2)
$\mathit{length}\ [1..\mathit{length}\ (\mathit{map}\ h\ [0..n])]$
$=$ by (2)
$\mathit{length}\ [1..\mathit{length}\ [0..n]]$
$=$ by the definitions of length and $[a..b]$
$\mathit{length}\ [0..n]$
$=$ by (2)
$\mathit{length}\ (\mathit{map}\ (\lambda k \to [0..k])\ [0..n])$
$=$ by the definition of ups
$\mathit{length}\ (\mathit{ups}\ n)$

Moreover, for every $k :: \mathit{Int}$ with $0 \leq k < \mathit{length}\ \mathit{jss}$, $h :: \mathit{Int} \to \mathit{Three}$, and associative $(\oplus) :: \mathit{Three} \to \mathit{Three} \to \mathit{Three}$, we

have:[3]

$$
\begin{array}{l}
\mathit{foldl1}\ (\oplus)\ (\mathit{map}\ h\ (\mathit{jss}!!k)) \\
\ \ = \text{ by (6)} \\
(\mathit{map}\ (\mathit{foldl1}\ (\oplus)\ \cdot\ \mathit{map}\ h)\ \mathit{jss})!!k \\
\ \ = \text{ by the precondition} \\
(\mathit{scanl1}\ (\oplus)\ (\mathit{map}\ h\ [0..n]))!!k \\
\ \ = \text{ by the definition of } \mathit{scanl1} \\
(\mathit{map}\ (\lambda k \rightarrow \mathit{foldl1}\ (\oplus)\ (\mathit{take}\ k\ (\mathit{map}\ h\ [0..n])))\\
\qquad [1..\mathit{length}\ (\mathit{map}\ h\ [0..n])])!!k \\
\ \ = \text{ by (2)} \\
(\mathit{map}\ (\lambda k \rightarrow \mathit{foldl1}\ (\oplus)\ (\mathit{take}\ k\ (\mathit{map}\ h\ [0..n])))\\
\qquad [1..\mathit{length}\ [0..n]])!!k \\
\ \ = \text{ by (6)} \\
(\lambda k \rightarrow \mathit{foldl1}\ (\oplus)\ (\mathit{take}\ k\ (\mathit{map}\ h\ [0..n])))\\
([1..\mathit{length}\ [0..n]]!!k) \\
\ \ = \text{ by the definitions of } [a..b] \text{ and } (!!) \\
\mathit{foldl1}\ (\oplus)\ (\mathit{take}\ (k+1)\ (\mathit{map}\ h\ [0..n])) \\
\ \ = \text{ by (3)} \\
\mathit{foldl1}\ (\oplus)\ (\mathit{map}\ h\ (\mathit{take}\ (k+1)\ [0..n])) \\
\ \ = \text{ by the definitions of } \mathit{take} \text{ and } [a..b] \\
\mathit{foldl1}\ (\oplus)\ (\mathit{map}\ h\ [0..k])
\end{array}
$$

By Lemma 4, this implies $\mathit{jss}!!k = [0..k]$ for every $k :: \mathit{Int}$ with $0 \leq k < \mathit{length}\ \mathit{jss}$. Since also

$$
\begin{array}{l}
(\mathit{ups}\ n)!!k \\
\ \ = \text{ by the definition of } \mathit{ups} \\
(\mathit{map}\ (\lambda k \rightarrow [0..k])\ [0..n])!!k \\
\ \ = \text{ by (6)} \\
(\lambda k \rightarrow [0..k])\ ([0..n]!!k) \\
\ \ = \text{ by the definitions of } [a..b] \text{ and } (!!) \\
[0..k]
\end{array}
$$

for every such k, we have $\mathit{jss} = \mathit{ups}\ n$ by Lemma 2.

The first (and more complicated) half of our 0-1-2-Principle can now be proved using the previous lemma in combination with (more precisely, by further instantiating) the instantiation of the free theorem for $\mathit{candidate}$'s type we prepared in the previous subsection.

PROPOSITION 1. *If for every associative* $(\oplus) :: \mathit{Three} \rightarrow \mathit{Three} \rightarrow \mathit{Three}$ *and* $xs :: [\mathit{Three}]$, $\mathit{candidate}\ (\oplus)\ xs$ *and* $\mathit{scanl1}\ (\oplus)\ xs$ *give equal results, then for every* $n :: \mathit{Int}$ *with* $n \geq 0$,

$$\mathit{candidate}\ (+\!\!+)\ (\mathit{map}\ \mathit{wrap}\ [0..n]) \quad = \quad \mathit{ups}\ n$$

PROOF: Let $n :: \mathit{Int}$ with $n \geq 0$. For every $h :: \mathit{Int} \rightarrow \mathit{Three}$ and associative $(\oplus) :: \mathit{Three} \rightarrow \mathit{Three} \rightarrow \mathit{Three}$,

$$
\begin{array}{l}
\mathit{map}\ (\mathit{foldl1}\ (\oplus)\ \cdot\ \mathit{map}\ h)\ (\mathit{candidate}\ (+\!\!+)\ (\mathit{map}\ \mathit{wrap}\ [0..n])) \\
\ \ = \text{ by Lemma 3} \\
\mathit{candidate}\ (\oplus)\ (\mathit{map}\ h\ [0..n]) \\
\ \ = \text{ by the precondition} \\
\mathit{scanl1}\ (\oplus)\ (\mathit{map}\ h\ [0..n])
\end{array}
$$

By Lemma 5 this implies $\mathit{candidate}\ (+\!\!+)\ (\mathit{map}\ \mathit{wrap}\ [0..n]) = \mathit{ups}\ n$.

4.5 Proving Proposition 2

The remaining half of our 0-1-2-Principle (i.e., Sheeran's original observation) is now just a matter of some calculation and another use of the already partially instantiated free theorem for $\mathit{candidate}$'s type we prepared earlier.

[3] Note that $0 \leq k < \mathit{length}\ [1..\mathit{length}\ [0..n]] = \mathit{length}\ [0..n]$ since we have $0 \leq k < \mathit{length}\ \mathit{jss}$ and just proved that $\mathit{length}\ \mathit{jss} = \mathit{length}\ [1..\mathit{length}\ [0..n]] = \mathit{length}\ [0..n]$.

PROPOSITION 2. *If for every* $n :: \mathit{Int}$ *with* $n \geq 0$,

$$\mathit{candidate}\ (+\!\!+)\ (\mathit{map}\ \mathit{wrap}\ [0..n]) \quad = \quad \mathit{ups}\ n$$

then for every type τ, *associative* $(\oplus) :: \tau \rightarrow \tau \rightarrow \tau$, *and* $xs :: [\tau]$, $\mathit{candidate}\ (\oplus)\ xs$ *and* $\mathit{scanl1}\ (\oplus)\ xs$ *give equal results.*

PROOF: Since

$$
\begin{array}{l}
\mathit{length}\ xs \\
\ \ = \text{ by the definitions of } \mathit{length} \text{ and } [a..b] \\
\mathit{length}\ [0..\mathit{length}\ xs - 1] \\
\ \ = \text{ by (2)} \\
\mathit{length}\ (\mathit{map}\ (xs!!)\ [0..\mathit{length}\ xs - 1])
\end{array}
$$

and for every $k :: \mathit{Int}$ with $0 \leq k < \mathit{length}\ xs$,

$$
\begin{array}{l}
xs!!k \\
\ \ = \text{ by the definitions of } [a..b] \text{ and } !! \\
xs!!([0..\mathit{length}\ xs - 1]!!k) \\
\ \ = \text{ by (6)} \\
(\mathit{map}\ (xs!!)\ [0..\mathit{length}\ xs - 1])!!k
\end{array}
$$

Lemma 2 implies

$$xs \quad = \quad \mathit{map}\ (xs!!)\ [0..\mathit{length}\ xs - 1]$$

Thus,

$$
\begin{array}{l}
\mathit{candidate}\ (\oplus)\ xs \\
\ \ = \text{ by the above argument} \\
\mathit{candidate}\ (\oplus)\ (\mathit{map}\ (xs!!)\ [0..\mathit{length}\ xs - 1]) \\
\ \ = \text{ by Lemma 3 with } h = (xs!!) \text{ and } n = \mathit{length}\ xs - 1 \\
\mathit{map}\ (\mathit{foldl1}\ (\oplus)\ \cdot\ \mathit{map}\ (xs!!))\\
\qquad (\mathit{candidate}\ (+\!\!+)\ (\mathit{map}\ \mathit{wrap}\ [0..\mathit{length}\ xs - 1])) \\
\ \ = \text{ by the precondition} \\
\mathit{map}\ (\mathit{foldl1}\ (\oplus)\ \cdot\ \mathit{map}\ (xs!!))\ (\mathit{ups}\ (\mathit{length}\ xs - 1)) \\
\ \ = \text{ by the definition of } \mathit{ups} \\
\mathit{map}\ (\mathit{foldl1}\ (\oplus)\ \cdot\ \mathit{map}\ (xs!!))\ (\mathit{map}\ (\lambda k \rightarrow [0..k])\\
\qquad\qquad [0..\mathit{length}\ xs - 1]) \\
\ \ = \text{ by (1)} \\
\mathit{map}\ (\lambda k \rightarrow \mathit{foldl1}\ (\oplus)\ (\mathit{map}\ (xs!!)\ [0..k]))\ [0..\mathit{length}\ xs - 1] \\
\ \ = \text{ by (7) and the definitions of } \mathit{map} \text{ and } [a..b] \\
\mathit{map}\ (\lambda k \rightarrow \mathit{foldl1}\ (\oplus)\ (\mathit{take}\ (k+1)\ xs))\ [0..\mathit{length}\ xs - 1] \\
\ \ = \text{ by (1)} \\
\mathit{map}\ (\lambda k \rightarrow \mathit{foldl1}\ (\oplus)\ (\mathit{take}\ k\ xs))\ (\mathit{map}\ (\lambda k \rightarrow k+1)\\
\qquad\qquad [0..\mathit{length}\ xs - 1]) \\
\ \ = \text{ by the definitions of } \mathit{map} \text{ and } [a..b] \\
\mathit{map}\ (\lambda k \rightarrow \mathit{foldl1}\ (\oplus)\ (\mathit{take}\ k\ xs))\ [1..\mathit{length}\ xs] \\
\ \ = \text{ by the definition of } \mathit{scanl1} \\
\mathit{scanl1}\ (\oplus)\ xs
\end{array}
$$

For the uses of Lemma 3 and of the precondition, note that $n = \mathit{length}\ xs - 1 \geq 0$ since $xs :: [\tau]$ is non-empty.

5. Some Reflection

Inside the proof of Lemma 4 we used the lemma's precondition that "for every $h :: \mathit{Int} \rightarrow \mathit{Three}$ and associative $(\oplus) :: \mathit{Three} \rightarrow \mathit{Three} \rightarrow \mathit{Three}$, ..." only for very particular h and $\oplus$. This begs the question whether as a consequence our main result, the 0-1-2-Principle for parallel prefix computation, could similarly be based on weaker requirements than having to check correctness of $\mathit{candidate}$ for every associative operation on Three and every input list over that type. And indeed, tracing the proofs leading up to Theorem 1 very carefully, one finds that what we actually proved is the following theorem.

THEOREM 2. *Let $\oplus_1$, $\oplus_2$, h_1, and h_2 be as in the proof of Lemma 4. If $candidate\ (\oplus)\ xs$ and $scanl1\ (\oplus)\ xs$ give equal results for*

- $\oplus = \oplus_1$ *and* $xs = map\ (h_1\ k\ i)\ [0..n]$ *for every* $n, k, i :: Int$ *with* $0 \leq i \leq k \leq n$*, as well as for*
- $\oplus = \oplus_2$ *and* $xs = map\ (h_2\ i)\ [0..n]$ *for every* $n, i :: Int$ *with* $0 \leq i < n$*,*

then the same holds for every type τ, associative $(\oplus) :: \tau \to \tau \to \tau$, and $xs :: [\tau]$.

In other words, to establish correctness of $candidate$ at arbitrary type, it suffices to show that it delivers correct results for $\oplus_1$ on every input list of the form

$$[(Zero,)^*\ One\ (,Zero)^*\ (,Two)^*]$$

as well as for $\oplus_2$ on every input list of the form

$$[(Zero,)^*\ One, Two\ (,Zero)^*]$$

where the $()^*$-notation means arbitrary, including empty, repetition of elements.

As a further note, the associativity of $(\oplus) :: \tau \to \tau \to \tau$ is essential in both Theorems 1 and 2. Via Proposition 2 it is traced back to Lemma 3. It is easy to see that none of these statements would hold for arbitrary $\oplus$.

6. A Challenge

An interesting extension on our work here would be to consider the situation for operations $\oplus$ that satisfy additional algebraic properties beyond associativity. For example, more circuit optimization is possible in practice if $\oplus$ is idempotent in addition (Stone and Kogge 1973; Lynch and Swartzlander 1991; Knowles 2001), and having a Knuth-like principle for corresponding solution candidates would be interesting as well. For associative, idempotent $\oplus$ the counterexample $candidate$ in Section 2 is actually equivalent to *scanl1*, but maybe there are other counterexamples showing a 0-1-Principle to be impossible here? What is the smallest type that can serve as discriminator between good and bad candidates in this situation? Our feeling is that the answers to these questions will be more difficult to obtain than for the setting of associativity only as considered in the present paper. The reason for this is that the corresponding algebraic structures, so-called bands, appear to have a far more complicated theory than semigroups do.

7. Conclusion

Day et al. (1999) demonstrated that Knuth's 0-1-Principle for oblivious sorting algorithms can be proved as a free theorem, expressed in terms of Haskell. Except for Dybjer et al. (2004) and Bove and Coquand (2006), this has found surprisingly little echo in a community that is otherwise so fond of functional pearls. Even though Day et al.'s paper is not explicitly a pearl, we think that the parts on proving Knuth's principle via relational parametricity, and on how this might open up possibilities for proving statements of a similar spirit, but not specific to sorting, should really be regarded as such. In fact, ever since reading their account, we have been looking for other classes of algorithms that might yield to the same approach. The inspiration for trying that hammer in search of nails on parallel prefix computation is entirely due to Sheeran. In her work on hardware design using functional languages, she has employed the 0-1-Principle for verifying sorting and median networks (Sheeran 2003). She also employs parametric polymorphism in various other ways via so-called non-standard interpretations that help to analyze circuits with respect to different criteria, and even to draw circuit pictures (Sheeran 2005). It is not surprising, then, that she would come up with Proposition 2. We improved on this by replacing the infinite type $[Int]$ as discriminator between good and bad candidates with a three-valued type, which is the best one can hope for, given our counterexample ruling out the boolean type.

Throughout, our reasoning was done under certain simplifying assumptions about the semantics of Haskell types and functions. For example, we restricted attention to non-empty, finite lists, while Haskell allows empty, partially defined, and infinite lists as well, for which Lemma 2 is not necessarily true. More generally, we have completely ignored the presence of undefined values, i.e., of $\bot$, in Haskell. In that sense, our reasoning has been fast and loose, but morally correct (Danielsson et al. 2006). This was done on purpose to expose our main ideas without too much added noise during calculation. Giving a more pedantic account of basically the same material is no doubt possible. It would build on the lessons of Johann and Voigtländer (2004) regarding the possible interactions between $\bot$ and free theorems in Haskell, and potentially on those of Gibbons and Hutton (2005) regarding reasoning about non-finite lists.

Even independently of intricacies involving partial or infinite values, the reader might be worried about the overall correctness of our development, in particular in view of our "proof by pictures" for Lemma 4. Can we really be sure that we have not slipped up somewhere, with potentially dire consequences? Yes we can, thanks to Böhme (2007b), who has reproduced our whole line of reasoning with an interactive proof assistant. That is, there is now a complete proof script for Isabelle (Nipkow et al. 2002) leading from Lemma 1 to Theorem 1.

Acknowledgments

I thank Mary Sheeran for sharing her draft paper on systematic search for new solutions to the parallel prefix computation task (Sheeran 2007). I also thank the POPL reviewers for their feedback.

References

G.E. Blelloch. Prefix sums and their applications. In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufmann, 1993.

S. Böhme. Free theorems for sublanguages of Haskell. Master's thesis, Technische Universität Dresden, 2007a.

S. Böhme. Much ado about two. Formal proof development. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/MuchAdoAboutTwo.shtml`, 2007b.

A. Bove and T. Coquand. Formalising bitonic sort in type theory. In *Types for Proofs and Programs, TYPES 2004, Revised Selected Papers*, volume 3839 of *LNCS*, pages 82–97. Springer-Verlag, 2006.

R.P. Brent and H.T. Kung. The chip complexity of binary arithmetic. In *ACM Symposium on Theory of Computing, Proceedings*, pages 190–200. ACM Press, 1980.

N.A. Danielsson, R.J.M. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Principles of Programming Languages, Proceedings*, pages 206–217. ACM Press, 2006.

N.A. Day, J. Launchbury, and J. Lewis. Logical abstractions in Haskell. In *Haskell Workshop, Proceedings*. Technical Report UU-CS-1999-28, Utrecht University, 1999.

P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology*, 46(15):1011–1025, 2004.

J. Gibbons and G. Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, 2005.

R. Hinze. An algebra of scans. In *Mathematics of Program Construction, Proceedings*, volume 3125 of *LNCS*, pages 186–210. Springer-Verlag, 2004.

P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

S. Knowles. A family of adders. In *Computer Arithmetic, Proceedings*, pages 277–284. IEEE Press, 2001.

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

R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

Y.-C. Lin and J.-W. Hsiao. A new approach to constructing optimal parallel prefix circuits with small depth. *Journal of Parallel and Distributed Computing*, 64(1):97–107, 2004.

T. Lynch and E. Swartzlander. The redundant cell adder. In *Computer Arithmetic, Proceedings*, pages 165–170. IEEE Press, 1991.

T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

S.L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier Science Publishers B.V., 1983.

M. Sheeran. Finding regularity: Describing and analysing circuits that are not quite regular. In *Correct Hardware Design and Verification Methods, Proceedings*, volume 2860 of *LNCS*, pages 4–18. Springer-Verlag, 2003.

M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

M. Sheeran. Searching for prefix networks to fit in a context using a lazy functional programming language. Talk at *Hardware Design and Functional Languages*, 2007.

J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(6):226–231, 1960.

H.S. Stone and P.M. Kogge. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 22(8):786–793, 1973.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.