Maximum Segment Sum is Back

Deriving Algorithms for Two Segment Problems with Bounded Lengths

Shin-Cheng Mu

Institute of Information Science, Academia Sinica, Taiwan

scm@iis.sinica.edu.tw

Abstract

It may be surprising that variations of the maximum segment sum (MSS) problem, a textbook example for the *squiggolists*, are still active topics for algorithm designers. In this paper we examine the new developments from the view of relational program calculation. It turns out that, while the classical MSS problem is solved by the Greedy Theorem, by applying the Thinning Theorem, we get a linear-time algorithm for MSS with upper bound on length. To derive a linear-time algorithm for the *maximum segment density* problem, on the other hand, we purpose a variation of thinning based on an extended notion of monotonicity. The concepts of *left-negative* and *right-screw* segments emerge from the search for monotonicity conditions. The efficiency of the resulting algorithms crucially relies on exploiting properties of the set of partial solutions and design efficient data structures for them.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Languages

Keywords Program Derivation, Maximum Segment Sum, Maximum Segment Density

1. Introduction

Given a list of numbers, the maximum segments sum (MSS) problem is to compute the maximum sum of a consecutive subsequence. The problem can be traced to as early as 1984 in Dijkstra and Feijen's *Een methode van programmeren* [13], but was probably made famous by Bentley [2, Column 7], and became a pet topic of the program derivation community after being given a formal treatment by Gries [20]. It appears that every textbook or tutorial on program derivation has to eventually talk about the MSS problem, be it functional [16, 17] or procedural [22].

The "squiggolists" have been successful in tackling various problems related to list segments. The Bird-Meertens formalism [4] started as a program calculi for lists [3]. Many individual problems were solved in published papers or as exercises in journal columns [30, 21, 27, for instances]. After the extensive study of Zantema [31], it appeared, for a while, that nothing more could be said about segment problems, let alone the most fundamental MSS.

PEPM'08, January 7–8, 2008, San Francisco, California, USA.

It may thus come as a surprise that variations of MSS are still active research topics in the algorithm design community, with new results being discovered in a steady pace. For the MSS problem with upper bound U on segment length (MSS$_U$), the first $O(n)$ algorithm (independent from U) was presented by Lin et al. [24] in 2002. In the paper, they also discussed the *maximum segment density* (MSD$_L$) problem maximising the average of all viable segments, and gave an $O(n \log L)$ algorithm where L is the lower bound on segment length. Goldwasser et al. [18, 19] reduced the complexity for MSD$_L$ to linear-time in the same year. Subsequently in 2003, Chuang and Lu [9, 10] generalised the linear-time algorithm to cover the case when in the input list have non-uniform widths, while Fan et al. [14] improved the algorithm for MSS$_U$. Chen et al. published an algorithm to compute [7] maximum density of disjoint segments in 2005. The interest in segment problems is partly motivated by applications in bioinformatics [24, for a summary], but extensions to trees have also being studied, for example, by Lau et al. [23] in 2006.

Meanwhile, the program derivation community has also been rapidly developing. Various "morphisms" on datatypes have been proposed and studied. Bird and de Moor [5] systematically studied derivation of optimisation problems, basing their theories on a relational theory of datatypes [1]. Curtis analysed dynamic programming [11] and greedy algorithms [12] in a relational setting. Sasano and Hu et al. [29, 28] studied a useful class of *maximum marking* problems and proposed linear time algorithms for them.

Program derivation is initially motivated by the wish to discover algorithms whose correctness is made evident by the derivation. However, the derivation of an algorithm not only suggests a way to understand it, but also allows one to identify the properties and theorems that made the derivation work. This is essential for making programming a science: to observe and study individual phenomena, identify common patterns, and propose unifying theories.

Being aware of the new developments in MSS, one naturally wants to know whether the recent theories of derivation can be applied. This paper derives two algorithms, by relational program calculation, solving MSS$_U$ and MSD$_L$, after reviewing the classical MSS for completeness. All three derivations follow a similar pattern. The problem is first decomposed to a sub-problem about prefixes, specified by a fold returning the set of possible solutions. We then attempt to reduce the number of possible solutions by some monotonicity property. Finally, the specification is refined to a linear-time algorithm by exploiting the properties of the set of solutions. While the Greedy Theorem is used to downsize the solution set for MSS, the MSS$_U$ turns out to be a natural candidate for applying the Thinning Theorem, both theorems proposed by Bird and de Moor [5]. To deal with MSD$_L$, on the other hand, we propose a variation of thinning based on an extended notion of monotonicity. The key concepts of *left-negative* and *right-screw* segments proposed by Lin et al. [24] can both be seen as conse-

quences of the attempts to meet the monotonicity conditions for the Thinning Theorem.

After introducing some basic concepts in Section 2 and reviewing the derivation for MSS in Section 3, we deal with MSSU and MSDL respectively in Section 4 and Section 5.

2. Preliminaries

In this section we give a minimal review the basic concepts we need. The reader is referred to Bird, Gibbons and Mu [6] for a short tutorial to relational program calculation for optimisation problems, and to Bird and de Moor [5] for a thorough introduction.[1]

2.1 Sets, Functions, and Relations

A relation R from set A to set B, written $R :: A \to B$, is a subset of the set $\{(a,b) \mid a \in A \wedge b \in B\}$. When $(a,b) \in R$, we say R maps a to b and sometimes also write $b \leftarrow R\,a$. A function is a special case of relations such that $(a,b) \in R$ and $(a,b') \in R$ implies $b = b'$. In such case we can safely write $b = R\,a$. In this paper, free occurrences of identifiers f, g, and h refer to functions. A total function is a function such that every $a \in A$ is mapped to some b.

Given relations $R :: B \to C$ and $S :: A \to B$, their composition $R \cdot S$ is defined by:

$$(a,c) \in R \cdot S \equiv \exists b : (a,b) \in S \wedge (b,c) \in R.$$

Composition is monotonic with respect to set inclusion. The identity function $id\,a = a$ is the unit of composition. A partial function that is a subset of id is called a *coreflexive*, often used to filter inputs satisfying certain predicates.

The *power transpose* operator Λ converts a relation to a set-valued total function:

$$\Lambda R\,a \quad = \quad \{b \mid (a,b) \in R\}.$$

An equivalent definition is $f = \Lambda R \equiv \in \cdot f = R$, where $\in$ is the relation mapping a set to one of its elements. Take $f = \Lambda R$, we get $\in \cdot \Lambda R = R$.

The *existential image* functor E converts a relation $R :: A \to B$ to a function mapping sets of A to sets of B: [2]

$$\mathsf{E}R\,x \quad = \quad \{b \mid \exists a \in x : b \leftarrow R\,a\}.$$

An alternative definition is $\mathsf{E}R = \Lambda(R \cdot \in)$. They satisfy the absorption law:

$$\mathsf{E}R \cdot \Lambda S \quad = \quad \Lambda(R \cdot S). \tag{1}$$

Take $S = T \cdot \in$, the absorption law implies $\mathsf{E}R \cdot \mathsf{E}T = \mathsf{E}(R \cdot T)$. We also have $\mathsf{E}R = \mathsf{E}(\in \cdot \Lambda R) = \mathsf{E} \in \cdot \mathsf{E}(\Lambda R)$. The function $\mathsf{E}\in$ is usually called *union* because it joins a set of sets into a single set.

Let $fst\,(a,b) = a$ and $snd\,(a,b) = b$. The "split" operator and the product functor are defined respectively by:

$$\begin{array}{lcl}(b,c) \leftarrow \langle R,S\rangle\,a & \equiv & b \leftarrow R\,a \wedge c \leftarrow S\,a,\\ (R \times S) & = & \langle R \cdot fst, S \cdot snd\rangle,\end{array}$$

They satisfy the absorption law:

$$(R \times S) \cdot \langle T,U\rangle \quad = \quad \langle R \cdot T, S \cdot U\rangle. \tag{2}$$

Functional programmers may be more familiar with the special case when both arguments are functions: $\langle f,g\rangle\,a = (f\,a, g\,a)$, and $(f \times g)\,(a,b) = (f\,a, g\,b)$.

The relation *distr* maps (x,ys) to all (x,y) if $y \in ys$:

$$distr \quad = \quad (id \times \in).$$

Therefore, we have $\Lambda distr\,(x,ys) = \{(x,y) \mid y \in ys\}$. The following equality relates *distr*, the split, and Λ.

$$\Lambda\langle f,R\rangle \quad = \quad \Lambda distr \cdot \langle f, \Lambda R\rangle \tag{3}$$

2.2 Relational Fold on Lists

Functional programmers are familiar with fold on lists, defined by:[3]

$$\begin{array}{lcl}foldr\,f\,e\,[\,] & = & e,\\ foldr\,f\,e\,(a{:}x) & = & f\,(a, foldr\,f\,e\,x).\end{array}$$

In fact, a function satisfies the two equations above for $foldr\,f\,e$ iff it equals $foldr\,f\,e$. One of the consequences of the uniqueness of fold is the *fold-fusion* rule:[4]

$$h \cdot foldr\,f\,e = foldr\,g\,(h\,e) \quad \Leftarrow \quad h \cdot f = g \cdot (id \times h).$$

When the fusion premise does not hold for h, a common technique is to introduce another function k and try to show that the fusion premise holds for $\langle h,k\rangle$. This technique is referred to as *tupling* and used a number of times in this paper.

The sum and length of a list can both be computed by a fold:

$$\begin{array}{lcl}sum & = & foldr\,plus\,0, \quad \textbf{where}\ plus\,(a,b) = a+b,\\ len & = & foldr\,((1+) \cdot snd)\,0.\end{array}$$

The familiar function $map\,f$, applying f to every element of the given list, is also a fold:

$$map\,f \quad = \quad foldr\,(cons \cdot (f \times id))\,[\,],$$

where $cons\,(a,x) = a{:}x$. Let *setify* be the function collecting the elements in a list to a set, we have the following property:

$$\mathsf{E}f \cdot setify \quad \supseteq \quad setify \cdot map\,f. \tag{4}$$

The function *tails*, returning all the suffixes of a list in decreasing lengths, can also be defined as a fold:

$$tails \quad = \quad foldr\,f\,[[\,]], \quad \textbf{where}\ f\,(a,x{:}xs) = (a{:}x){:}x{:}xs.$$

The function $scanr\,f\,e$ applies $foldr\,f\,e$ to every suffix of a list:

$$scanr\,f\,e \quad = \quad map\,(foldr\,f\,e) \cdot tails. \tag{5}$$

An important property we will make use of later is that $scanr\,f\,e$ itself can be implemented as a fold:

$$\begin{array}{l}scanr\,f\,e \quad = \quad foldr\,step\,[e],\\ \quad \textbf{where}\ step\,(a,b{:}x) = f\,(a,b){:}b{:}x\ .\end{array}$$

The property can be verified using fold fusion.

Relational folds is defined in terms of functional folds:

$$foldr\,R\,e \quad = \quad \in \cdot foldr\,\Lambda(R \cdot distr)\,\{e\}.$$

The fold fusion rule for relational folds generalise to inclusion:

$$\begin{array}{l}R \cdot foldr\,S\,e \supseteq foldr\,T\,d \ \Leftarrow\\ \qquad R \cdot S \supseteq T \cdot (id \times R) \ \wedge\ d \leftarrow R\,e.\end{array}$$

For example, the relation *prefix*, mapping the input list to one of its prefixes, can be defined as a fold:

$$prefix \quad = \quad foldr\,(const\,[\,] \cup cons)\,[\,],$$

where $const\,k\,x = k$. Note that $\cup$ denotes the union of two relations (sets of pairs) $const\,[\,]$ and *cons*, resulting in a relation mapping (a,x) to either $[\,]$ or $a{:}x$.

Dual to *prefix*, the relation *suffix*, defined as the reflexive, transitive closure of the partial function $tail\,(a{:}x) = x$, maps a list to one of its suffixes. It cannot be defined as a fold. However, we do have the following property:

$$setify \cdot tails \quad = \quad \Lambda suffix. \tag{6}$$

[1] The book presents program derivation from a category theoretical perspective. For the purpose of this paper, we restrict ourselves to sets and relations. Some properties may not necessarily be valid for all categories.

[2] In this paper we may assume that the power functor P coincides with E.

[3] Deviating from most functional languages, we use uncurried version of f.

[4] Free variables are universally quantified.

2.3 Maximum

A *preorder* $\trianglelefteq$ is a relation of type $A \to A$ for some A that is reflexive and transitive. We use the infix notation $a \trianglelefteq b$ to denote $(a,b) \in \trianglelefteq$. A preorder $\trianglelefteq$ is called a connected preorder if for all $a, b \in A$, either $a \trianglelefteq b$ or $b \trianglelefteq a$. The relation $max_{\trianglelefteq}$ picks a maximum element from the input set:

$$a \leftarrow max_{\trianglelefteq}\, xs \quad \equiv \quad a \in xs \;\wedge\; (\forall b \in xs : b \trianglelefteq a).$$

The definition itself assumes nothing of $\trianglelefteq$. However, some theorems we will introduce later demands $\trianglelefteq$ to be a preorder. Furthermore, if $\trianglelefteq$ is not connected, $max_{\trianglelefteq}\, xs$ may be undefined for some xs. For the rest of the paper we assume that $\trianglelefteq$ is a connected preorder if it is used in $max_{\trianglelefteq}$.

The following rule allows us to move *max* around by inventing new orderings:

$$max_{\trianglelefteq} \cdot \mathsf{E}f = f \cdot max_{\trianglelefteq_f} \qquad \text{where } a \trianglelefteq_f b \equiv f\,a \trianglelefteq f\,b, \tag{7}$$

while the following rule allows us to distribute *max* into *union*:

$$max_{\trianglelefteq} \cdot union \quad \supseteq \quad max_{\trianglelefteq} \cdot \mathsf{E}max_{\trianglelefteq}. \tag{8}$$

A naïve way to implement a set is to use a list. Let $maxlist_{\trianglelefteq}$ be the counterpart of $max_{\trianglelefteq}$ on lists. We only demand that $maxlist_{\trianglelefteq}$ satisfies the property:

$$max_{\trianglelefteq} \cdot setify \quad \supseteq \quad maxlist_{\trianglelefteq}. \tag{9}$$

One of the possible implementations for *maxlist* is $foldr\,(\uparrow)\;-\infty$ where $a \uparrow b$ yields the larger one between a and b.

In this paper, we model segment problems in the form:

$$max \cdot \Lambda(foldr\,R\,e).$$

By the definition of relational fold, the expression is equivalent to

$$max \cdot foldr\,\Lambda(R \cdot distr)\,\{e\}.$$

The fold steps through the input list and maintains a set of all potential solutions, from which a best one is chosen by *max*. If certain properties hold, however, we may not need to keep all the potential solutions in each step. A relation $R :: (A \times B) \to B$ is *monotonic on a preorder* $\trianglelefteq$ if for all b and b' in the range of R:

$$c \leftarrow R\,(a,b) \;\wedge\; b \trianglelefteq b' \quad \Rightarrow \quad (\exists c' :\; c' \leftarrow R\,(a,b') \;\wedge\; c \trianglelefteq c').$$

Consider two potential solutions b and b' such that $b \trianglelefteq b'$. That R is monotonic on $\trianglelefteq$ means that for every c we can generate from $R\,(a,b)$, there must be some c' resulting from $R\,(a,b')$ that is no worse than c. Therefore, there is no point keeping the lessor solution c. Intuitively, if R is monotonic on $\trianglelefteq$, in each step of $foldr\,R\,e$ we only need to keep the best solution with respect to $\trianglelefteq$. This is justified by the following theorem [4] provable by fold fusion:

THEOREM 2.1 (Greedy Theorem (for Lists)). Let R be monotonic on preorder $\trianglelefteq$, then we have:

$$max_{\trianglelefteq} \cdot \Lambda(foldr\,R\,e) \quad \supseteq \quad foldr\,(max_{\trianglelefteq} \cdot \Lambda R)\,e.$$

Sometimes, it is too strong a demand for $\trianglelefteq$ to be a connected preorder. When the preorder is not connected, we instead appeal to a Thinning Theorem, to be introduced later. For now, let us look at an example using the Greedy Theorem.

3. Classical Maximum Segment Sum

To familiarise the readers with relational program calculation, let us review how the classical MSS is solved in a relational setting. A segment of a list is a prefix of a suffix. The classical MSS problem is thus specified by:

$$mss = max_{\leq} \cdot \Lambda(sum \cdot prefix \cdot suffix).$$

The input has type $[V]$ for some numerical type V, and $\leq$ is the "less than or equal to" order on V. From now on, when $\leq$ is the designated order, we omit the subscript.

The standard derivation starts with some book-keeping manipulation distributing *max* into the set-construction:

$$\begin{aligned}
& max \cdot \Lambda(sum \cdot prefix \cdot suffix) \\
= \quad & \{ \text{ by (1) and } \mathsf{E}R = \mathsf{E}{\in} \cdot \mathsf{E}\Lambda R \ \} \\
& max \cdot union \cdot \mathsf{E}\Lambda(sum \cdot prefix) \cdot \Lambda suffix \\
= \quad & \{ \text{ by (8) } \} \\
& max \cdot \mathsf{E}(max \cdot \Lambda(sum \cdot prefix)) \cdot \Lambda suffix.
\end{aligned}$$

Let us concentrate on $max \cdot \Lambda(sum \cdot prefix)$. The aim is to transform it into the form of $max \cdot \Lambda(foldr\,R\,e)$ so that we can apply the Greedy Theorem. Since $sum \cdot (const\,[\,] \cup cons) = (const\,0 \cup plus) \cdot (id \times sum)$, by fold fusion we can fuse *sum* into *prefix*:

$$sum \cdot prefix \quad = \quad foldr\,(const\,0 \cup plus)\,0.$$

Furthermore, one can see that $const\,0 \cup plus$ is monotonic on $(\leq)$ — the monotonicity condition expands to:

$$\begin{aligned}
& b \leq b' \wedge (c = 0 \vee c = a + b) \\
& \quad \Rightarrow (\exists c' \in \{0, a + b'\} : c \leq c').
\end{aligned}$$

When $const\,0$ is used to generate c, we can always choose $c' = 0$; when *plus* is used, we can choose $c' = a + b'$. Both solutions are no worse than c. Thus the Greedy Theorem applies: we can promote *max* into the fold:

$$max \cdot \Lambda(sum \cdot prefix) \quad \supseteq \quad foldr\,(max \cdot \Lambda(const\,0 \cup plus))\,0.$$

Let $plz = max \cdot \Lambda(const\,0 \cup plus)$. After some calculation we have that $plz\,(a,b) = 0 \uparrow (a + b)$.

We continue with the derivation:

$$\begin{aligned}
& mss \\
\supseteq \quad & \{ \text{ derivation so far } \} \\
& max \cdot \mathsf{E}(foldr\,plz\,0) \cdot \Lambda suffix \\
\supseteq \quad & \{ \text{ by (6), (4), and (9) } \} \\
& maxlist \cdot map\,(foldr\,plz\,0) \cdot tails \\
= \quad & \{ \text{ by (5) } \} \\
& maxlist \cdot scanr\,plz\,0.
\end{aligned}$$

The penultimate step above implements sets by lists, while the introduction of *scanr* in the last step turns a quadratic-time algorithm to linear-time. Given an input of length n, the algorithm still uses $O(n)$ space, which can be improved by:

$$\begin{aligned}
& maxlist \cdot scanr\,plz\,0 \\
= \quad & \{ \text{ tupling } \} \\
& snd \cdot \langle head, maxlist \rangle \cdot scanr\,plz\,0 \\
= \quad & \{ \text{ fold fusion, } step \text{ defined below } \} \\
& snd \cdot foldr\,step\,(0,0).
\end{aligned}$$

We have therefore derived a linear-time, constant space algorithm for MSS:

$$\begin{aligned}
& mss \quad \supseteq \quad snd \cdot foldr\,step\,(0,0), \\
& \quad \textbf{where } step :: (V,(V,V)) \to (V,V) \\
& \qquad step\,(a,(b,m)) = (c, m \uparrow c) \textbf{ where } c = 0 \uparrow (a+b).
\end{aligned}$$

It is easy to check that the right-hand side defines a total function, since all its components are total functions.

4. Maximum Segment Sum with Bounded Lengths

Maximum segment sum with an upper bound U on the length of the segment is specified by inserting a coreflexive *fitlen* to the specification:

$$mssu \quad = \quad max \cdot \Lambda(sum \cdot fitlen \cdot prefix \cdot suffix),$$

where $fitlen = \{(xs, xs) \mid len\, xs \leq U\}$. Can we also derive a linear-time algorithm from this specification? In fact, the approach for maximum marking problems developed by Sasano and Hu et al. [29, 28] suggests a $O(nU)$ algorithm, where n is the length of the input list. We are, however, aiming for an algorithm whose complexity is independent from U.

Like the previous section, we start with distributing *max* into E, resulting in:

$$max \cdot \mathsf{E}(max \cdot \Lambda(sum \cdot fitlen \cdot prefix)) \cdot \Lambda suffix.$$

The next step is to transform $max \cdot \Lambda(sum \cdot fitlen \cdot prefix)$ into the form of $max \cdot \Lambda(foldr\, R\, e)$. However, $sum \cdot fitlen \cdot prefix$ does not constitute a fold. We have to return both the sum and the length of a list. We derive:

$$\begin{array}{ll} & max \cdot \Lambda(sum \cdot fitlen \cdot prefix) \\ = & \quad \{ \text{ let } fit = \{((s,l),(s,l)) \mid l \leq U\} \} \\ & max \cdot \Lambda(fst \cdot fit \cdot \langle sum, len\rangle \cdot prefix) \\ = & \quad \{ \text{ fusing } fit \cdot \langle sum, len\rangle \text{ into } prefix, \text{ see below } \} \\ & max \cdot \Lambda(fst \cdot foldr\, (zero \cup fit \cdot add)\, (0,0)) \\ = & \quad \{ \text{ by (7), see below} \} \\ & fst \cdot max_{\leq_1} \cdot \Lambda(foldr\, (zero \cup fit \cdot add)\, (0,0)), \end{array}$$

where the partial function *fit* yields value only if the length is smaller than U, and the auxiliary definitions are given below:

$$\begin{array}{lcl} zero\, (a, xs) & = & (0,0), \\ add\, (a, (s,l)) & = & (a+s, 1+l), \\ (s_1,l_1) \leq_1 (s_2,l_2) & \equiv & s_1 \leq s_2. \end{array}$$

From now on, we denote the type of a pair of sum and length by *SL*. When a pair (s,l) denotes a sum and a length of a segment, we call it an SL-pair. When we say so we imply that $l \geq 0$.

Now that we have an expression in the form $max \cdot \Lambda(foldr\, R\, e)$, we wish to apply the Greedy Theorem. Unfortunately, $zero \cup fit \cdot add$ is not monotonic on $\leq_1$. Consider two SL-pairs (s_1,l_1) and (s_2,l_2), where $s_1 \leq s_2$. The monotonicity condition expands to:

$$\begin{array}{l} s_1 \leq s_2 \wedge (s_3 = 0 \vee s_3 = a + s_1) \\ \quad \Rightarrow\ (\exists (s_4,l_4) \in \{(0,0),(a+s_2,1+l_2)\} : \\ \qquad s_3 \leq s_4 \wedge l_4 \leq U). \end{array}$$

Assume s_3 is assigned $a + s_1$ and that it is positive. It could be the case that $1 + l_2$ is too long, leaving the only choice for (s_4,l_4) to be $(0,0)$, therefore $s_3 \leq s_4$ does not hold. It shows that we cannot drop (s_1,l_1) in favour of (s_2,l_2) immediately, even if $s_1 < s_2$.

We can, however, conclude that it is definitely not worth to keep (s_1,l_1) if this prefix is neither bigger nor shorter than (s_2,l_2). Define:

$$(s_1,l_1) \preceq (s_2,l_2) \quad \equiv \quad s_1 \leq s_2 \wedge l_1 \geq l_2.$$

The relation $zero \cup fit \cdot add$ is indeed monotonic on $(\preceq)$. However, $(\preceq)$ is not a connected preorder. The Greedy Theroem is true but useless because $max_{\preceq}$ may simply yield nothing for most inputs.

4.1 The Thinning Theorem

Given preorder $\preceq$, let $thin_{\preceq}$ be a relation between sets of solutions so that:

$$ys \leftarrow thin_{\preceq}\, xs \ \equiv\ ys \subseteq xs \wedge (\forall b \in xs : (\exists b' \in ys : b \preceq b')).$$

While *max* relates a set *xs* to an element, *thin* relates it to a subset *ys* obtained by removing some elements, with one condition: for every b in *xs*, there must be a solution b' remaining in *ys* that is at least as good as b, therefore nothing is lost.

Given a connected preorder $\trianglelefteq$ and a preorder $\preceq$ that is a sub-ordering of $\trianglelefteq$ (that is, $x \preceq y \Rightarrow x \trianglelefteq y$), we have:

$$max_{\trianglelefteq} \quad \supseteq \quad max_{\trianglelefteq} \cdot thin_{\preceq}. \tag{10}$$

The Thinning Theorem [4] below can also be proved by fold fusion:

THEOREM 4.1 (Thinning Theorem (for Lists)). Let R be monotonic on preorder $\preceq$, then we have:

$$thin_{\preceq} \cdot \Lambda(foldr\, R\, e) \quad \supseteq \quad foldr\, (thin_{\preceq} \cdot \Lambda(R \cdot distr))\, \{e\}.$$

The Thinning Theorem is very general. It is not specified how thoroughly the set of solutions should be "thinned". Depending on how the $thin_{\preceq}$ inside the *foldr* is further refined, the theorem covers the entire spectrum from a greedy algorithm (if $\preceq$ happens to be connected) to generating all potential solutions by brute force (since *id* also satisfies the specification for $thin_{\preceq}$). Nor does the theorem specify how the thinning is performed. The efficiency of the algorithm depends critically on whether we can design a data structure representing the set of solutions such that thinning can be efficiently carried out.

4.2 Thinning the Set of Prefixes

Back to MSSU. Since $\preceq$ is a sub-ordering of $\leq_1$, we reason

$$\begin{array}{ll} & max_{\leq_1} \cdot \Lambda(foldr\, (zero \cup fit \cdot add)\, (0,0)) \\ \supseteq & \quad \{ \text{ by (10) } \} \\ & max_{\leq_1} \cdot thin_{\preceq} \cdot \Lambda(foldr\, (zero \cup fit \cdot add)\, (0,0)) \\ \supseteq & \quad \{ \text{ Thinning Theorem } \} \\ & max_{\leq_1} \cdot foldr\, (thin_{\preceq} \cdot \Lambda((zero \cup fit \cdot add) \cdot distr)) \\ & \qquad \{(0,0)\}. \end{array}$$

Let us further refine $thin_{\preceq} \cdot \Lambda((zero \cup fit \cdot add) \cdot distr)$ to a function by picking a refinement of $thin_{\preceq}$. The function *zero* produces a new element $(0,0)$. Since all lengths are non-negative, $(0,0)$ is not worse than any elements returned by $fit \cdot add$ and will not be removed by $thin_{\preceq}$. With the presence of $(0,0)$, all elements returned by $fit \cdot add$ that has a non-positive sum can safely be removed. Therefore, one of the possibilities is to refine $thin_{\preceq} \cdot \Lambda((zero \cup fit \cdot add) \cdot distr)$ to:

$$\begin{array}{lcl} tadd & :: & (V, \{SL\}) \to \{SL\}, \\ tadd\, (a, xs) & = & \{(0,0)\} \cup \Lambda(pos \cdot fit \cdot add \cdot distr)\, (a, xs), \\ & = & \{(0,0)\} \cup \mathsf{E}(pos \cdot fit \cdot add_a)\, xs, \end{array}$$

where $pos = \{((s,l),(s,l)) \mid s > 0\}$, and the function add_a appends a to an SL-pair:

$$add_a\, (s,l) \quad = \quad (a+s, 1+l).$$

The specification for *mssu* has been refined to:

$$max \cdot \mathsf{E}(fst \cdot max_{\leq_1} \cdot foldr\, tadd\, \{(0,0)\}) \cdot \Lambda suffix.$$

The next few steps of the derivation follows a pattern similar to that of Section 3. We implement the outer set by a list, yielding:

$$maxlist \cdot map\, (fst \cdot max_{\leq_1}) \cdot map\, (foldr\, tadd\, \{(0,0)\}) \cdot tails.$$

Expression $map\, (foldr\, tadd\, \{(0,0)\}) \cdot tails$ can be implemented by *scanr*, which is then fused with $\langle head, maxlist \cdot map\, (fst \cdot max_{\leq_1})\rangle$ after a tupling, yielding:

$$\begin{array}{l} mssu \ \supseteq\ snd \cdot foldr\, step\, (\{(0,0)\}, 0), \\ \\ step\ ::\ (V, (\{SL\}, V)) \to (\{SL\}, V), \\ step\, (a, (xs, m)) \ =\ (xs', fst\, (max_{\leq_1}\, xs') \uparrow m) \\ \quad \textbf{where}\ xs' = tadd\, xs \end{array}$$

-2	4	-3	2	2	-1	4	-2	1	-1		
$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	
~~$(-2,1)$~~	$(4,1)$	~~$(-3,1)$~~	$(2,1)$	$(2,1)$	~~$(-1,1)$~~	$(4,1)$	~~$(-2,1)$~~	$(1,1)$	~~$(-1,1)$~~		*xs*
$(2,2)$	~~$(5,4)$~~	~~$(-1,2)$~~	$(4,2)$	$(5,3)$	$(3,2)$		~~$(-1,2)$~~				
		$(1,3)$	~~$(7,4)$~~								
5	5	5	5	5	4	4	1	1	0	0	*m*

Figure 1. Computing *mssu* $[-2,4,-3,2,2,-1,4,-2,1,-1]$ with $U=3$.

Figure 1 shows how *mssu* $[-2,4,-3,2,2,-1,4,-2,1,-1]$ is computed when $U=3$. The input list, processed by *foldr* from right to left, is written as the first row. The second and the third rows show how *xs* and *m* evolve in each step of the fold, starting from $xs=\{(0,0)\}$ and $m=0$. SL-pairs struck out by a horizontal line are those removed for having a non-positive sum, while those struck out by a wavy line are those removed for being too long. The output of the algorithm is 5, the bottom-left value of *m*.

We have not yet talked about how the set *xs* is represented. The reader may have noticed possible inefficiency if we naïvely implement the set *xs* as a list of SL-pairs. In the next subsection, we will exploit some crucial properties of the set to produce an efficient implementation.

Remark Lin et al. [24] defined a *left-negative* segment as a segment whose sum is positive, while the sums of all its proper prefixes are negative. The prefixes preserved under the ordering $\preceq$ are exactly those composed of only its left-negative segments. We have thus shown that the concept of left-negative segments naturally emerges from looking for an ordering on which $(zero \cup fit \cdot add)$, the relation building the solutions, is monotonic. □

4.3 An Efficient Representation of Sets of Prefix Sums

The reader might have noticed, from Figure 1, that if we sort the SL-pairs in increasing lengths, the sums come in increasing order, too. It is a consequence of the ordering $\preceq$ — a longer SL-pair that does not have a larger sum need not be kept. This is a hint that we can implement the set of SL-pairs as a double-ended queue [8]. Newer and shorter elements are added to the front end, keeping the SL-pairs sorted by length. Removal of SL-pairs longer than *U* always starts from the back end. The empty prefix $(0,0)$, always present in the set, need not be explicitly represented. Since the deque is also sorted by increasing sum, the check $a+s>0$ can be performed only on the front segment of the queue, while $max_{\preceq_1}$ can be implemented by returning the first SL-pair from the end of the deque. Therefore, *step* can be refined to:

$$
\begin{array}{l}
step :: (V,(Deque\,SL,V)) \to (Deque\,SL,V),\\
step\,(a,(xs,m)) = (xs',lst\,xs' \uparrow m),\\
\quad \textbf{where}\ xs' = (dropl\,neg \cdot dropr\,long)\,((a,1){:}map\,add_a\,xs),\\
\qquad neg\,(s,l) = s<0,\\
\qquad long\,(s,l) = l>U,\\
\qquad lst\,[\,] = 0,\\
\qquad lst\,(ys +\!\!+ [(s,l)]) = s.
\end{array}
$$

We denote deques by a notation similar to lists: appending element *a* to the front of deque *x* is denoted by *a*:*x*, while appending to the end is denoted by $x +\!\!+ [a]$. The function *dropl p* removes elements from the front of the queue as long as the predicate *p* holds, while *dropr p* removes elements from the back:

$$
\begin{array}{lcll}
dropl\,p\,[\,] & = & [\,], & \\
dropl\,p\,(a{:}xs) & = & a{:}xs, & \text{if } \neg(p\,a),\\
 & = & dropl\,p\,xs, & \text{otherwise,}
\end{array}
$$

and *dropr* is defined similarly, replacing *a*:*xs* by $xs +\!\!+ [a]$. The function *dropr long* removes those SL-pairs that are longer than *U*, while *dropl neg* removes those that have a negative sum.

With the deque implementation, all operations in *step*, except $map\,add_a$, run in amortised linear time. To see that, consider the size of *xs*. It increases at most *n* times with each $((a,1){:})$ operation, while each recursive call of *dropl* or *dropr* decreases it by one.

The last performance bottleneck is $map\,add_a$. Luckily, we can avoid having to perform add_a to every element in *xs* by a change of presentation. We define:

$$
\begin{array}{l}
diffs\,((c,n),xs) = ((c,n),map\,(dif\,(c,n))\,xs),\\
\quad \textbf{where}\ dif\,(c,n)\,(s,l) = (c-s,n-l),
\end{array}
$$

The function *diffs* replaces every SL-pair in *xs* by its *difference* from the reference point (c,n). The advantage is that, wherever we need to increase each SL-pair in *xs*, now we merely need to increase the reference point (c,n). For example, consider adding an element 2 to the following *xs*:

$$
\begin{array}{lcl}
xs & = & [(2,1),(5,3)],\\
xs' & = & (2,1){:}\,map\,add_2\,xs = [(2,1),(4,2),(7,4)].
\end{array}
$$

It takes a linear time operation to compute xs' from *xs*. Pick a reference point, for example $(3,6)$, we have:

$$
\begin{array}{lcl}
ys & = & diffs\,((3,6),xs) = ((3,6),[(1,5),(-2,3)]),\\
ys' & = & diffs\,(add_2\,(3,6),xs'),\\
 & = & ((5,7),[(3,6),(1,5),(-2,3)]).
\end{array}
$$

We can compute ys' from *ys* in constant time. To get *xs* and xs' back, we observe that $diffs \cdot diffs = id$.

By introducing *diffs* and fuse it into the fold, we come up with the final algorithm:

$$
mssu = snd \cdot foldr\,stepd\,(((0,0),[\,]),0),
$$

where *stepd* is defined by:

$$
\begin{array}{l}
stepd :: (V,((SL,Deque\,SL),V)) \to ((SL,Deque\,SL),V),\\
stepd\,(a,(((c,n),ys),m)) = ((add_a\,(c,n),ys),lst'\,ys' \uparrow m),\\
\quad \textbf{where}\ ys' = (dropl\,neg' \cdot dropr\,long')\,((c,n){:}ys),\\
\qquad neg'\,(s,l) = a+c-s<0,\\
\qquad long'\,(s,l) = 1+n-l>U,\\
\qquad lst'\,[\,] = 0,\\
\qquad lst'\,(zs +\!\!+ [(s,l)]) = a+c-s.
\end{array}
$$

This is an $O(n)$ time, $O(U)$ space algorithm.

The seemingly cryptic definitions of neg', $long'$, etc., are results of fusing *diffs* into the fold. The derivation is tedious but routine, and we will merely sketch an outline. Let $assocl\,(a,(b,c)) = ((a,b),c)$, we derive:

$$
\begin{array}{ll}
 & snd \cdot foldr\,step\,([\,],0)\\
= & \quad \{\ \text{tupling, abbreviating } \langle sum,len\rangle \text{ to } sumlen\ \}\\
 & snd \cdot snd \cdot \langle sumlen, foldr\,step\,([\,],0)\rangle\\
= & \quad \{\ \text{some pair manipulation}\ \}\\
 & snd \cdot (diffs \times id) \cdot assocl \cdot \langle sumlen, foldr\,step\,([\,],0)\rangle.
\end{array}
$$

One can show that $\langle sumlen, foldr\ step\ ([\,],0)\rangle$ is a fold by the uniqueness of fold:

$$\langle sumlen, foldr\ step\ ([\,],0)\rangle\ =\ foldr\ stepl\ ((0,0),([\,],0)),$$

$$\begin{array}{l} stepl :: (V,(SL,(Deque\ SL,V))) \to (SL,(Deque\ SL,V)),\\ stepl\ (a,((c,n),(ys,m)))\ =\ (add_a\ (c,n),(ys',lst'\ ys' \uparrow m)),\\ \quad \textbf{where}\ ys'\ =\ (dropl\ neg \cdot dropr\ long)\\ \qquad\qquad\qquad ((a,1) {:}\ map\ add_a\ ys). \end{array}$$

Now we try to fuse $(diffs \times id) \cdot assocl$ into $foldr\ stepl\ ((0,0),([\,],0))$. That is, we have to construct a *stepd* such that:

$$\begin{array}{l} (diffs \times id) \cdot assocl \cdot stepl\ =\\ \quad stepd \cdot (id \times ((diffs \times id) \cdot assocl)). \end{array}$$

We will make use of the following properties, which can be proved from the definitions of *diffs* and add_a,

$$\begin{array}{rcll} diffs \cdot (add_a \times map\ add_a) & = & (add_a \times id) \cdot diffs, & (11)\\ dropl\ p \cdot map\ f & = & map\ f \cdot dropl\ (p \cdot f). & (12) \end{array}$$

Property (11) allows us to lift add_a out of the set construction, avoiding altering each prefix. Property (12), valid only for total f and p, is used to construct new predicates for *dropl* and *dropr*:

$$\begin{array}{rcl} neg'\ (s,l) & = & (neg \cdot add_a \cdot dif\ (c,n))\ (s,l),\\ long'\ (s,l) & = & (long \cdot add_a \cdot dif\ (c,n))\ (s,l), \end{array}$$

which simplifies to the definitions given earlier.

5. Maximum Segment Density

With so much discussion about sums and lengths, it is natural to consider a related problem: to maximise the *density* (the sum divided by the length) of a segment. Without a constraint on the minimal length of the segment, the problem reduces to searching for the largest element of the input, since averaging it with its neighbours would not make it any larger. Therefore we consider the maximum density among segments no shorter than some positive number L, and assume that the input is no shorter, either.

For brevity, we introduce several auxiliary definitions:

$$\begin{array}{lcl} div\ (s,l) & = & s/l, \qquad \text{for } l > 0,\\ avg\ x & = & div\ (sum\ x, len\ l),\\ dav\ (x,(s,l)) & = & div\ (sum\ x + s, len\ x + l),\\ (s_1,l_1) \leq_{\mathsf{d}} (s_2,l_2) & \equiv & div\ (s_1,l_1) \leq div\ (s_2,l_2),\\ (s_1,l_1).(s_2,l_2) & = & (s_1+s_2, l_1+l_2). \end{array}$$

Note that $add_a\ (s,l) = (a,1).(s,l)$. To model MSDL, one may follow the "prefix-of-suffix" approach of the previous sections:

$$\begin{array}{rcl} msdl & = & max \cdot \Lambda(avg \cdot prefix_L \cdot suffix),\\ & = & max \cdot \mathsf{E}(max \cdot \Lambda(avg \cdot prefix_L)) \cdot \Lambda suffix, \end{array}$$

where $prefix_L$ maps a list to one of its prefixes no shorter than L.

Like before, we will concentrate on $max \cdot \Lambda(avg \cdot prefix_L)$. The relation $prefix_L$ can be defined in a number of ways. For example, let $split_L$ be the function that, given input x, returns a pair of the first $min\ \{L, len\ x\}$ elements and the rest. We may define $prefix_L$ by:

$$prefix_L\ =\ cat \cdot (id \times prefix) \cdot split_L,$$

where $cat\ (x,y) = x +\!\!+ y$. We reason:

$$\begin{array}{ll} & max \cdot \Lambda(avg \cdot cat \cdot (id \times prefix) \cdot split_L)\\ = & \quad \{\ \text{since } split_L \text{ is a function}\ \}\\ & max \cdot \Lambda(avg \cdot cat \cdot (id \times prefix)) \cdot split_L\\ = & \quad \{\ \text{by the definitions}\ \}\\ & max \cdot \Lambda(dav \cdot (id \times sumlen \cdot prefix)) \cdot split_L\\ = & \quad \{\ \text{by (1)}\ \}\\ & max \cdot \mathsf{E}dav \cdot \Lambda(id \times sumlen \cdot prefix) \cdot split_L\\ = & \quad \{\ \text{by (3)}\ \}\\ & max \cdot \mathsf{E}dav \cdot \Lambda distr \cdot (id \times \Lambda(sumlen \cdot prefix)) \cdot split_L \end{array}$$

Given an prefix longer than L, $(id \times \Lambda(sumlen \cdot prefix)) \cdot split_L$ splits the prefix into two parts: an L-sized *header* and a set of *appendants*. In the next section, we will try to perform some thinning on the set of SL-pairs of appendants.

5.1 Properties of Appendants

Given a set of appendants, represented as SL-pairs, we aim to remove some "useless" candidates. But how do we know that an appendent is useless? The following key observations were made by Lin et al. [24]:

LEMMA 5.1. Let x,y be SL-pairs such that $x <_{\mathsf{d}} y$. Then $x <_{\mathsf{d}} x.y <_{\mathsf{d}} y$.

Proof. Let $x = (s_1,l_1)$ and $y = (s_2,l_2)$. For $x <_{\mathsf{d}} x.y$:

$$\begin{array}{ll} & div\ (s_1+s_2, l_1+l_2)\\ = & div\ x \times l_1/(l_1+l_2) + div\ y \times l_2/(l_1+l_2)\\ > & \quad \{\ \text{since } x <_{\mathsf{d}} y\ \}\\ & div\ x \times l_1/(l_1+l_2) + div\ x \times l_2/(l_1+l_2)\\ = & div\ x \end{array}$$

Other cases follow similarly.

□

COROLLARY 5.2. Let z,x,y be SL-pairs and $x \leq_{\mathsf{d}} y$, then at least one of $z.x \leq_{\mathsf{d}} z$ and $z.x <_{\mathsf{d}} z.x.y$ *is true.*

Proof. Either $x \leq_{\mathsf{d}} z$ or $x >_{\mathsf{d}} z$. If $x \leq_{\mathsf{d}} z$, we reason:

$$\begin{array}{ll} & x \leq_{\mathsf{d}} z\\ \Rightarrow & \quad \{\ \text{Lemma 5.1 and } x =_{\mathsf{d}} z \Rightarrow x =_{\mathsf{d}} z.x =_{\mathsf{d}} z\ \}\\ & x \leq x.z \leq z\\ \Rightarrow & \quad \{\ \text{since } x.z = z.x\ \}\\ & z.x \leq_{\mathsf{d}} z \end{array}$$

If $z <_{\mathsf{d}} x$, we reason:

$$\begin{array}{ll} & z <_{\mathsf{d}} x \leq_{\mathsf{d}} y\\ \Rightarrow & \quad \{\ \text{Lemma 5.1}\ \}\\ & z <_{\mathsf{d}} z.x <_{\mathsf{d}} x \leq_{\mathsf{d}} y\\ \Rightarrow & \quad \{\ \text{Lemma 5.1}\ \}\\ & z.x <_{\mathsf{d}} z.x.y \end{array}$$

□

Consider a set of solutions containing $(0,0)$, x, and $x.y$ with $x \leq_{\mathsf{d}} y$. By Corollary 5.2, we can safely remove x because for any header z, at least one of z $(= z.(0,0))$ and $z.x.y$ constitutes a solution no worse than $z.x$.

However, this intuition is difficult to be modelled as an ordering between *individual* appendants because neither z nor $z.x.y$ alone is always preferred over $z.x$. Consider $x = (5,1)$ and $y = (10,1)$, and header $z = (1,1)$. Among them, $z.x.y$ has the largest average. If we extend the header to $z' = (25,1).z = (26,2)$, however, $z'.x.y$ is no longer better than $z'.x$. If we further extend z' with a small value, the appendant $x.y$ may be preferred again. To model such situation, we need a notion of monotonicity on a set of solutions.

5.2 The Pruning Theorem

We call a relation $\ll$ between B and powersets of B an *absorptive* relation if it is:

- reflexive in the sense that $x \ll xs$ if $x \in xs$, and
- transitive in the sense that: $x \ll ys$ and $(\forall y \in ys : y \ll zs)$ implies $x \ll zs$.

d	e	f	3	10	-1	4	6	
e	f	3	10	-1	4	6	-5	x
f	3	10	-1	4	6	-5	11	
3	10	-1	4	6	-5	11		b
$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	
~~$(3,1)_3$~~	$(10,1)_{10}$	~~$(-1,1)_{-1}$~~	~~$(4,1)_4$~~	$(6,1)_6$	~~$(-5,1)_{-5}$~~	$(11,1)_{11}$		ys
$(13,2)_{6\cdot5}$	$(25,6)_{4\cdot16}$	~~$(9,3)_3$~~	$(10,2)_5$	$(12,3)_4$	$(6,2)_3$			
$(28,7)_4$		$(15,5)_3$	$(16,4)_4$					

Figure 2. Computing $msdl\,[d,e,f,3,10,-1,4,6,-5,11]$ with $L=3$.

The name comes from the idea that x is absorbed by the set xs if the latter contains an element as good as x. A relation $R :: (A\times B)\to B$ is said to be monotonic on absorptive relation $\ll$ if for all $b :: B$ and $xs :: \{B\}$ included in the range of R:

$$c \leftarrow R\,(a,b)\ \wedge\ b \ll xs\ \Rightarrow\ c \ll \Lambda(R\cdot distr)\,xs.$$

Intuitively, given a set xs in which there are some elements as good as b, monotonicity demands that if we apply R to all elements in xs, there always exists some elements in the resulting set that is as good as c. For a absorptive relation $\ll$, we define a relation $prune_{\ll} :: \{B\}\to\{B\}$:

$$ys \leftarrow prune_{\ll}\,xs \quad\equiv\quad ys\subseteq xs\ \wedge\ (\forall b\in xs : b\ll ys).$$

Given a preorder $\preceq$, one can always construct a absorptive relation $x \ll ys \equiv (\exists y\in ys : x\preceq y)$. In this case, $prune_{\ll}$ reduces to $thin_{\preceq}$. Not every absorptive relation can be expressed in terms of a preorder, however.

We propose the following Pruning Theorem, whose proof is not too different from that of the Thinning Theorem.

THEOREM 5.3 (Pruning Theorem). Let R be monotonic on absorptive relation $\ll$. We have:

$$prune_{\ll}\cdot\Lambda(foldr\,R\,e)\ \supseteq\ foldr\,(prune_{\ll}\cdot\Lambda(R\cdot distr))\,\{e\}.$$

5.3 Pruning the Appendants

Back to MSDL. Consider the last expression in the derivation before entering Section 5.1 and abbreviate $max\cdot \mathsf{E}dav\cdot\Lambda distr$, of type $([V]\times\{SL\})\to V$, to $maxap$. Given a header x and a set of SL-pairs ys, $maxap\,(x,ys)$ picks an appendant y from ys such that $div\,(x.y)$, the returned value, is maximised. If we define:

$$x\ll ys\quad\equiv\quad(\forall z : \exists y\in ys : z.x\le_{\mathsf{d}} z.y),$$

it is clear from its definition that:

$$maxap\supseteq maxap\cdot(id\times prune_{\ll}). \tag{13}$$

Recall that $sumlen\cdot prefix = foldr\,(zero\cup add)\,(0,0)$. It is a trivial task to check that $\ll$ is reflexive and transitive, and that $zero\cup add$ is monotonic on $\ll$. Therefore, we reason:

$$\begin{array}{ll} & maxap\cdot(id\times\Lambda(sumlen\cdot prefix))\cdot split_L\\ \supseteq & \quad\{\text{ by (13) }\}\\ & maxap\cdot(id\times prune_{\ll}\cdot\Lambda(sumlen\cdot prefix))\cdot split_L\\ \supseteq & \quad\{\text{ Pruning Theorem }\}\\ & maxap\cdot(id\times foldr\,(prune_{\ll}\cdot\Lambda((zero\cup add)\cdot distr))\\ & \{(0,0)\})\cdot split_L\end{array}$$

The next step is to refine $prune_{\ll}$ to a function. Corollary 5.2 basically says that if we define a function rm as below:

$$rm\,xs\quad=\quad\{x\in xs\mid\neg(\exists y : x.y\in xs\ \wedge\ x\le_{\mathsf{d}} y)\},$$

we have $rm\subseteq prune_{\ll}$. Note that it is *not* true, by defining $x\trianglelefteq y\equiv\forall z : z.x\le_{\mathsf{d}} z.y$, that $rm\subseteq thin_{\trianglelefteq}$. As stated in the previous section, Corollary 5.2 guarantees that for all header z, there exists some appendix as good as x. That particular appendix, however, is not always as good as x for any header z. That is why we need $prune_{\ll}$.

Recalling that x and y are actually SL-pairs, the condition in the set construction in rm is equivalent to:

$$\neg(\exists(s_2,l_2)\in xs : (l_1<l_2)\ \wedge\ (s_1/l_1\le(s_2-s_1)/(l_2-l_1))).$$

Some calculation shows that $s_1/l_1\le(s_2-s_1)/(l_2-l_1)$ is equivalent to $s_1/l_1\le s_2/l_2$ when $l_1<l_2$. Therefore we get the following definition of rm:

$$\begin{array}{l}rm\,xs=\{(s_1,l_1)\in xs\mid\\ \quad\neg(\exists(s_2,l_2)\in xs : (l_1<l_2)\ \wedge\ (s_1/l_1\le s_2/l_2))\}.\end{array}$$

Abbreviate $rm\cdot\Lambda((zero\cup add)\cdot distr)$ to $rmadd$. Since $(0,0)$ has the least length and thus cannot be removed, $rmadd$ can be simplified to:

$$rmadd\,(a,xs)=\{(0,0)\}\cup rm\,(\mathsf{E}add_a\,xs).$$

So far, the specification $msdl$ has been refined to:

$$\begin{array}{lll}msdl & \supseteq & max\cdot\mathsf{E}(maxap\cdot\\ & & \quad(id\times foldr\,rmadd\,\{(0,0)\})\cdot split_L)\cdot\Lambda suffix.\end{array}$$

To apply properties (5) and (6), we try to turn $(id\times foldr\,rmadd\,\{(0,0)\})\cdot split_L$ into a fold. Since $split_L$ can be written as a fold:

$$\begin{array}{l}split_L = foldr\,shift\,([\,],[\,]),\\ shift\,(a,(x,y))=(a{:}x,y),\ \text{ if } len\,x<L,\\ shift\,(a,(x+\!\!+[b],ys))=(a{:}x,b{:}y),\ \text{ otherwise.}\end{array}$$

By fold fusion we get:

$$\begin{array}{lll}msdl & \supseteq & max\cdot\mathsf{E}(maxap\cdot foldr\,rmshift\,([\,],\{(0,0)\}))\cdot\\ & & \quad\Lambda suffix,\end{array}$$

$$\begin{array}{l}rmshift\,(a,(x,ys))=(a{:}x,ys),\ \text{ if } len\,x<L,\\ rmshift\,(a,(x+\!\!+[b],ys))=(a{:}x,rmadd\,(b,ys)),\ \text{ otherwise.}\end{array}$$

Figure 2 shows a fragment of the computation when $U=3$ and the input is $[d,e,f,3,10,-1,4,6,-5,11]$ for some d, e, and f whose values are not relevant to this example. The figure skips the first few steps and starts with the "buffer" x loaded with $[6,-5,11]$, while the set ys consists of only $(0,0)$, as shown in the right-most column. As each new number is pushed into x, the last element b of x is squeezed out and added into ys by $rmadd$. Except for the $(0,0)$s, densities of the SL-pairs in ys are written as subscripts. The pair $(-1,1)$ in the third column from the left, for example, is removed because its density -1 is smaller than $9/3=3$, and $(9,3)$ is removed because its density 3 is no larger than $15/5=3$. The role of $maxap$ is not yet shown in Figure 2. It is supposed to, for each column, pick an SL-pair in ys that yields the largest density when combined with x.

5.4 Refining to a Linear-Time Algorithm

Still, more analysis on the structure of the set is needed to speed up the algorithm. We first note that $(0,0)$ is again always in the set.

Let us store the the SL-pairs of appendants in a deque in increasing lengths. Apart from $(0,0)$, they come in decreasing density. This is due to the constraint maintained by *rm*: given two surviving appendants x and $x.y$ where the latter is longer, we must have $x >_d y$, thus $x >_d x.y$. Lin et al. [24] defined a *right-screw* segment to be a segment x such that $take_i\, x \leq_d drop_i\, x$ for $0 \leq i \leq len\, x$. The appendants we collect are exactly the *decreasing* right-screw segments.

Since the appendants come in decreasing density, *rm* may start from the front of *ys*, and compare consecutive elements only:

$$\begin{array}{lcll} rm\,[y] & = & [y], & \\ rm\,(y{:}z{:}ys) & = & y{:}z{:}ys, & \text{if } safe\, y\, z, \\ & = & rm\,(z{:}ys), & \text{otherwise,} \end{array}$$
$$\textbf{where } safe\,(s_1,l_1)\,(s_2,l_2) = s_1/l_1 > s_2/l_2.$$

The empty case is not needed because the deque is never empty. Like in the previous section, we can also introduce *diffs* to save the cost of applying add_b to every element in the deque. We will do so as the last step. Once it is done, all operations, apart from *maxap*, run in amortised linear time if *cons* and *last* (extracting from the tail) can be performed in $O(1)$ time.

How do we implement *maxap* efficiently? The following lemma is adapted from Lin et al. [24]:

LEMMA 5.4. Let $y_1 \ldots y_n$ be a fully pruned set of SL-pairs, representing the appendants derived from the same prefix and sorted by increasing lengths. Let k be the greatest index that maximises $div\,(x.y_k)$. Then for all $i \in [0, n-1]$, $i \geq k$ if and only if $div\,(x.y_i) > div\,(x.y_{i+1})$.

Therefore, *maxap* should find the smallest k such that $div\,(x.y_k) > div\,(x.y_{k+1})$. Lin et al. used a binary search to find such k. They also proved that the segment having the maximum density cannot be longer than $2L-1$: otherwise we can always cut it in half and pick the better half. Their algorithm is therefore $O(n \log L)$ (although their experiments showed that it was actually quicker to perform a naïve linear search on *ys*).

In fact, it is possible to do better. Let us temporarily assume a naïve, linear-searching implementation of *maxap*, but start the search from the right-end of the deque. The following function *cut* drops elements from the right-end until it sees a y_{k-1} such that $div\,(x.y_{k-1}) > div\,(x.y_k)$ stops to hold:

$$\begin{array}{l} cut :: [V] \to Deque\, SL \to DequeSL, \\ cut\, x\, [y_1] = [y_1], \\ cut\, x\, (ys +\!\!+ [y_{k-1}, y_k]) \\ \quad = ys +\!\!+ [y_{k-1}, y_k], \qquad \text{if } dav\,(x, y_{k-1}) \not> dav\,(x, y_k), \\ \quad = cut\, x\, (ys +\!\!+ [y_{k-1}]), \quad \text{otherwise.} \end{array}$$

With *cut*, we can define $maxap\,(x, ys) = dav\,(x, last\,(cut\, x\, ys))$.

The key observation leading us to a linear-time algorithm was made by Goldwasser et al. [18]. Assume that w yields the best density, m_1, among the prefixes of the list $w +\!\!+ y +\!\!+ z$. If the best density of $a : w +\!\!+ y +\!\!+ z$ comes from $a +\!\!+ w +\!\!+ y$, it cannot be larger than m_1. The observation is made precise in this lemma:

LEMMA 5.5. Let *ys* be a fully pruned set of appendants. Assume that $m_1 = maxap\,(x, ys)$ results from an appendent with length l_1, and $m_2 = maxap\,(rmshift\,(a, (x, ys)))$, results from an appendant with length l_2. If $l_2 > l_1$, then $m_2 < m_1$.

Lemma 5.5 implies that, after extending *ys* with *rmshift*, we only need to consider those appendants that are not longer than the appendant chosen by *maxap*. In other words, we have:

$$\begin{array}{r} maxap\,(rmshift\,(a, (x, ys))) \uparrow maxap\,(x, ys) = \\ maxap\,(rmshift\,(cut\,(a, (x, ys)))) \uparrow maxap\,(x, ys). \end{array} \tag{14}$$

$$\begin{array}{l} msdl \quad \supseteq \quad snd \cdot foldr\, step\,(([\,], ((0,0), [\,])), -\infty) \\ \\ step :: (V, ((Queue\, V, (SL, Deque\, SL)), V)) \\ \qquad \to ((Queue\, V, (SL, Deque\, SL)), V) \\ step\,(a, ((x, ((c,n), ys)), m)) = \\ \quad ((a{:}x, ((c,n), ys)), avg\,(a{:}x)) \qquad \text{if } len\, x < L \\ step\,(a, ((x +\!\!+ [b], ((c,n), ys)), m)) = \\ \quad ((a{:}x, (cn', ys')), m' \uparrow m) \qquad \text{otherwise} \\ \quad \textbf{where } cn' = (b+c, 1+n) \\ \qquad sl = sumlen\,(a{:}x) \\ \qquad ys' = cut\, sl\, cn'\,(rm\,((c,n){:}ys)) \\ \qquad m' = \textbf{if } null\, ys' \textbf{ then } div\, sl \textbf{ else } davd\,(sl.cn')\,(last\, ys') \\ \\ rm :: SL \to Deque\, SL \to Deque\, SL \\ rm\, cn\, [y] = [y] \\ rm\, cn\, (y{:}z{:}ys) = y{:}z{:}ys \qquad \text{if } avgd\, cn\, y > avgd\, cn\, z \\ \qquad = rm\, cn\, (z{:}ys) \qquad \text{otherwise} \\ \\ davd\,(c,n)\,(s,l) \quad = \quad div\,(c-s, n-l) \\ \\ cut :: SL \to SL \to Deque\, SL \to Deque\, SL \\ cut\, x\, cn\, [y] = \textbf{if } div\, x > davd\,(x.cn)\, y \textbf{ then } [\,] \textbf{ else } [y] \\ cut\, x\, cn\, (ys +\!\!+ [z,y]) = \textbf{if } davd\,(x.cn)\, z \not> davd\,(x.cn)\, y \\ \qquad\qquad \textbf{then } ys +\!\!+ [z,y] \textbf{ else } cut\, x\, cn\, (ys +\!\!+ [z]) \end{array}$$

Figure 3. Derived Program for MSDL.

Continuing with the derivation, we transform fold to scan, and eliminate the list introduced by *scanr* using fold fusion, like in the previous sections:

$$\begin{array}{ll} & max \cdot \mathsf{E}(maxap \cdot foldr\, rmshift\,([\,], \{(0,0)\})) \cdot \Lambda suffix \\ \supseteq & \quad \{ \text{ refining sets to deques, let } e = ([\,], [(0,0)]) \} \\ & maxlist \cdot map\,(maxap \cdot foldr\, rmshift\, e) \cdot tails \\ = & \quad \{ \text{ by (5) } \} \\ & maxlist \cdot map\, maxap \cdot scanr\, rmshift\, e \\ = & \quad \{ \text{ tupling } \} \\ & snd \cdot \langle cut \cdot head, maxlist \cdot map\, maxap \rangle \cdot scanr\, rmshift\, e \\ = & \quad \{ \text{ fold fusion } \} \\ & snd \cdot foldr\, step\,(e, -\infty). \end{array}$$

With the help of (14), the function *step* can be constructed as:

$$\begin{array}{l} step :: (V, (([V], Deque\, SL), V)) \to (([V], Deque\, SL), V), \\ step\,(a, ((x, ys), m)) = \\ \quad ((a{:}x, ys), avg\,(a{:}x)), \quad \text{if } len\, x < L, \\ step\,(a, ((x +\!\!+ [b], ys), m)) = \\ \quad ((a{:}x, ys'), m' \uparrow m), \quad \text{otherwise,} \\ \quad \textbf{where } ys' = cut\,(a{:}x)\,((0,0){:}rm\,(map\, add_b\, ys)) \\ \qquad m' = dav\,(a{:}x, last\, ys'). \end{array}$$

As the last step, we apply the transformation introducing *diffs* to lift $map\, add_b$ out. The final program for MSDL is given in Figure 3.[5] It is a short but complex program which, were it not derived, would be rather difficult to comprehend. It runs in linear time for the same reason as the previous section: consider the size of *ys* in *step*. Every recursive call to *rm* or *cut* decreases the size by 1.

6. Conclusion and Related Work

Extending the classical derivation for MSS, we have derived two linear-time algorithms tackling MSSU and MSDL. The idea of thinning determines the structure of the algorithm. The concept of

[5] Expository programs for the algorithms in this paper are available on the author's homepage.

the left-negative segments [24], introduced to solve MSSU, naturally emerges from our intuition choosing an ordering such that the monotonicity condition for the Thinning Theorem can be satisfied. For MSDL, we need an extended notion of monotonicity relating an element to a set. Right-screw segments are formed when we attempt to determine when a solution can be safely dropped.

The story does not end there. Like most derivations with the Thinning Theorem, the key to efficiency is to analyse the set of solutions and design a fast data structure. We would like to know whether previous work on data structures for segment problems and dynamic programming [15, 25] could help to formalise our reasoning. We have dealt with merely the basic form of MSSU and MSDL. It should be possible to unify their variations [10] under a general framework and construct efficient solutions. While this paper follows the framework of Bird and de Moor [5], Maehara [26] recently reviewed the algorithm for MSSU by Lin and Fan from the viewpoint of a calculational formalisation of the *windowing technique*. That group is working on an alternative formalisation of optimisation algorithms more suitable for automatic derivation.

Acknowledgements

The author would like to thank Max Schäfer and Akiko Nakata for checking through the first draft of this paper, Sharon Curtis and Roland Backhouse for insightful discussions, and Zhenjiang Hu and Akimasa Morihata for pointing me to Takanori Maehara's thesis, as well as painstakingly reading through my early draft and promptly giving me plenty of valuable and thorough comments.

References

[1] R. C. Backhouse, P. de Bruin, G. Malcolm, E. Voermans, and J. van der Woude. Relational catamorphisms. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier Science Publishers, 1991.

[2] J. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[3] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, number 36 in NATO ASI Series F, pages 3–42. Springer-Verlag, 1987.

[4] R. S. Bird. A calculus of functions for program derivation. In D. A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 287–308. Addison-Wesley, 1990.

[5] R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[6] R. S. Bird, J. Gibbons, and S.-C. Mu. Algebraic methods for optimization problems. In R. C. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, number Lecture Notes in Computer Science 2297, pages 281–307. Springer-Verlag, January 2002.

[7] Y. H. Chen, H.-I. Lu, and C. Y. Tang. Disjoint segments with maximum density. In *Proceedings of the 2005 International Workshop on Bioinformatics Research and Applications (IWBRA 05)*, Lecture Notes in Computer Science 3515, pages 845–850, May 2005.

[8] T.-R. Chuang and B. Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[9] K.-M. Chung and H.-I. Lu. An optimal algorithm for the maximum-density segment problem. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA 2003)*, Lecture Notes in Computer Science 2832, pages 136–147, September 2003.

[10] K.-M. Chung and H.-I. Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34(2):373–387, 2004.

[11] S. Curtis. Dynamic Programming: a different perspective. In *Proceedings of the IFIP TC2 Working Conference on Algorithmic Languages and Calculi, Le Bischenberg, France*, February 1997.

[12] S. Curtis. The classification of greedy algorithms. *Science of Computer Programming*, 49(1-3):125–157, 2003.

[13] E. W. Dijkstra and W. H. J. Feijen. *Een methode van programmeren*. Academic Service, 's Gravenhage, the Netherlands, 1984.

[14] T.-H. Fan, S. Lee, H.-I. Lu, T.-S. Tsou, T.-C. Wang, and A. Yao. An optimal algorithm for maximum-sum segment and its application in bioinformatics. In *Proceedings of the 8th International Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science 2759, pages 46–66. Springer-Verlag, July 2003.

[15] Y. Futamura, C. Shirai, Y. Liu, N. Futamura, and K. Kakehi. Data structures for solving programming problems concerning segments in a sequence. In *Workshop on Algorithm Engineering (WAE 97)*, pages 11–13, 1997.

[16] J. Gibbons. An introduction to the Bird-Meertens formalism. *New Zealand Formal Program Development Colloquium Seminar, Hamilton*, November 1994.

[17] J. Gibbons. Calculating functional programs. In *Proceedings of ISRG/SERG Research Colloquium*. Oxford Brookes University, November 1997.

[18] M. H. Goldwasser, M.-Y. Kao, and H.-I. Lu. Fast algorithms for finding maximum-density segments of a sequence with applications to bioinformatics. In *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics (WABI 2002)*, pages 157–171, September 2002.

[19] M. H. Goldwasser, M.-Y. Kao, and H.-I. Lu. Linear-time algorithms for computing maximum-density sequence segments with bioinformatics applications. *Journal of Computer and System Sciences*, 70(2):128–144, 2005.

[20] D. Gries. The maximum-segment-sum problem. In E. W. Dijkstra, editor, *Formal Development Programs and Proofs*, University of Texas at Austin Year of Programming Series, pages 33–36. Addison-Wesley, 1989.

[21] J. T. Jeuring. The derivation of on-line algorithms, with an application to finding palindromes. *Algorithmica*, 11:146–184, 1994.

[22] A. Kaldewaij. *Programming: the Derivation of Algorithms*. Prentice Hall, 1990.

[23] H. C. Lau, T. H. Ngo, and B. N. Nguyen. Finding a length-constrained maximum-sum or maximum-density subtree and its application to logistics. *Discrete Optimization*, 3(4), 2006.

[24] Y.-L. Lin, T. Jiang, and K.-M. Chao. Efficient algorithms for locating the length-constrained heaviest segments, with applications to biomolecular sequence analysis. In *Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 2420, pages 459–470. Springer-Verlag, 2002.

[25] Y. A. Liu and S. D. Stoller. Program optimization using indexed and recursive data structures. In *Proceedings of ACM SIGPLAN 2002 Symposium on Partial Evaluation and Program Manipulation (PEPM)*, pages 108–118. ACM Press, January 2002.

[26] T. Maehara. プログラム運算によるWindowing技法の定式化とその応用 (Formalization and Application of the Windowing Technique based on Program Calculation). Bachelor's Thesis (in Japanese), University of Tokyo, February 2007.

[27] M. Rem. Small programming exercises 20. *Science of Computer Programming*, 10(1):99–105, 1988.

[28] I. Sasano, Z. Hu, and M. Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In *ACM SIGPLAN Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, Lecture Notes in Computer Science 2196, pages 72–91. Springer-Verlag, September 2001.

[29] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it Practical: A Generic Linear-Time Algorithm for Solving Maximum-Weightsum Problems. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 137–149. ACM Press, September 2000.

[30] J. P. H. W. van den Eijnde. Left-bottom and right-top segments. *Science of Computer Programming*, 15(1):79–94, 1990.

[31] H. Zantema. Longest segment problems. *Science of Computer Programming*, 18(1):39–66, 1992.