JFP **21** (2): 119–128, 2011. © Cambridge University Press 2011
doi:10.1017/S0956796810000316 First published online 5 January 2011

FUNCTIONAL PEARL

Building a consensus: A rectangle covering problem

RICHARD S. BIRD
Computing Laboratory, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
(*e-mail:* bird@comlab.ox.ac.uk)

1 Introduction

The other day, over a very pleasant lunch in the restaurant of Oxford's recently renovated Ashmolean Museum,[1] Oege de Moor gave me a problem about rectangles. The problem is explained more fully later, but roughly speaking one is given a finite set of rectangles RS and a rectangle R completely covered by RS. The task is to construct a single rectangle covering R among the elements of a larger set of rectangles associated with RS, called the *saturation* of RS. The saturation of RS is the closure of RS under so-called *consensus* operations, a term coined in (Quine, 1959), in which two rectangles are combined in two distinct ways to form new rectangles. The rectangle problem is a simplified version of containment-checking, a crucial component in a type inference algorithm for Datalog programs (Schäfer & de Moor, 2010). In the Schäfer-de Moor algorithm the problem is generalised to cubes in n-space rather than rectangles in two-space, the components of each cube are given by propositional formulae rather than by intervals on the real line, and certain equality and inhabitation constraints are taken into account. Oege felt that the central proof, Lemma 15 in (Schäfer & de Moor, 2010), deserved to be simplified so he posed the rectangle problem as a special case. This pearl was composed in response to the challenge.

2 The problem

Consider Figure 1, which depicts four rectangles A, B, C and D, covering another rectangle X framed with dotted lines. It happens in this example that none of the four rectangles can be removed without uncovering some part of X, but that is not essential to the problem. Next, consider two binary operations, $\oplus_0$ and $\oplus_1$, on rectangles. Assuming A and B overlap, the operation $A \oplus_0 B$ forms a new rectangle by taking the union of the horizontal intervals defining A and B and the intersection of the vertical intervals. The operation $A \oplus_1 B$ takes the intersection of the horizontals

[1] Well worth a visit if you are ever in Oxford, especially as entry to the museum is free.

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

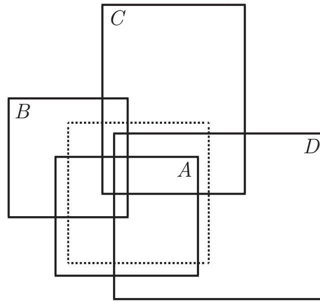

Fig. 1. Four rectangles covering the dotted rectangle.

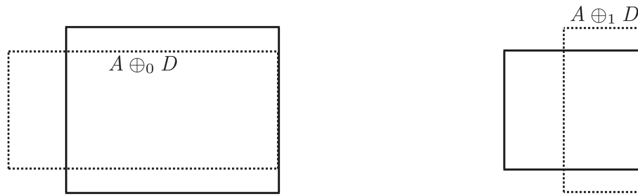

Fig. 2. The dotted rectangles are $A \oplus_0 D$ and $A \oplus_1 D$.

and the union of the verticals. These two operations are the *consensus* operations on A and B. For example, Figure 2 shows the two consensus rectangles for the rectangles A and D of Figure 1. The consensus operations also apply when the two rectangles do not overlap, though the result is either the empty rectangle or two rectangles.

Formally we define a rectangle A to be a pair (X_A, Y_A) of intervals on the real line.[2] The points in A are the elements in the cartesian product $X_A \times Y_A$. The two consensus operations are defined by

$$
\begin{aligned}
A \oplus_0 B &= (X_A \cup X_B, Y_A \cap Y_B) \\
A \oplus_1 B &= (X_A \cap X_B, Y_A \cup Y_B)
\end{aligned}
$$

Both $\oplus_0$ and $\oplus_1$ are associative, commutative and idempotent; moreover each operation distributes over the other. These properties follow from the same properties of union and intersection. By definition, the *saturation* of a set of rectangles RS is the least set XS containing RS and containing $A \oplus_0 B$ and $A \oplus_1 B$ for all A and B in XS. Assuming RS covers R, our task is to construct a single rectangle X in the saturation of RS that covers R. We do so by building a *consensus expression* that describes X. A consensus expression is a binary tree whose leaves are labelled with suitable names for the rectangles in RS and whose nodes are labelled with one or other of the two consensus operations. Failure to construct a consensus means that the rectangles RS do not cover R. To appreciate the problem, the reader is invited to construct a consensus expression over the four rectangles A, B, C, D of

[2] The assumption that X_A and Y_A are intervals is considered further in Section 4.1.

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

Figure 1 that covers the fifth rectangle X. The answer is given at the beginning of the following section.

The rectangle problem admits an immediate generalisation to n-cubes, which are cubes in n-dimensional space. An n-cube A is defined by an n-tuple of sets $[X_A, Y_A, \ldots, Z_A]$. The points in A are the elements of $X_A \times Y_A \times \cdots \times Z_A$, and there are n consensus operations $\oplus_j$ for $0 \leqslant j < n$. The value of $A \oplus_j B$ is the cube in which the kth component for $k \neq j$ is the intersection of the kth components of A and B for $0 \leqslant k < n$, and the jth component is the union of the jth components of A and B.

Finally, let us briefly relate the above consensus operations to those of same name in the Quine–McCluskey method for finding prime implicants. Consider a boolean formula in conjunctive normal form over n propositional variables. For example, taking $n = 3$ we might have the expression

$$a'b'c + abc' + a'bc + abc$$

in which the propositional variables are a, b and c, and dashed letters denote negation. By definition, the consensus of $a'b'c$ and $a'bc$ is the formula $a'c$, equivalently $a'(b + b')c$. If we interpret a term $a'b'c$ as a cube $[a', b', c]$ in which a' and b' are the complements of a and b with respect to some fixed universe, then the consensus of $a'b'c$ and $a'bc$ is just $[a', b', c] \oplus_1 [a', b, c]$.

3 Proof and first construction

One way of covering X in Figure 1 is with the rectangle

$$(B \oplus_0 C) \oplus_1 (A \oplus_0 C) \oplus_1 (A \oplus_0 D)$$

but there are many other expressions that also do the job.

Now, generalising from rectangles to n-cubes, how shall we prove that there is a consensus expression over a set of n-cubes XS that covers a given cube X if and only if XS does? To answer this question let us first make a simplifying assumption, namely that each component of each cube is a *finite set of integers*. We will see in Section 4 that this assumption entails no loss of generality. Given that each component has finite size, we can prove the theorem by induction on the size of X, defined to be the product of the sizes of the components of X. A cube X of size 1 has singleton components and so contains just a single point. A single point can clearly be covered by a single cube in XS if and only if XS covers X. This establishes the base case. Otherwise, by way of induction, assume that all cubes of size less than s can be covered by a single cube in the saturation of XS. Let X be a cube of size $s > 1$, so at least one component of X is a non-singleton set. Say the kth component is not a singleton. Split X into two cubes Y and Z by splitting the kth component into two smaller sets, leaving the other components unchanged. Each of Y and Z has size smaller than s, so by induction there exist cubes A and B in the saturation of XS that cover Y and Z, respectively. Now X is covered by $A \oplus_k B$.

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

The proof easily converts into a constructive procedure for computing the required expression. Turning to Haskell, we first declare

$$\begin{array}{lcl} \textbf{type}\ Cube & = & [Component] \\ \textbf{type}\ Component & = & [Int] \end{array}$$

A cube is a list of n components and each component is a finite, possibly empty list of integers in strictly increasing order. We allow empty components because the result of a consensus operation might be the empty cube. Each cube represents a set of points in n-space:

$$\begin{array}{lcl} \textbf{type}\ Point & = & [Int] \\ points & :: & Cube \rightarrow [Point] \\ points\ [xs] & = & [[x] \mid x \leftarrow xs] \\ points\ (xs : xss) & = & [x : ys \mid x \leftarrow xs,\ ys \leftarrow points\ xss] \end{array}$$

A set xcs of cubes covers a cube xc if for each point of xc there is some member of xcs that contains it:

$$\begin{array}{lcl} covers & :: & [Cube] \rightarrow Cube \rightarrow Bool \\ covers\ xcs\ xc & = & and\ [contains\ xcs\ p \mid p \leftarrow points\ xc] \\ contains\ xcs\ p & = & or\ [p \in points\ xc \mid xc \leftarrow xcs] \end{array}$$

Any set of cubes trivially covers the empty cube because the empty cube contains no points.

The operation $\oplus_k$ is implemented as a function $consensus\ k$:

$$\begin{array}{lcl} consensus & :: & Int \rightarrow Cube \rightarrow Cube \rightarrow Cube \\ consensus\ k\ xc\ yc & = & zipWith\ cap\ xc1\ yc1 \mathbin{+\!\!+} [cup\ xs\ ys] \mathbin{+\!\!+} zipWith\ cap\ xc2\ yc2 \\ & & \quad \textbf{where}\ (xc1, xs : xc2) = splitAt\ k\ xc \\ & & \quad \phantom{\textbf{where}}\ (yc1, ys : yc2) = splitAt\ k\ yc \end{array}$$

Definitions of cup and cap, the union and intersection functions on ordered lists, are omitted.

Next, we declare a suitable type for consensus expressions:

$$\begin{array}{lcl} \textbf{data}\ Exp & = & Val\ Label \mid Op\ Int\ Exp\ Exp \\ \textbf{type}\ Label & = & Int \end{array}$$

The evaluation function $eval$ is defined by

$$\begin{array}{lcl} eval & :: & [Cube] \rightarrow Exp \rightarrow Cube \\ eval\ xcs\ (Val\ j) & = & xcs\ !!\ j \\ eval\ xcs\ (Op\ k\ u\ v) & = & consensus\ k\ (eval\ xcs\ u)\ (eval\ xcs\ v) \end{array}$$

Cubes are labelled by their positions in the given list of cubes.

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

The function $cover0$ is a straightforward implementation of the induction proof (later on, we consider other versions, $cover1$, $cover2$, and so on):

```
cover0  ::  [Cube] → Cube → Exp
cover0 xcs xc
        =   if null yss
            then Val (head [j | (j, yc) ← zip [0 ..] xcs, covers [yc] xc])
            else Op (length xss) (cover0 xcs yc) (cover0 xcs zc)
            where
            (xss, yss)  =  span single xc
            yc          =  xss ++ [[head (head yss)]] ++ tail yss
            zc          =  xss ++ [tail (head yss)] ++ tail yss
```

The test $single$ returns $True$ on a singleton list and $False$ otherwise. The expression $span\ single\ xc$ splits the components of xc into two lists, xss and yss, where xss consists of singleton sets only and yss is either empty or begins with a non-singleton set. If yss is empty, then xc consists of a single point. By construction,

$$covers\ xcs\ xc \quad \Rightarrow \quad covers\ [eval\ xcs\ (cover0\ xcs\ xc)]\ xc$$

For example, $cover0\ xcs\ [[1,2,3],[1,2],[1,2]]$ produces the expression

$$\begin{array}{l}((111 \oplus_2 112) \oplus_1 (121 \oplus_2 122)) \oplus_0 \\ ((211 \oplus_2 212) \oplus_1 (221 \oplus_2 222)) \oplus_0 \\ ((321 \oplus_2 312) \oplus_1 (311 \oplus_2 322))\end{array}$$

in which the term abc represents the label of the first cube in xcs that covers the point $[a,b,c]$.

4 Digitization

We now show why we can assume that the components of cubes are finite sets of integers. An interval on the real line is defined by a pair (a,b) of numbers representing the set of real numbers x in the range $a \leqslant x < b$. The length of (a,b) is $b-a$, so a pair (a,a) denotes an empty interval. There is no loss in generality in requiring a and b to be integers, but, even so, there is an infinite number of points in each nonempty interval.

The solution is to digitize real intervals as integer intervals. Consider for example the three rectangles of Figure 3. Suppose their component intervals are as follows:

$$A = [(0,50),(40,80)],\ B = [(30,70),(10,60)],\ C = [(40,80),(0,50)]$$

The dotted lines shown in the figure extend each horizontal and vertical line to the x and y axes and partition the axes into five horizontal segments

$$H_0 = (0,30),\ H_1 = (30,40),\ H_2 = (40,50),\ H_3 = (50,70),\ H_4 = (60,80)$$

and five vertical segments

$$V_0 = (0,10),\ V_1 = (10,40),\ V_2 = (40,50),\ V_3 = (50,60),\ V_4 = (60,80)$$

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

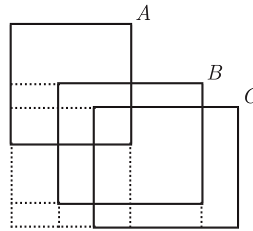

Fig. 3. Dividing the axes into segments.

By construction, the components of A, B and C can each be expressed as the union of segments. For example, $X_A = H_0 \cup H_1 \cup H_2$ and $Y_A = V_2 \cup V_3 \cup V_4$. Going one step further, we can now represent each rectangle by the lists of integer labels of the horizontal and vertical segments that make up its coordinates. That gives new rectangles

$$A = [[0 \mathinner{..} 2], [2 \mathinner{..} 4]],\; B = [[1 \mathinner{..} 3], [1 \mathinner{..} 3]],\; C = [[2 \mathinner{..} 4], [0 \mathinner{..} 2]]$$

whose components are intervals of integers. Moreover, containment-checking on the original problem returns the same result as on the digitized version. However, digitization is possible only because each cube is assumed to have components that are intervals.

To implement digitization we first declare the types

$$\begin{aligned}&\textbf{type}\ \mathit{RealCube} = [\mathit{Interval}]\\&\textbf{type}\ \mathit{Interval} = (\mathit{Int}, \mathit{Int})\end{aligned}$$

We digitize each list of corresponding components separately:

$$\begin{array}{lcl}\mathit{digitize} & :: & [\mathit{RealCube}] \to [\mathit{Cube}]\\ \mathit{digitize} & = & \mathit{transpose} \cdot \mathit{map}\ \mathit{digitizeRow} \cdot \mathit{transpose}\end{array}$$

The list of cubes is transposed to bring each corresponding component into the same row. Each row is then digitized and the result is transposed back to give a new list of cubes. To implement *digitizeRow*, we first sort the boundaries of each interval into strictly increasing order, discarding duplicates:

$$\begin{array}{lcl}\mathit{boundaries} & :: & [\mathit{Interval}] \to [\mathit{Int}]\\ \mathit{boundaries} & = & \mathit{sort} \cdot \mathit{concatMap}\ (\lambda(a,b) \to [a,b])\end{array}$$

The definition of *sort* is omitted. The function *mksegs* constructs a new list of intervals out of the boundaries:

$$\begin{array}{lcl}\mathit{mksegs} & :: & [\mathit{Int}] \to [\mathit{Interval}]\\ \mathit{mksegs}\ \mathit{xs} & = & \mathit{zip}\ \mathit{xs}\ (\mathit{tail}\ \mathit{xs})\end{array}$$

Each new interval is then encoded as a component:

$$\begin{array}{lcl}\mathit{encode} & :: & [\mathit{Interval}] \to \mathit{Interval} \to \mathit{Component}\\ \mathit{encode}\ \mathit{row}\ (x,y) & = & [j \mathinner{..} k]\\ \quad\quad \textbf{where}\ j & = & \mathit{head}\ [j \mid (j,(a,b)) \leftarrow \mathit{jabs},\ a = x]\\ \quad\quad\quad\quad k & = & \mathit{head}\ [j \mid (j,(a,b)) \leftarrow \mathit{jabs},\ b = y]\\ \quad\quad\quad\quad \mathit{jabs} & = & \mathit{zip}\ [0 \mathinner{..}]\ \mathit{row}\end{array}$$

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

That gives

$$digitizeRow\ row \quad = \quad map\,(encode\,(mksegs\,(boundaries\ row)))\ row$$

4.1 Propositional components

We mentioned briefly in the Introduction that in Datalog containment-checking the components of rectangles were propositional formulae rather than intervals. For example, consider the rectangles

$$\begin{array}{lcl} A & = & (a+c, a+bc) \\ B & = & (a, b'+c) \\ C & = & (bc, a+b'c) \\ D & = & (a+bc, b') \\ X & = & (a+bc, a+b') \end{array}$$

over the propositional variables a, b and c. The rectangle X is covered by A, B, C and D in the sense that

$$\begin{array}{lcl} (a+bc) & \Rightarrow & (a+c)+a+bc+(a+bc) \\ a+b' & \Rightarrow & (a+bc)+(b'+c)+(a+b'c)+b' \end{array}$$

We can convert this version of the rectangle problem into an integer version by coding formulae using the truth table for a, b and c:

	abc		*abc*
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Using the labels of the entries in this table, the five rectangles above translate to the integer rectangles

$$\begin{array}{lcl} A & = & ([1,3,4,5,6,7],[3,4,5,6,7]) \\ B & = & ([4,5,6,7],[0,1,3,4,5]) \\ C & = & ([3,7],[1,4,5,6,7]) \\ D & = & ([3,4,5,6,7],[0,1,4,5]) \\ X & = & ([3,4,5,6,7],[0,1,4,5,6,7]) \end{array}$$

Of course, the components are not intervals, but *cover*0 still works because it did not assume interval components.

5 Optimisation

Let us now return to cubes whose components are lists of integers. The trouble with *cover*0 *xcs xc* is that each point of *xc* is associated with a cube in *xcs* and these cubes are then assembled into a potentially very large consensus expression. Because adjacent points of *xc* may well be associated with the same cube, and each consensus operation is idempotent, there is room for improvement both in speed and expression size.

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

One way to reduce the size of the final consensus expression is to reduce the size of the cubes using digitization. For example, consider again the rectangles of Figure 3, this time as intervals of integers:

$$A = [[0\,..\,49],[40\,..\,79]],\ B = [[30\,..\,69],[10\,..\,59)],\ C = [[40\,..\,79],[0\,..\,49]]$$

Each of these rectangles is of size 2000, which is quite large. But digitization converts the problem into one whose rectangles each have size 9, a substantial reduction. The function *cover* 1 is defined by

$$\begin{array}{lcl} cover1 & :: & [Cube] \to Cube \to Exp \\ cover1\ xcs\ xc & = & cover0\ ycs\ yc \\ \quad \textbf{where}\ yc : ycs = digitize\,(map\,(map\ toPair)\,(xc : xcs)) & & \\ toPair\ xs & = & (head\ xs, last\ xs + 1) \end{array}$$

The function *cover* 1, which is a valid optimisation only if the components of the cubes are contiguous intervals of integers, may reduce the size of the consensus expression but not necessarily minimize it.

In search of an algorithm that does minimize expression size, let us first recast *cover* 0 in iterative form. To this end, consider the function *expand k* that splits a cube into s_k subcubes, where s_k is the size of the kth component:

$$\begin{array}{lcl} expand & :: & Int \to Cube \to [Cube] \\ expand\ kxc & = & [xss +\!\!+ [[y]] +\!\!+ yss \mid y \leftarrow ys] \\ & & \textbf{where}\ (xss, ys : yss) = splitAt\ k\ xc \end{array}$$

Using *expand* we can define

$$\begin{array}{lcl} cover2\ xcs\ xc = pcover\ 0\ xc & & \\ \textbf{where} & & \\ pcover\ kxc & = & \textbf{if}\ k = n \\ & & \textbf{then}\ Val\,(head\,[j \mid (j,yc) \leftarrow zip\,[0\,..\,]\,xcs,\ covers\,[yc]\,xc]) \\ & & \textbf{else}\ foldr1\,(Op\ k)\,(map\,(pcover\,(k{+}1))\,(expand\ k\ xc)) \\ n & = & length\ xc \end{array}$$

We omit the proof that *cover* 0 = *cover* 2. The function *cover* 2 is faster than *cover* 0, though not dramatically so. However, *cover* 2 gets us, quite literally, to the nub of the problem. Let $\oplus$ be some associative, commutative and idempotent operation. The critical fact about $\oplus$ is that

$$foldr1\,(\oplus) \quad = \quad foldr1\,(\oplus) \cdot nub$$

where *nub* is the standard Haskell library function that removes duplicates from a list. The value of $foldr1\,(\oplus)\,xs$ depends only on the *set* of elements in *xs* and not on the order of the elements or on duplications.

We could minimize the size of the final consensus expression simply by inserting a *nub* after $foldr1\,(Op\ k)$ in the definition of *pcover*, but *nub* uses a quadratic number of equality tests and each equality test compares two consensus expressions, so the new version, *cover* 3 say, may well be slower than *cover* 2.

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

Here is a better way to remove duplicates. The idea is to convert the target cube xc to an n-dimensional array of points, then to replace each point by the label of the first cube in xcs which covers it. This array of labels is then pruned, dimension by dimension, by removing all duplicates in each row, then all duplicate rows, all duplicate planes, and so on. To represent an n-dimensional array in Haskell we need trees:

$$\textbf{data}\ \mathit{Tree}\ a \;=\; \mathit{Leaf}\ a \mid \mathit{Fork}\ [\mathit{Tree}\ a]$$

The function *convert* converts an n-cube into a list of trees of points:

$$\begin{array}{lcl}
\mathit{convert} & :: & \mathit{Cube} \rightarrow [\mathit{Tree\ Point}] \\
\mathit{convert}\ [xs] & = & [\mathit{Leaf}\ [x] \mid x \leftarrow xs] \\
\mathit{convert}\ (xs : xss) & = & [\mathit{Fork}\ (\mathit{map}\ (\mathit{cons}\ x)\ ts) \mid x \leftarrow xs] \\
 & & \textbf{where}\ ts = \mathit{convert}\ xss \\
\mathit{cons}\ x\ (\mathit{Leaf}\ xs) & = & \mathit{Leaf}\ (x : xs) \\
\mathit{cons}\ x\ (\mathit{Fork}\ ts) & = & \mathit{Fork}\ (\mathit{map}\ (\mathit{cons}\ x)\ ts)
\end{array}$$

The function *covering xcs* labels a tree of points with the first cube in *xcs* that covers each point:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{covering}\ ::\ [\mathit{Cube}] \rightarrow \mathit{Cube} \rightarrow [\mathit{Tree\ Label}]} \\
\multicolumn{3}{l}{\mathit{covering}\ xcs\ xc\ =\ \mathit{map}\ \mathit{label}\ (\mathit{convert}\ xc)} \\
\multicolumn{3}{l}{\textbf{where}} \\
\mathit{label}\ (\mathit{Leaf}\ xs) & = & \mathit{Leaf}\ (\mathit{lookup}\ xs) \\
\mathit{label}\ (\mathit{Fork}\ ts) & = & \mathit{Fork}\ (\mathit{map}\ \mathit{label}\ ts) \\
\mathit{lookup}\ x & = & \mathit{head}\ [j \mid (j, xc) \leftarrow \mathit{zip}\ [0\,..\,]\ xcs,\ \mathit{member}\ x\ xc]
\end{array}$$

The function *member* is defined by

$$\mathit{member}\ x\,xc \;=\; \mathit{and}\ [a \leqslant y \wedge y < b \mid ((a,b),y) \leftarrow \mathit{zip}\ (\mathit{map}\ \mathit{toPair}\ xc)\ x]$$

The function *prune* removes duplicate labels:

$$\begin{array}{lcl}
\mathit{prune} & :: & [\mathit{Tree\ Label}] \rightarrow [\mathit{Tree\ Label}] \\
\mathit{prune}\ ts@(\mathit{Leaf}\ _ : _) & = & \mathit{remdups}\ ts \\
\mathit{prune}\ ts & = & \mathit{remdups}\ [\mathit{Fork}\ (\mathit{prune}\ vs) \mid \mathit{Fork}\ vs \leftarrow ts] \\
\mathit{remdups} & :: & \mathit{Eq}\ a \Rightarrow [a] \rightarrow [a] \\
\mathit{remdups} & = & \mathit{foldr}\ \mathit{op}\ [\,] \\
\quad \textbf{where}\ \mathit{op}\,x\ [\,] & = & [x] \\
\qquad\quad \mathit{op}\,x\ (y : ys) & = & \textbf{if}\ x = y\ \textbf{then}\ y : ys\ \textbf{else}\ x : y : ys
\end{array}$$

The function *remdups* removes adjacent duplicates only, but that is sufficient because in the cubes problem all duplicates will occur together. The function *lookup* always returns the first cube in *xcs* that covers the point x and that fact guarantees that duplicate cubes occur together.

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press

Now we can define our final version:

$$
\begin{array}{lcl}
\mathit{cover}4 & :: & [\mathit{Cube}] \to \mathit{Cube} \to \mathit{Exp} \\
\mathit{cover}4\ \mathit{xcs}\ \mathit{xc} & = & \mathit{mkexp}\ 0\ (\mathit{prune}\ (\mathit{covering}\ \mathit{ycs}\ \mathit{yc})) \\
\quad \textbf{where}\ \mathit{yc} : \mathit{ycs} = \mathit{digitize}\ (\mathit{map}\ (\mathit{map}\ \mathit{toPair})\ (\mathit{xc} : \mathit{xcs})) & & \\
\mathit{mkexp}\ k\ \mathit{ts}@(\mathit{Leaf}\ _ : _) & = & \mathit{foldr}1\ (\mathit{Op}\ k)\ [\mathit{Val}\ x \mid \mathit{Leaf}\ x \leftarrow \mathit{ts}] \\
\mathit{mkexp}\ k\ \mathit{ts} & = & \mathit{foldr}1\ (\mathit{Op}\ k)\ [\mathit{mkexp}\ (k{+}1)\ \mathit{vs} \mid \mathit{Fork}\ \mathit{vs} \leftarrow \mathit{ts}]
\end{array}
$$

6 Experimental results

Using ghci we carried out a brief comparison of the five versions of *cover*. Recall that *cover* 0 was the divide and conquer version, *cover* 1 the version that reduced problem size by digitization, *cover* 2 was the linearised form of *cover* 0, *cover* 3 used the Haskell *nub* function, and *cover* 4 the super-duper version that used both digitization and a tailored implementation for removing duplicates. The results, using three sample sets of cubes, are summarised in the table below. Times are in seconds, and size refers to the size of the consensus expression. The set *xcs* 0 is a set of rectangles of total area 525, the set *xcs* 1 a set of rectangles of total area 161, and *xcs* 2 a set of four-dimensional cubes of area 8097.

	xcs 0		*xcs* 1		*xcs* 2	
	time	size	time	size	time	size
cover 0	0.05	241	1.64	503	38.97	1295
cover 1	0.00	49	0.02	31	0.06	23
cover 2	0.05	241	1.63	503	38.89	1295
cover 3	0.05	17	1.59	3	38.75	9
cover 4	0.02	17	0.02	3	0.02	9

Clearly *cover* 4 was best by a long way, so it was worth the effort to obtain it.

Acknowledgments

The author thank Jeremy Gibbons and two anonymous referees for their constructive remarks and suggestions. As a result the pearl has been thoroughly overhauled.

References

Quine, W. V. (1959) On cores and prime implicants of truth functions. *Am. Math. Mon.*, **66**(9), 755–760.

Schäfer, M. & de Moor, O. (2010) Type inference for datalog with complex type hierarchies. *Princ. Program. Lang. 2010. ACM SIGPLAN Not.*, **45**(1), 145–156.

https://doi.org/10.1017/S0956796810000316 Published online by Cambridge University Press