ttp/papers/working/squigol.tex

Tom Pressburger
January 18, 1988

Contents

1 Introduction

This paper reviews an algebraic language (SQUIGOL), a theory of lists, and an equational style of reasoning (due to Richard S. Bird [1] and Lambert Meertens [3]), and then carries out some derivations due to Douglas R. Smith [4]. I first introduce the operators of the algebraic language, along with some of their useful properties, and present a few results from Bird's Theory of Lists. Then I give (partial) derivations for divide and conquer algorithms for the *maximum sum subsegment* and *maximum sum subrectangle* problems. The *tupling* program transformation proves useful, and is described.

1.1 Related Work

Bird gave a derivation of the minimum sum subsegment algorithm in the equational style, using a singleton-split divide-and-conquer algorithm. Smith gave derivations of both the maximum sum subsegment and maximum sum subrectangle algorithms using a middle-split algorithm. Smith and Richard King explored the possibilities for pipelining in Smith's middle-split algorithm. Edgser W. Dijkstra presented a derivation of the minimum sum subrectangle problem in his note EWD900a, December 2, 1984[1]. William McColl (Oxford) noticed that a standard heuristic, the "sweep-technique", gives a reasonable algorithm for the subrectangle problem, so perhaps the problem wasn't so hard in the first place. [The sweep technique corresponds to a singleton-split divide-and-conquer algorithm.?]

2 Operators and Properties

2.1 Notation

For integers l and u, the set $\{l\,..\,u\}$ denotes the set of integers i such that $l \leq i \leq u$. Sequences will be considered finite total functions with domain $\{1\,..\,n\}$ for some $0 \leq n$. The sequence $[l\,..\,u]$ is $[l, l+1, \ldots, u-1, u]$, if $l \leq u$, else $[l\,..\,u] = [\,]$.

The variables f, g, and h denote unary functions. The variables P, Q will denote predicates. Prefix function application is written $f\ x$. This is similar to mathematical notation, e.g. $\log x$, $\sin x$. Parentheses will be added to clarify grouping. All functions will be unary, though a function may apply to a single tuple of values. The application of f to a pair $\langle x, y\rangle$ is written $f\ \langle x, y\rangle$. The symbols $\oplus$ and $\otimes$ will denote associative functions that apply to pairs. An application of such a function could be written $\oplus\ \langle x, y\rangle$, but the infix function application $x \oplus y$ often looks better. The same function symbol is often overloaded to express the curried form of the function. For example, $3+$ is the function that adds 3 to its argument. Operators that combine objects will be overloaded to also combine functions.

$$\begin{array}{rcll}
(x\oplus) & = & \lambda y.x \oplus y & \\
(\oplus y) & = & \lambda x.x \oplus y & \\
P \wedge Q & = & \lambda x.(P\ x) \wedge (Q\ x) & \\
\langle f, g\rangle & = & \lambda x.\langle f\ x, g\ x\rangle & \\
[f, g, h] & = & \lambda x.[f\ x, g\ x, h\ x] & \\
f{\times}g & = & \lambda\langle x, y\rangle.\langle f\ x, g\ y\rangle & f{\times}g\ \langle x, y\rangle = \langle f\ x, g\ y\rangle \\
K_c & = & \lambda x.c & f \circ K_c = K_{(f\ c)} \\
\mathbf{id} & = & \lambda x.x & \\
\widetilde{\oplus} & = & \lambda\langle x, y\rangle.y \oplus x & x \widetilde{\oplus} y = y \oplus x \\
(P \longrightarrow f; g) & = & \lambda x.\mathbf{if}\ P\ x\ \mathbf{then}\ f\ x\ \mathbf{else}\ g\ x &
\end{array}$$

Note that if $\oplus$ is associative, commutative, or idempotent, then so is $\widetilde{\oplus}$.

Recursive functions will be defined by giving equations that handle the various input cases; e.g. for the empty sequence, a singleton sequence, or for a sequence that is the concatenation of two other non-empty sequences. In the latter case, the equation will hold for any of the ways of breaking up the sequence into the two others, though there may be a difference in efficiency. I assume that the sequences are implemented so that breaking them up into contiguous pieces takes constant

[1]In EWD900a, Dijkstra refers to a derivation of minimum sum subsegment in EWD897, but my copy of EWD897 is about another subject. End of footnote.

time. This is true for front-singleton-splits for sequences implemented as lists, and for any split, if the sequence is implemented in an array.

2.2 Type Language

The following types and type constructors are provided, where α and β stand for any types:

$\mathbf{B}$	$\{\mathbf{true}, \mathbf{false}\}$ i.e. booleans
$\mathbf{N}$	$\{0, 1, \ldots\}$ i.e. the natural numbers
$\mathbf{N}^+$	$\{1, 2, \ldots\}$ i.e. the positive numbers
$\mathbf{Z}$	$\{0, 1, -1, \ldots\}$ i.e. the integers
$[\alpha]$	sequences whose elements have type α
$\{\alpha\}$	sets whose elements have type α
$\alpha \to \beta$	maps from elements of type α to elements of type β.
$\alpha \times \cdots \times \omega$	Tuples whose first element has type α, etc.

Syntactically, $\times$ binds more tightly than $\to$, so that $\alpha \times \beta \to \delta$ parses as $(\alpha \times \beta) \to \delta$. Also, $\to$ associates to the right, so that $\alpha \to \beta \to \delta$ parses as $\alpha \to (\beta \to \delta)$.

2.3 Operators

The following describes some of the operators we will be dealing with.

minimum	$\_ \downarrow \_ : \mathbf{Z} \times \mathbf{Z} \to \mathbf{B}$	$x \downarrow y$	$=$	$\mathbf{if}\ x < y\ \mathbf{then}\ x\ \mathbf{else}\ y$
maximum	$\_ \uparrow \_ : \mathbf{Z} \times \mathbf{Z} \to \mathbf{B}$	$x \uparrow y$	$=$	$\mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y$
compose	$\_ \circ \_ : (\beta \to \delta) \times (\alpha \to \beta) \to (\alpha \to \delta)$	$(f \circ g)\ x$	$=$	$f\ (g\ x)$
concatenation	$\_ {+\!\!+} \_ : [\alpha] \times [\alpha] \to [\alpha]$			
length	$\#\_ : [\alpha] \to \mathbf{N}$			
cardinality	$\#\_ : \{\alpha\} \to \mathbf{N}$			
sequify	$[]\_ : \alpha \to [\alpha]$	$[]x$	$=$	$[x]$
setify	$\{\}\_ : \alpha \to \{\alpha\}$	$\{\}x$	$=$	$\{x\}$
pair	$\_\lozenge\_ : \alpha \times \beta \to \alpha \times \beta$	$x \lozenge y$	$=$	$\langle x, y \rangle$
image of f	$f : \alpha \to \beta \Rightarrow f \star \_ : [\alpha] \to [\beta]$	$f \star [x_1, \ldots, x_n]$	$=$	$[f\ x_1, \ldots, f\ x_n]$
reduce by $\oplus$	$\oplus : \alpha \times \alpha \to \alpha \Rightarrow \oplus/\_ : [\alpha] \to \alpha$	$\oplus/[x_1, \ldots, x_n]$	$=$	$x_1 \oplus \cdots \oplus x_n$
cross by $\otimes$	$\otimes : \alpha \times \alpha \to \beta \Rightarrow \_ \times_{\otimes} \_ : [\alpha] \times [\alpha] \to [\beta]$			

$$[x_1, \ldots, x_n] \times_{\otimes} [y_1, \ldots, y_n] = [x_1 \otimes y_1, \ldots, x_1 \otimes y_n, \ldots, x_n \otimes y_1, \ldots, x_n \otimes y_n]$$

In many applications, the order of the elements returned by the $\times_{\oplus}$ operator is unimportant, so we can think of it returning a bag of elements. Some of the properties below are only true when we think of its output is unordered. We can also think of this operator applying to two bags or two sets, returning bags or sets.

2.3.1 Reduction

The expression $\oplus/[\,]$ is defined to be $e_{\oplus}$, i.e. the identity element for $\oplus$. Also $\oplus/[x] = x$, for any operator $\oplus$. The operator $\oplus$ must be associative. If it is also commutative, then the order of the elements in the sequence does not matter, so for any permutation s_2 of s_1, $\oplus/s_1 = \oplus/s_2$. In this case, we can overload $\oplus/$ so that it also applies to a bag of values[2]. If $\oplus$ is also idempotent (i.e. $(\forall x) x \oplus x = x$, e.g. $\cup$), then duplicates do not matter, so that if $range(s_2) = range(s_1)$, then $\oplus/s_1 = \oplus/s_2$. In this case, we can overload $\oplus/$ so that it also applies to a set of values.

[2]Perhaps we should allow it to apply to a set of values, too.

The function $+/$ applied to a sequence returns the sum of the elements of the sequence. A convenient shorthand for this function used extensively below is $\sum$; i.e. $\sum s = +/s$. Similarly, $\prod s = */s$.

One can "implement" the $\oplus/$ operator in several ways. Because $\oplus$ is assumed to be associative, the grouping is unimportant. Hence, we may evaluate

$$\oplus/[x_1, x_2, x_3, x_4] = x_1 \oplus (x_2 \oplus (x_3 \oplus x_4)).$$

This corresponds to defining $\oplus/$ in terms of g as follows:

$$\begin{array}{rcl} \oplus/s & = & g\,s \\ g\,[\,] & = & e_\oplus \\ g\,[x] +\!\!+ s_1 & = & x \oplus g\,s_1 \end{array}$$

A different grouping is

$$\oplus/[x_1, x_2, x_3, x_4] = ((x_1 \oplus x_2) \oplus x_3) \oplus x_4.$$

This corresponds to computing $\oplus/$ in terms of f with an "accumulating" variable A as follows:

$$\begin{array}{rcl} \oplus/s & = & f\,\langle s, e_\oplus\rangle \\ f\,\langle [\,], A\rangle & = & A \\ f\,\langle [x] +\!\!+ s_1, A\rangle & = & f\,\langle s_1, A \oplus x\rangle. \end{array}$$

Another possibility is to evaluate $\oplus/[x_1, x_2, x_3, x_4] = (x_1 \oplus x_2) \oplus (x_3 \oplus x_4)$. The depth of the expression tree for $\oplus/s$ would be $\log \# s$, hence this approach takes only logarithmic time on a parallel processor.

$$\begin{array}{rcll} \oplus/s & = & b\,s & \\ b\,[\,] & = & e_\oplus & \\ b\,[x] & = & x & \\ b\,s_1 +\!\!+ s_2 & = & b\,s_1 \oplus b\,s_2 & \text{if } 1 < \#(s_1 +\!\!+ s_2) \wedge \# s_1 = \lfloor \frac{\#(s_1 +\!\!+ s_2)}{2} \rfloor \end{array}$$

2.4 Properties of the Operators

	$*$	$+$	$\downarrow$	$\uparrow$	$+\!\!+$	$\circ$	$\times_\oplus$	$f\star$	$\oplus/$	$\#$
associative	y	y	y	y	y	y	if $\oplus$ is			
commutative	y	y	y	y	n	n	if $\oplus$ is			
idempotent	n	n	y	y	n	n	n			
identity	1	0	∞	$-\infty$	$[\,]$	$\lambda x.x$	$[e_\oplus]$	$\lambda x.x$		
fixpoint	0		$-\infty$	∞			$[\,]$			
distributes over	$+$	$\downarrow$,$\uparrow$	$\uparrow$,$\downarrow$	$\downarrow$,$\uparrow$		$\star$	$+\!\!+$	$+\!\!+$	$+\!\!+$	$+\!\!+$

$$\begin{array}{rcll} x * (y \downarrow z) & = & (x * y) \downarrow (x * z) & \text{if } x \geq 0 \\ & & (x * y) \uparrow (x * z) & \text{if } x < 0 \\ (f \circ g) \star x & = & f \star (g \star x) & \\ s_1 \times_\otimes s_2 & = & \otimes \star (s_1 \times_\between s_2) & \\ [x] \times_\otimes s & = & (x\otimes) \star s & \\ s \times_\otimes [x] & = & (\otimes x) \star s & \\ f \star (s_1 \times_\otimes s_2) & = & s_1 \times_{f \circ \otimes} s_2 & \\ (g \star s_1) \times_\otimes (g \star s_2) & = & s_1 \times_{\otimes \circ g \times g} s_2 & \\ \oplus/(s_1 \times_\otimes s_2) & = & (\oplus/s_1) \otimes (\oplus/s_2) & \text{if } \otimes \text{ distributes over } \oplus,\ \oplus \text{ commutative} \end{array}$$

Some properties may be expressed more neatly in a functional style.

$$\begin{array}{rcll}
(\oplus/)\circ[] & = & \mathbf{id} & \\
(f\star)\circ[] & = & []\circ f & \\
(f\circ g)\star & = & (f\star)\circ(g\star) & \\
\mathsf{X}_{\otimes} & = & (\otimes\star)\circ\mathsf{X}_{\lozenge} & \\
(f\star)\circ\mathsf{X}_{\otimes} & = & \mathsf{X}_{f\circ\otimes} & \\
\mathsf{X}_{\otimes}\circ g\star{\times}g\star & = & \mathsf{X}_{\otimes\circ g\times g} & \\
(\oplus/)\circ\mathsf{X}_{\otimes} & = & \otimes\circ\oplus/{\times}\oplus/ & \text{if } \otimes \text{ distributes over } \oplus
\end{array}$$

Any associative, commutative, idempotent operator distributes over itself.

Because we will be making heavy use of $+\!\!+$, $\uparrow$, $+$ and $\sum$ in what follows, we reiterate some of their important properties.

$$\begin{array}{rcl}
\sum(s_1+\!\!+s_2) & = & \sum s_1+\sum s_2\\
\sum\star(s_1\,\mathsf{X}_{+\!\!+}s_2) & = & (\sum\star s_1)\mathsf{X}_{+}(\sum\star s_2)\\
x+(y\uparrow z) & = & (x+y)\uparrow(x+z)\\
\uparrow/(s_1\,\mathsf{X}_{+}s_2) & = & (\uparrow/s_1)+(\uparrow/s_2)
\end{array}$$

Note that the number of additions on the left side of the last equation is $(\#s_1)*(\#s_2)$ but only 1 on the right side.

2.5 Homomorphisms

A function h is a $\oplus$-*homorphism*, where $\oplus$ is associative, if there is a function $\otimes$ satisfying

$$\begin{array}{rcll}
h(x_1\oplus x_2) & = & (h\,x_1)\otimes(h\,x_2) & \text{or, expressed functionally,}\\
h\circ\oplus & = & \otimes\circ h{\times}h & \text{or also}\\
h\,\oplus/[x_1,x_2] & = & \otimes/h\star[x_1,x_2] &
\end{array}$$

If $\oplus$ is associative, commutative, or idempotent, then so will be $\otimes$. Also, if $e_\oplus$ is the identity of $\oplus$, then $h\,\oplus$ is the identity of $\otimes$.

The last equation above suggests a generalization that we prove below, namely that if $h\,(\oplus/s)=\otimes/h\star s$ for sequences s of length 2, then in fact it holds for sequences of any length. The cases for sequences of length 0 and 1 hold independently of h. Assume it holds for sequences of length $n-1$, and assume s is length n. Then

$$\begin{array}{rcll}
h\,(\oplus/s) & = & h\,(\oplus/[x_1,\ldots,x_n]) & \\
 & = & h\,(x_1\oplus\oplus/[x_2,\ldots,x_n]) & \\
 & = & (h\,x_1)\otimes h\,(\oplus/[x_2,\ldots,x_n]) & \text{which is, by induction,}\\
 & = & (h\,x_1)\otimes((h\,x_2)\otimes\cdots\otimes(h\,x_n)) & \text{by associativity of }\otimes\\
 & = & \otimes/[h\,x_1,\ldots,h\,x_n] & \\
 & = & \otimes/h\star[x_1,\ldots,x_n] & \\
 & = & \otimes/h\star s &
\end{array}$$

Expressed functionally, this is $h\circ(\oplus/)=(\otimes/)\circ(h\star)$.

Now we show that $h\star$ is a $\mathsf{X}_{\oplus}$-homomorphism:

$$(h\star)\circ\mathsf{X}_{\oplus}=\mathsf{X}_{h\circ\oplus}=\mathsf{X}_{\otimes\circ h\times h}=\mathsf{X}_{\otimes}\circ h\star{\times}h\star.$$

For n arguments:

$$\begin{array}{rcll}
h\star\mathsf{X}_{\oplus}/s & = & \mathsf{X}_{\otimes}/(h\star)\star s & \text{or, expressed functionally,}\\
(h\star)\circ(\mathsf{X}_{\oplus}/) & = & (\mathsf{X}_{\otimes}/)\circ(h\star)\star &
\end{array}$$

Every one-to-one function h is a $\oplus$-homomorphism, with $u{\otimes}v{=}h((h^{-1}\,u){\oplus}(h^{-1}\,v))$.

2.6 Sequence Homomorphisms

Every sequence s satisfies $s = +\!\!+/[] \star s$. This implies that every $+\!\!+$-homomorphism h, where $h \circ +\!\!+ = \otimes \circ h{\times}h$ may be expressed as $(\otimes/) \circ (f\star)$ where $f = h \circ []$. The following reasoning justifies this:

$$h\, s = h\,(+\!\!+/[] \star s) = \otimes/h \star [] \star s = \otimes/(h \circ []) \star s$$

Thus, a $+\!\!+$-homomorphism h is completely defined by: (1) its action on singletons; and (2) $\otimes$.

We also note that the right inverse under $\circ$ of $+\!\!+/$ is $[]\star$, because

$$(+\!\!+/) \circ ([]\star) = \mathbf{id}.$$

[When we can obtain an inverse f of $\oplus/$, we learn something about $\oplus$-homomorphisms, because then

$$h = h \circ \oplus/ \circ f = \otimes/ \circ (h\star) \circ f.$$

For example, the right-inverse of $*/$ is the function **prime-factors**; the inverse of $+/$ is the function $\lambda n.[\underbrace{1, \ldots, 1}_{n}]$. So a $*$-homomorphism is determined by $\otimes$ and its action on primes, and a $+$-homomorphism is determined by $\otimes$ and its action on 1.]

2.6.1 Sequence Examples

A familiar homomorphism on sequences is the length function $\#$. Because $\# \circ +\!\!+ = + \circ \#{\times}\#$ and $\# \circ [] = K_1$, we have $\# = (+/) \circ (K_1\star)$. Also $\# \circ (+\!\!+/) = (+/) \circ \#\star$.

For any function f, the function $f\star$ is a $+\!\!+$-homomorphism. Also, for any operator $\oplus$, $\oplus/$ is a $+\!\!+$-homomorphism.

$$\begin{array}{rcl} (f\star) \circ (+\!\!+/) & = & (+\!\!+/) \circ (f\star)\star \\ (\oplus/)+\!\!+(+\!\!+/) & = & (+\!\!+/) \circ (\oplus/)\star \end{array}$$

A familiar injective function is the function **rev** (reverse of a sequence). Because **rev** is one-to-one, it is a $+\!\!+$-homomorphism. Because $\mathbf{rev}^{-1} = \mathbf{rev}$, we have

$$\begin{array}{rcl} x \otimes y & = & \mathbf{rev}((\mathbf{rev}\, x)+\!\!+(\mathbf{rev}\, y)) = \widetilde{+\!\!+} \\ f & = & \mathbf{rev} \circ [] = [] \end{array}$$

so therefore $\mathbf{rev}\, s = \widetilde{+\!\!+}/[] \star s$. The reader may find it interesting to examine the various ways to evaluate the reduction. The classic paper on fold/unfold transformations [2] shows how one can transform one form of reverse into the accumulating reverse. We see that both are implementations of the reduction operator.

The function "group by zeroes" **gbz** is a homomorphism:

$$\mathbf{gbz}[1,2,2,0,9,6,0,4,0,0,3] = [[1,2,2,0],[9,6,0],[4,0],[0],[3]].$$

The function **sort** is a $+\!\!+$-homorphism. It turns out that $\mathbf{sort}\, s = \mathbf{merge}/[] \star s$, where the function **merge** merges two sorted sequences to form a sorted sequence. A binary split implementation of the reduce yields MergeSort, and a singleton-split implementation (either one) yields InsertionSort.

2.7 Subsequence Operators

A *subsequence* of a sequence s is a not-necessarily contiguous selection of elements from s. A *(sub)segment* of s is a contiguous subsequence of s. If s is a sequence, then the *subsegment of s from g to d* is $s \circ [g \,..\, d]$. In particular, $s = s \circ [1 \,..\, \#s]$.

$$\begin{aligned}
\mathbf{seqs}\, s &= \text{all subsequences of } s \\
\mathbf{inits}\, [x_1, \ldots, x_n] &= [[x_1], [x_1, x_2], \ldots, [x_1, \ldots, x_n]] \\
\mathbf{tails}\, [x_1, \ldots, x_n] &= [[x_1, \ldots, x_n], [x_2, \ldots, x_n], \ldots, [x_n]] \\
\mathbf{segs}\, s &= \text{all nonempty subsegments of } s
\end{aligned}$$

$$\begin{aligned}
\mathbf{seqs}\, s &= \times_{++}/(\lambda x.[[\,], [x]]) \star s \qquad\qquad \text{or, expressed functionally,} \\
\mathbf{seqs} &= (\times_{++}/) \circ [K_{[\,]}, []]\star \\
\mathbf{inits}\, [x] &= [[x]] \\
\mathbf{inits}\, s_1 {++} s_2 &= \mathbf{inits}\, s_1 {++} ([s_1] \times_{++} \mathbf{inits}\, s_2) \\
\mathbf{tails}\, [x] &= [[x]] \\
\mathbf{tails}\, s_1 {++} s_2 &= ((\mathbf{tails}\, s_1) \times_{++} [s_2]) {++} \mathbf{tails}\, s_2 \\
\mathbf{segs}\, [x] &= [[x]] \\
\mathbf{segs}\, s_1 {++} s_2 &= (\mathbf{segs}\, s_1) {++} (\mathbf{segs}\, s_2) \\
&\quad {++} ((\mathbf{tails}\, s_1) \times_{++} (\mathbf{inits}\, s_2))
\end{aligned}$$

The order of the results of **segs** and **seqs** is usually unimportant. Perhaps they should return a bag, or set, in which case the ++ above should be the appropriate operator.

2.8 Tupling

The tupling transformation was identified in [2]. An instance of it's use is given below.

Suppose $f\ (s{++}t) = C_1\ \langle f\ s, f\ t, C_2\ \langle g\ s, h\ t\rangle\rangle$ for functions g, h, and inexpensive C_1 and C_2. This is almost an efficient divide-and-conquer algorithm, except for the C_2 term. If g and h are expensive, the recurrence for f may be expensive. However, an inexpensive computation for f may be achieved if computations for g and h can be combined with f's.

A more precise performance analysis proceeds as follows. Let $c_f\, n$ be the cost of computing f on a sequence of length n; similarly define c_g and c_h. If the cost of splitting the input and computing C_1 and C_2 is constant, say c, then the following recurrence holds:

$$c_f\, n = 2 * (c_f\, \frac{n}{2}) + (c_g\, \frac{n}{2}) + (c_h\, \frac{n}{2}) + c.$$

In particular, if c_g and c_h are linear, then c_f is $n \log n$.

Suppose $g\ (s{++}t){=}C_g\ \langle g\ s, g\ t\rangle$ and $h\ (s{++}t){=}C_h\ \langle h\ s, h\ t\rangle$. Then define $H\ s = \langle f\ s, g\ s, h\ s\rangle$. If there were an efficient form for H, then we could compute f by $f\ s = (H\ s).1$. It turns out that H can be computed by the divide-and-conquer algorithm:

```
$H s{++}t =$
        let $\langle s_f, s_g, s_h\rangle = H\ s, \langle t_f, t_g, t_h\rangle = H\ t$
          in $\langle C_1\ \langle s_f, t_f, C_2\ \langle s_g, t_h\rangle\rangle, C_g\ \langle s_g, t_g\rangle, C_h\ \langle s_h, t_h\rangle\rangle$
```

For the cost assumptions above, c_H, the cost of H, is linear; hence c_f is linear.

3 Equational Derivation of the Maximum Sum Subsegment Problem

Given a sequence s of integers find the sum of the maximum sum contiguous subsequence, i.e.

$$\mathbf{MSS}\ s = \uparrow/\sum\star\,\mathbf{segs}\ s$$

We will proceed by equational reasoning on **MSS** acting on the input s split into two pieces such that $s = s_1 +\!\!+ s_2$. If s cannot be split into two pieces, i.e. $s = [x]$, then

$$\mathbf{MSS}\ [x] = \uparrow/\sum\star\,\mathbf{segs}\ [x] = \uparrow/\sum\star\,[[x]] = \uparrow/[\sum[x]] = \uparrow/[x] = x$$

This assumes that the empty sequence is not one of the subsegments of s; if **segs** is defined to include the empty sequence, then if x is negative, the maximum sum subsegment is $[\,]$, which has sum 0.

$$\begin{array}{lcl}
\mathbf{MSS}\ s_1 +\!\!+ s_2 & = & \uparrow/\sum\star\boxed{\mathbf{segs}\ (s_1 +\!\!+ s_2)} \\
 & = & \uparrow/\sum\star\,(\boxed{\mathbf{segs}\ s_1} +\!\!+ \boxed{\mathbf{segs}\ s_2}) +\!\!+ \boxed{(\mathbf{tails}\ s_1) \times_{+\!\!+} (\mathbf{inits}\ s_2)}) \\
 & = & \uparrow/\boxed{\sum\star\,\mathbf{segs}\ s_1} +\!\!+ \boxed{\sum\star\,\mathbf{segs}\ s_2} +\!\!+ \boxed{\sum\star\,((\mathbf{tails}\ s_1) \times_{+\!\!+} (\mathbf{inits}\ s_2))} \\
 & = & \boxed{\uparrow/(\sum\star\,\mathbf{segs}\ s_1)} \uparrow \boxed{\uparrow/(\sum\star\,\mathbf{segs}\ s_2)} \uparrow \boxed{\uparrow/(\sum\star\,((\mathbf{tails}\ s_1) \times_{+\!\!+} (\mathbf{inits}\ s_2)))} \\
 & = & \mathbf{MSS}\ s_1 \uparrow \mathbf{MSS}\ s_2 \uparrow \boxed{\uparrow/\sum\star\,((\mathbf{tails}\ s_1) \times_{+\!\!+} (\mathbf{inits}\ s_2))} \\
 & = & \mathbf{MSS}\ s_1 \uparrow \mathbf{MSS}\ s_2 \uparrow\ \uparrow/(\boxed{\sum\star\,\mathbf{tails}\ s_1} \times_{+} \boxed{\sum\star\,\mathbf{inits}\ s_2}) \\
 & = & \mathbf{MSS}\ s_1 \uparrow \mathbf{MSS}\ s_2 \uparrow\ (\boxed{\uparrow/\sum\star\,\mathbf{tails}\ s_1} + \boxed{\uparrow/\sum\star\,\mathbf{inits}\ s_2}) \\
 & = & \mathbf{MSS}\ s_1 \uparrow \mathbf{MSS}\ s_2 \uparrow (\mathbf{MST}\ s_1 + \mathbf{MSI}\ s_2)
\end{array}$$

We have just introduced two unary functions:

$$\begin{array}{rcl}
\mathbf{MST}\ x & = & \uparrow/\sum\star\,\mathbf{tails}\ x \\
\mathbf{MSI}\ x & = & \uparrow/\sum\star\,\mathbf{inits}\ x
\end{array}$$

We now try to derive the same sort of recurrence for these two functions as for **MSS** . If we are successful, we will be able to tuple the functions together to form an efficient divide-and-conquer algorithm. We show only the derivation of **MST** below; **MSI** works similarly. The base case **MST** $[x]$ reduces to x, using reasoning similar to that for the other base case.

$$\begin{array}{lcl}
\mathbf{MST}\ s_1 +\!\!+ s_2 & = & \uparrow/\sum\star\boxed{\mathbf{tails}\ (s_1 +\!\!+ s_2)} \\
 & = & \uparrow/\sum\star\,(\boxed{(\mathbf{tails}\ s_1) \times_{+\!\!+} [s_2]} +\!\!+ \boxed{\mathbf{tails}\ s_2}) \\
 & = & \uparrow/(\boxed{\sum\star\,((\mathbf{tails}\ s_1) \times_{+\!\!+} [s_2])} +\!\!+ \boxed{\sum\star\,\mathbf{tails}\ s_2}) \\
 & = & \uparrow/((\boxed{\sum\star\,\mathbf{tails}\ s_1} \times_{+} \boxed{\sum\star\,[s_2]}) +\!\!+ \boxed{\sum\star\,\mathbf{tails}\ s_2})
\end{array}$$

$$
\begin{aligned}
&= (\boxed{\uparrow/\sum\star\,\textbf{tails}\,s_1} + \boxed{\uparrow/\sum\star\,[s_2]}) \uparrow \boxed{\uparrow/\sum\star\,\textbf{tails}\,s_2} \\
&= \boxed{(\textbf{MST}\,s_1) + \sum s_2} \uparrow \textbf{MST}\,s_2
\end{aligned}
$$

Of course $\sum$ obeys the same sort of recurrence, namely $\sum(s_1 +\!\!+ s_2) = (\sum s_1) + (\sum s_2)$. Hence, the tupling technique can be used to form a divide-and-conquer algorithm from the recurrences for **MST** , $\sum$, **MSI** , and **MSS** .

4 Specification of the Maximum Sum Subrectangle Problem

A generalization extends the segment problem to two dimensions. In this case we are given a rectangular array of integers, and are asked to find the sum of the maximum sum subrectangle. We model the array as a function A that applies to coordinate pairs $\langle i, j\rangle$, for $i \in xc$ and $j \in yc$, where xc and yc are the indices for the x-coordinate and y-coordinate, respectively; e.g. $xc = [1\,..\,m_x]$, $yc = [1\,..\,m_y]$. If x and y are subsegments of xc and yc respectively, then $x \times_{\lozenge} y$ is the bag of coordinates describing the subrectangle specified by x and y, and $A \star (x \times_{\lozenge} y)$ is the bag of A's values in that subrectangle. The following definition thus specifies the maximum sum subrectangle problem:

$$\textbf{MSSR}\,A = \uparrow/\sum\star\,(A\star)\star(\times_{\lozenge})\star((\textbf{segs}\,xc)\times_{\lozenge}(\textbf{segs}\,yc))$$

If we now attempt to split xc into xc_1 and xc_2 and yc into yc_1 and yc_2, the distributive property of **segs** and $\times_{\lozenge}$ will produce the following 16 cross-products:

$$
\begin{array}{ll}
\textbf{segs}\,xc_1 \times_{\lozenge} \textbf{segs}\,yc_1 & \textbf{segs}\,xc_2 \times_{\lozenge} \textbf{segs}\,yc_1 \\
\textbf{segs}\,xc_1 \times_{\lozenge} \textbf{segs}\,yc_2 & \textbf{segs}\,xc_2 \times_{\lozenge} \textbf{segs}\,yc_2 \\
\textbf{segs}\,xc_1 \times_{\lozenge} \textbf{tails}\,yc_1 & \textbf{segs}\,xc_2 \times_{\lozenge} \textbf{tails}\,yc_1 \\
\textbf{segs}\,xc_1 \times_{\lozenge} \textbf{inits}\,yc_2 & \textbf{segs}\,xc_2 \times_{\lozenge} \textbf{inits}\,yc_2 \\
 & \\
\textbf{tails}\,xc_1 \times_{\lozenge} \textbf{segs}\,yc_1 & \textbf{inits}\,xc_2 \times_{\lozenge} \textbf{segs}\,yc_1 \\
\textbf{tails}\,xc_1 \times_{\lozenge} \textbf{segs}\,yc_2 & \textbf{inits}\,xc_2 \times_{\lozenge} \textbf{segs}\,yc_2 \\
\textbf{tails}\,xc_1 \times_{\lozenge} \textbf{tails}\,yc_1 & \textbf{inits}\,xc_2 \times_{\lozenge} \textbf{tails}\,yc_1 \\
\textbf{tails}\,xc_1 \times_{\lozenge} \textbf{inits}\,yc_2 & \textbf{inits}\,xc_2 \times_{\lozenge} \textbf{inits}\,yc_2
\end{array}
$$

At this point, the need for computer assistance in carrying out derivations becomes apparent.

To be continued.

5 Future Work

1. Study related work in APL, FP, tupling.
2. **segs** is a homomorphism; what is it's form as $\oplus/f\star$? Is it useful? When is that normal form of a homomorphism useful?
3. Is there a sense of completeness for an equational axiomatization for this theory? A directed rewriting system?
4. Carry out the subrectangle derivation. Also for the subsegment case using indices.

5. Compare with notations that use bound variables (e.g. PERFORMO, REFINE). What programs are expressed more easily in REFINE? What would the derivations look like? Try returning not just the sum, but a subsegment or rectangle. Does the program get too cluttered?
6. Longest upsequence?
7. Try synthesizing split assuming join, instead of join assuming split, ala Doug.
8. Other derivations: minout, reduction over graphs, the periodic root.

6 Conclusions

The SQUIGOL notation is concise, inspired by algebra, APL and FP, and very amenable to hand manipulation. A SQUIGOL program can be rather dense, but spreading it out over more characters would not make it any clearer.

Bird has developed an interesting theory of sequences. Sequences are ubiquitous in computing, so computer scientists should develop a theory that helps solve problems that involve sequences, just as mathematicians have developed a theory of numbers that helps to solve problems that involve numbers.

The derivation shows the power of defining appropriate domain-specific operators, such as **segs**, and studying their properties.

Acknowledgements: I'm very grateful to Richard Bird for advising me during my brief stay at Oxford, and to Douglas Smith. I have also received interesting comments from Richard Jüllig and Allen Goldberg.

References

[1] Richard S. Bird. *Introduction to the Theory of Lists.* Technical Report PRG-56, Oxford University Computing Laboratory, Programming Research Group, October 1986.

[2] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[3] L. G. L. T. Meertens. *An Abstracto Reader prepared for IFIP WG 2.1.* Technical Report CWI Note CS-N8702, Centrum voor Wiskunde en Informatica, April 1987.

[4] Douglas R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, 1987. Technical Report KES.U.85.2, Kestrel Institute, Palo Alto, CA, 1985.

7 Algebraic Formulation of Some of the Laws

A *semigroup* is a pair $\langle S, \oplus \rangle$ of a *carrier* set S and an associative operation $\oplus$ over that set. A *monoid* is a triple $\langle M, \oplus, e_\oplus \rangle$, where $\langle M, \oplus \rangle$ is a semigroup and $e_\oplus$ is a left- and right-identity for $\oplus$. A *commutative monoid* is a monoid where the operation $\oplus$ is commutative. We will use the shorthand $M_\oplus$ to refer to the monoid with carrier set M, where M is one of the types described in Section 2.2, and $\oplus$ is an associative operation on M. The identity element will be left tacit. Bird gives ways in which the carrier set M and the operation $\oplus$ can be enlarged if an identity isn't present already.

If $M_\oplus$ is a semigroup, monoid, or commutative monoid then so is $[M]_{\times_\oplus}$.

A *monoid homomorphism* is a function h, where $h\colon M \to N$, and where $M_\oplus$ and $N_\otimes$ are monoids, satisfying $h \circ \oplus = \otimes \circ h{\times}h$. In this case we write $h\colon M_\oplus \to N_\otimes$. It can be easily shown that $h\, e_\oplus$ must be $e_\otimes$. If $h\colon M_\oplus \to M_\oplus$, then h is a *monoid automorphism*.

If $f\colon M \to N$, then $f\star\colon [M]_{+\!\!+} \to [N]_{+\!\!+}$.

If $M_\oplus$ is a monoid, then $\oplus/\colon [M]_{+\!\!+} \to M_\oplus$.

If $h\colon M_\oplus \to N_\otimes$, then $h\star\colon [M]_{\times_\oplus} \to [N]_{\times_\otimes}$.

A *semiring* is a quintuple $\langle M, \oplus, \otimes, e_\oplus, e_\otimes\rangle$ where $\langle M, \oplus, e_\oplus\rangle$ is a commutative monoid, $\langle M, \otimes, e_\otimes\rangle$ is a monoid, and $\otimes$ distributes over $\oplus$; i.e. for any x, both $\otimes x$ and $x\otimes$ are $M_\oplus$ automorphisms. The shorthand $M_{\oplus\otimes}$ will abbreviate that semiring.

If $M_{\oplus\otimes}$ is a semiring, then $\oplus/ : [M]_{\times_\otimes} \to M_\otimes$.

Sketch of Proof:

Assume true for $\#s_1 < n$ and prove for $\#s_1 = n$ as follows.

$$\begin{array}{lcll}
\oplus/(s_1 \times_\otimes s_2) & = & \oplus/(([x] +\!\!+ s_1') \times_\otimes s_2) & \\
 & = & \oplus/(([x] \times_\otimes s_2) +\!\!+ (s_1' \times_\otimes s_2)) & \text{distribute left } \times_\otimes,\ \oplus \text{ is commutative} \\
 & = & (\oplus/([x] \times_\otimes s_2)) \oplus (\oplus/(s_1' \times_\otimes s_2)) & \text{use induction hypothesis} \\
 & = & ((\oplus/[x]) \otimes (\oplus/s_2)) \oplus ((\oplus/s_1') \otimes (\oplus/s_2)) & \text{distribute left } \otimes \\
 & = & ((\oplus/[x]) \oplus (\oplus/s_1')) \otimes (\oplus/s_2) & \\
 & = & (\oplus/([x] +\!\!+ s_1')) \otimes (\oplus/s_2) & \\
 & = & (\oplus/s_1) \otimes (\oplus/s_2) &
\end{array}$$