A calculational derivation of the CASOP algorithm

Rob R. Hoogerwoord
Department of Mathematics and Computing Science
Eindhoven University of Technology
5600 MB Eindhoven
The Netherlands

abstract

A formal derivation is presented of an efficient algorithm for computing the "sums" of all segments, of a given length, of a sequence. Here, "sums" refers to the continued application of a binary operator of which associativity is the only known property. Recurrence relations are used to separate two concerns, viz. characterisation of the values to be computed and choosing the order in which these values will be computed.

keywords: program derivation, formula manipulation, recurrence relations, functional and sequential programming.

Introduction

Let $\oplus$ be an associative binary operator, let N, $1 \leqslant N$, be a natural number, and let $x(i{:}0 \leqslant i)$ and $y(i{:}0 \leqslant i)$ be infinite sequences coupled by

$$(\forall i : 0 \leqslant i : y{\cdot}i \;=\; x{\cdot}i \oplus x{\cdot}(i+1) \oplus \cdots \oplus x{\cdot}(i+N-1)) \quad .$$

I.e., $y{\cdot}i$ is the "sum" of the elements of segment $x(j{:}i \leqslant j < i+N)$.

In a recent publication J. Cooper and L. Kitchen [0] present an efficient algorithm for the computation of sequence y for given N and x . They do not, however, explain *how* they designed their algorithm and, for the alleged sake of brevity, they give only a sketch of the algorithm's proof of correctness. The crucial part of this sketch is a pictorial representation [0, fig.2] of the recurrence relations that lie at the heart of their design. Here we show that both birds -- heuristics and correctness -- can be killed by the same stone, by providing a calculational derivation of these recurrence relations. This derivation turns out to be so short that there is hardly a point in omitting it. Finally, the code of the program can be constructed from the recurrence relations in a straightforward way.

Derivation

A naive solution would not exploit the associativity of $\oplus$; it would require $N-1$ $\oplus$-operations per element of y . A less naive approach is based on the observation that successive elements of y depend on segments of x that have large *subsegments* in common; hence, by avoiding recomputation of the sums of such subsegments, it should be possible to obtain more efficient programs. Because in any solution segments of x with lengths less than N will play a role, we generalise -- this is a standard technique -- the specification of y ; thus we obtain function f defined by

$$(\forall i,j : 0 \leqslant i < j : f{\cdot}i{\cdot}j \;=\; x{\cdot}i \oplus x{\cdot}(i+1) \oplus \cdots \oplus x{\cdot}(j-1)) \quad .$$

In terms of f sequence y is now defined by

$$y{\cdot}i \;=\; f{\cdot}i{\cdot}(i+N) \quad .$$

Moreover, f has the following properties:

(0) $\quad(\forall i : 0 \leqslant i : f{\cdot}i{\cdot}(i+1) \;=\; x{\cdot}i)$

(1) $\quad(\forall i,j,k : 0 \leqslant i < j < k : f{\cdot}i{\cdot}k = f{\cdot}i{\cdot}j \oplus f{\cdot}j{\cdot}k) \quad .$

Note that property (1) captures the associativity of $\oplus$; this associativity will not be referred to anymore: the whole derivation will be carried out in terms of (0) and (1) . Property (1) offers quite a large amount of freedom; our main problem, therefore, is to exploit this freedom judiciously.

The following little calculation embodies the major design decision of Cooper and Kitchen's algorithm; for $i : 0 \leqslant i$, we have:

$y{\cdot}i$

= $\quad$ { definition of y }

$f{\cdot}i{\cdot}(i+N)$

= $\quad$ { $0 \leqslant i$; assume $i < N < i+N$: (1) }

$f{\cdot}i{\cdot}N \oplus f{\cdot}N{\cdot}(i+N)$

The assumption $i < N < i+N$ is equivalent to $0 < i < N$. The way in which we have applied (1) is about the *simplest* possible. Formulae $f{\cdot}i{\cdot}N$ and $f{\cdot}N{\cdot}(i+N)$ have the nice property that they are function applications in which one of the arguments is constant. Hopefully, this enables us to derive simple recurrence relations for them. Moreover, from y's definition it follows that $y{\cdot}0 = f{\cdot}0{\cdot}N$ and $y{\cdot}N = f{\cdot}N{\cdot}(N+N)$; so, the first $N+1$ elements of y can be expressed in terms of $f{\cdot}i{\cdot}N$, $0 \leqslant i < N$, and $f{\cdot}N{\cdot}(i+N)$, $0 < i \leqslant N$.

We now derive a recurrence relation for $f{\cdot}i{\cdot}N$, $0 \leqslant i < N$, as follows:

$$\begin{array}{ll}
 & f{\cdot}i{\cdot}N \\
= & \quad \{ \text{ assume } i < N-1 \text{ , then } 0 \leqslant i < i+1 < N : (1) \ \} \\
 & f{\cdot}i{\cdot}(i+1) \oplus f{\cdot}(i+1){\cdot}N \\
= & \quad \{ \ (0) \ \} \\
 & x{\cdot}i \oplus f{\cdot}(i+1){\cdot}N \quad ,
\end{array}$$

and:

$$\begin{array}{ll}
 & f{\cdot}(N-1){\cdot}N \\
= & \quad \{ \ (0) \ \} \\
 & x{\cdot}(N-1) \quad .
\end{array}$$

By a very similar calculation we can derive a recurrence relation for $f{\cdot}N{\cdot}(i+N)$. Thus we obtain the following relations, which define $y(i{:}0{\leqslant}i{\leqslant}N)$:

$$\begin{array}{lll}
y{\cdot}i & = f{\cdot}i{\cdot}(i+N) & , \ 0 \leqslant i \leqslant N \\
f{\cdot}i{\cdot}(i+N) & = f{\cdot}i{\cdot}N \oplus f{\cdot}N{\cdot}(i+N) & , \ 0 < i < N \\
f{\cdot}i{\cdot}N & = x{\cdot}i \oplus f{\cdot}(i+1){\cdot}N & , \ 0 \leqslant i < N-1 \\
f{\cdot}(N-1){\cdot}N & = x{\cdot}(N-1) & \\
f{\cdot}N{\cdot}(1+N) & = x{\cdot}N & \\
f{\cdot}N{\cdot}(i+1+N) & = f{\cdot}N{\cdot}(i+N) \oplus x{\cdot}(i+N) & , \ 0 < i < N
\end{array}$$

By means of these relations the first $N+1$ elements of y can be computed; if the computations are performed in the right order this requires $3*(N-1)$ $\oplus$-operations. So, on the average $3*(N-1)/(N+1)$ $\oplus$-operations are needed per element of y . Furthermore, *none* of the values occurring in this computation is of any use for the computation of $y(i{:}N{<}i)$; hence, it is not surprising that this gives rise to an algorithm in which y is computed in chunks of length $N+1$. The general case, i.e. the computation of $y(i{:}p{\leqslant}i{\leqslant}p+N)$, for any p , can be dealt with in the same way; this amounts to prefixing all indices in the above formulae with $p+$. This makes all formulae longer, but not more informative, which is why we --deliberately-- did not carry out the formal derivation in full generality.

A program

Based on the recurrence relations derived in the previous section, several programs can be constructed. Here we present one for the computation of $y(i{:}0\leq i\leq N)$, the elements of which are computed in the order of increasing index. The recurrence relations show that the values $f{\cdot}N{\cdot}(i+N)$ can be computed "on the fly", but that, in order to obtain an efficient program, the values $f{\cdot}i{\cdot}N$ must have been computed in advance and stored in an array $z(i{:}0\leq i<N)$, say. This yields the following program (in which z occurs as a constant):

```
{ $1 \leq N \wedge (\forall i : 0 \leq i < N : z{\cdot}i = f{\cdot}i{\cdot}N)$ }
  $y{\cdot}0 , a , n := z{\cdot}0 , x{\cdot}N , 1$
{ invariant: $(\forall i : 0 \leq i < n : y{\cdot}i = f{\cdot}i{\cdot}(i+N)) \wedge 1 \leq n \leq N \wedge a = f{\cdot}N{\cdot}(n+N)$
; bound:    $N - n$
}
; do $n \neq N$ → { $f{\cdot}n{\cdot}(n+N) = z{\cdot}n \oplus a \wedge f{\cdot}N{\cdot}(n+1+N) = a \oplus x{\cdot}(n+N)$ }
                 $y{\cdot}n , a , n := z{\cdot}n \oplus a , a \oplus x{\cdot}(n+N) , n+1$
  od
{ $(\forall i : 0 \leq i < N : y{\cdot}i = f{\cdot}i{\cdot}(i+N)) \wedge a = f{\cdot}N{\cdot}(N+N)$ }
; $y{\cdot}N := a$
{ $(\forall i : 0 \leq i \leq N : y{\cdot}i = f{\cdot}i{\cdot}(i+N))$ }  .
```

Initialisation of array z is straightforward; the following program does the job:

```
{ $1 \leq N$ }
  $z{\cdot}(N-1) , n := x{\cdot}(N-1) , N-1$
{ invariant: $(\forall i : n \leq i < N : z{\cdot}i = f{\cdot}i{\cdot}N) \wedge 0 \leq n < N$  ;  bound: $n$ }
; do $n \neq 0$ → { $f{\cdot}(n-1){\cdot}N = x{\cdot}(n-1) \oplus z{\cdot}n$ }
                 $z{\cdot}(n-1) , n := x{\cdot}(n-1) \oplus z{\cdot}n , n-1$
  od
{ $(\forall i : 0 \leq i < N : z{\cdot}i = f{\cdot}i{\cdot}N)$ }  .
```

These programs allow several interesting embellishments. It is, for instance, possible to modify them in such a way that the elements of x are inspected only once, in the order of increasing index, during the computation of the whole sequence y . This requires some buffering for which the same array z can be used. We leave such embellishments as exercises to the interested reader.

Conclusion

The best order in which many values are to be computed depends on the relations between these values. Therefore, it is a wise strategy to identify these relations before deciding on the order of computation. The example discussed in this paper confirms (again) that recurrence relations provide a suitable interface between these different concerns. For example, the invariant of the program for the initialisation of z has been obtained from its postcondition by replacing constant 0 by a variable; the choice *which* constant should be replaced is enforced by the recurrence relations. In this respect, it is somewhat surprising that the usefulness of recurrence relations seems to be under-estimated in methodological discussions on sequential programming.

Recurrence relations are relations between the values of functions in different points of their domains. Deriving such relations can be considered as a form of functional programming. Thus, although the example discussed in this paper is too simple to demonstrate this, functional programming techniques can be used in sequential programming.

reference

[0] J. Cooper, L. Kitchen
CASOP: a fast algorithm for computing the n-ary composition of a binary associative operator
Information Processing Letters 34(1990), pp. 209-213.

(Eindhoven, 6 June 1990)