JFP **14** (6): 613–622, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S095679680400512X Printed in the United Kingdom

FUNCTIONAL PEARL
On tiling a chessboard

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

Teacher: Good morning class. Today I would like you to construct a program for finding out how many ways a chessboard can be tiled with dominoes. For those who don't play games, a chessboard is a 8×8 board divided into 64 squares, and a domino is a 2×1 tile which can cover two squares of the board either vertically or horizontally. For example, a 2×2 board can be covered with two dominoes in exactly two ways, with both dominoes horizontal or both vertical.

Fun: Sounds like a fun problem. I would start by thinking about a function $tilings :: (Int, Int) \rightarrow [Tiling]$ such that $tilings\,(m, n)$ returns a list of all the distinct tilings of an $m \times n$ board. There seems no harm to me in generalising the problem right at the start. Having constructed *tilings* we can compose it with *length* to get the answer. That's the great thing about lazy functional programming, you can separate a program into its component parts and program compositionally without loss of efficiency.

Data: Well, you can't tile a 3×3 board, or any rectangular board with two odd-numbered sides, without leaving a hole somewhere. Let me think of a suitable data structure for *Tiling*. It should be a list of pairs of numbers, each pair representing the two squares that are covered by a single domino. I know a neat way of numbering the squares of a chessboard that ...

Set: Data, why do you always want to plunge straight in with a data structure? We have no idea of how we want to process elements of *Tiling* yet, so no idea of what to represent. And Fun, why do you always immediately go for lists? We might want a tree of tilings or some other structure to represent the *set* of possible tilings. I would think about a function *tilings* with type $(Int, Int) \rightarrow Set\ Tiling$ and worry about the representation later.

Rel: I would go further than Set and think about a relation $tiling :: (Int, Int) \rightsquigarrow Tiling$ that returned an arbitrary tiling. As you know, I think of relations as nondeterministic functions. Of course, we have to ensure that every legitimate tiling was a possible outcome of *tiling*.

Set: Rel, there you go again wanting to bring relations into everything. This is clearly a problem about sets, so why won't set-valued functions suffice?

Rel: Simplicity of notation. I can think of a relation $putDomino :: Tiling \rightsquigarrow Tiling$ that adds one more domino to a partial tiling. You have to think of a function $putDomino :: Tiling \to Set\ Tiling$ that returns all the ways a single domino can be added. Consequently, when you want to compose two *putDomino* functions you have either to bring in a set comprehension, or use *union* and *mapSet* all over the place. Both of these add to the complexity of expressions. I, on the other hand, can use relational composition. Right at the end we can lift everything to the set level, but the notational overhead is less if we stick to relations for most of the reasoning.

Set: I don't see that it makes much difference, but since Rel and I agree basically about how to start, I am willing to go along with this relational stuff. In fact, I can see how to specify the problem using *putDomino*; we have

$$tiling\ (m,n) = putDomino^N \cdot empty$$

where $N = \lfloor m * n/2 \rfloor$. The relation *empty* returns an empty tiling, and P^n composes the relation P with itself exactly n times.

Fun: Yes, of course. I was already thinking of decomposing my *tilings* function into a repetition of *putDomino* functions when I proposed it. Given *putDomino* it is easy to code up the result as a Haskell program. There will be a lot of backtracking on encountering dead-ends, but one of the advantages of lazy functional programming is that we don't have to program the backtracking ourselves. Indeed this was emphasized as one of the great points about lazy functional programming when it was first proposed.

Rel: True, but a program without backtracking is going to be faster than one with it. Let me abbreviate *putDomino* simply to the letter P, and *empty* to E. To avoid backtracking we want a refinement $Q \subseteq P$ so that, again with $N = \lfloor m * n/2 \rfloor$,

$$\begin{aligned} Q^N \cdot E &= P^N \cdot E && (1)\\ range\,(Q^n \cdot E) &\subseteq domain\,Q \qquad \text{for } 0 \leqslant n < N && (2)\end{aligned}$$

Equation (1) says that we don't lose any full tilings by considering Q rather than P, and (2) says that any partial tiling that has been constructed using Q can always have another Q applied to it, so backtracking isn't needed.

In fact we can go further. Since $Q \subseteq P$ we can take $Q = S \cdot P$ where $S \subseteq id$, so S is what is called a *coreflexive* relation. You can think of S as a filter that allows only "safe" tilings through. Now it is easy to show by induction that (1) follows from

$$\begin{aligned} S \cdot E &= E && (3)\\ S \cdot P^N \cdot E &= P^N \cdot E && (4)\\ S \cdot P^{n+1} \cdot E &= S \cdot P \cdot S \cdot P^n \cdot E \qquad \text{for } 0 \leqslant n < N && (5)\end{aligned}$$

Equation (3) says that the empty tiling is safe, and (4) that the full tiling is safe. Equation (5) is a little trickier to interpret, but it says that given any safe nonempty tiling we can always *remove* a domino to give a safe tiling. From (5) we get $S \cdot P^n \cdot E = Q^n \cdot E$, and (1) follows from this and (4).

Set: I see that. Since $S \subseteq id$ we have $S \cdot P \cdot S \cdot P^n \cdot E \subseteq S \cdot P^{n+1} \cdot E$, so the interesting direction is $S \cdot P^{n+1} \cdot E \subseteq S \cdot P \cdot S \cdot P^n \cdot E$, which has the interpretation you have just given. See, Rel, I remember that relational composition is monotonic under inclusion. And surely (5) is also necessary for (1) because if some safe $(n+1)$-tiling is not achievable by applying a safe move to a safe n-tiling, then we will miss all of its completions, and (2) says that there is at least one such completion. But how do you simplify (2)?

Rel: Well, $range\, X \subseteq domain\, Y$ just in the case that $X \subseteq Y^{\mathrm{T}} \cdot Y \cdot X$, where Y^{T} denotes the converse of Y. The converse of a relation is just like the transposition of a matrix, so we will use the same notation. Now,

$$\begin{array}{ll}
 & range\,(Q^n \cdot E) \subseteq domain\, Q \\
\equiv & \quad \{\text{above remark}\} \\
 & Q^n \cdot E \subseteq Q^{\mathrm{T}} \cdot Q \cdot Q^n \cdot E \\
\equiv & \quad \{\text{since (5) implies } Q^n \cdot E = S \cdot P^n \cdot E\} \\
 & S \cdot P^n \cdot E \subseteq Q^{\mathrm{T}} \cdot S \cdot P^{n+1} \cdot E \\
\equiv & \quad \{\text{since } Q = S \cdot P\} \\
 & S \cdot P^n \cdot E \subseteq (S \cdot P)^{\mathrm{T}} \cdot S \cdot P^{n+1} \cdot E \\
\equiv & \quad \{\text{since } (S \cdot P)^{\mathrm{T}} = P^{\mathrm{T}} \cdot S^{\mathrm{T}} = P^{\mathrm{T}} \cdot S \text{ as } S^{\mathrm{T}} = S \text{ for a coreflexive } S\} \\
 & S \cdot P^n \cdot E \subseteq P^{\mathrm{T}} \cdot S \cdot S \cdot P^{n+1} \cdot E \\
\equiv & \quad \{\text{since } S \cdot S = S \text{ for a coreflexive } S\} \\
 & S \cdot P^n \cdot E \subseteq P^{\mathrm{T}} \cdot S \cdot P^{n+1} \cdot E
\end{array}$$

Hence (2) is equivalent to

$$S \cdot P^n \cdot E \quad \subseteq \quad P^{\mathrm{T}} \cdot S \cdot P^{n+1} \cdot E \qquad (6)$$

And (6) says that given any safe partial tiling we can always *add* a domino to give a safe tiling. So a safe tiling has to be one to which you can add or remove a domino and leave a safe tiling.

Fun: I don't really see where all this is leading.

Set: Rel's point is that if we can find an appropriate definition of a safe tiling, and only construct bigger safe tilings out of smaller ones, then we will still get all possible tilings without having to backtrack: all safe tilings can be completed to a full board. The interesting question now is: What is a safe tiling?

Data: How about taking a safe tiling as one that contains no holes? Its certainly true of the empty and full tiling, and I bet we can show that your conditions are met, Rel.

Set: I think you would lose your bet, Data. Consider a full board but with the top-left and top-right squares missing. This contains no holes, and we can tile this shape to reach the position, but the trouble is we cannot complete the tiling with an extra domino. By the way, I nearly said remove the bottom-left and top-right squares but, as every puzzle solver knows, these two squares have the same colour, so one cannot tile this shape with dominoes. Every domino covers two squares of different colours, so in any partial tiling the number of covered White squares equals the number of covered Black squares.

Data: Well, how about no holes and the unfilled squares all belong to connected components of even size?

Fun: That may work, but isn't it going to be hard to keep track of the connected components? We want safe positions that are easy to recognize.

Rel: I am still thinking about (5) and (6); surely the proof that we can add a domino should be similar to the proof that we can remove one. There is a duality here that we are not exploiting. Suppose we let D, pronounced "dual", be a nondeterministic function that takes a tiling t and returns some tiling of the squares not covered by t. So D together with t tiles the whole board. In symbols,

$$D \cdot P^n \cdot E \quad = \quad P^{N-n} \cdot E \qquad \text{for } 0 \leqslant n \leqslant N \tag{7}$$

In particular, the dual of the empty board is a full board and conversely. We also have

$$D \cdot P \quad = \quad P^{\mathrm{T}} \cdot D \tag{8}$$

In words, if we can obtain a tiling by adding a domino and then taking a dual, then we can obtain the same tiling by first taking the dual and then removing some domino. Equation (8) also holds on the full board because both sides denote the empty relation. Now I think we can show that (5) and (6) are the same thing if we also assume that $S \cdot D = D \cdot S$, that is, the dual of a safe tiling is a safe tiling of the dual. Let us assume (5) holds, which by a change of variable is equivalent to

$$S \cdot P^{N-n} \cdot E \quad = \quad S \cdot P \cdot S \cdot P^{N-n-1} \cdot E$$

for $0 \leqslant n < N$. Taking the dual of the left-hand side, we have

$$\begin{array}{cl} & D \cdot S \cdot P^{N-n} \cdot E \\ = & \quad \{\text{assuming } D \cdot S = S \cdot D\} \\ & S \cdot D \cdot P^{N-n} \cdot E \\ = & \quad \{\text{by } (7)\} \\ & S \cdot P^n \cdot E \end{array}$$

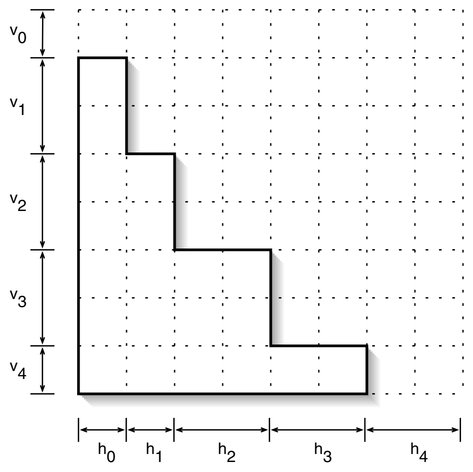
Doing the same for the right-hand side, we have

$$
\begin{array}{cl}
 & D \cdot S \cdot P \cdot S \cdot P^{N-n-1} \cdot E \\
= & \quad \{\text{assuming } D \cdot S = S \cdot D\} \\
 & S \cdot D \cdot P \cdot S \cdot P^{N-n-1} \cdot E \\
= & \quad \{\text{using (8)}\} \\
 & S \cdot P^{\mathrm{T}} \cdot D \cdot S \cdot P^{N-n-1} \cdot E \\
= & \quad \{\text{using } D \cdot S = S \cdot D \text{ again, and then (7)}\} \\
 & S \cdot P^{\mathrm{T}} \cdot S \cdot P^{n+1} \cdot E
\end{array}
$$

So we have proved (6). I am sure we can go the other way, but it's enough that we have reduced two proofs to one.

Data: Well, I can't follow all your details, but I do see that we want a safe tiling to be one whose dual is also safe. That obviously prevents us defining a safe shape as one with no holes, as its dual can have a very big hole! Let me think. (*Pause.*) I know! Let a safe tiling be one that looks like a step function. Suppose we start tiling in the bottom left-hand corner of the board. Then the step shape will be one that descends in steps from somewhere from up the left-hand side to the "floor" of the board. The dual of this step shape will also be a step shape, provided we view it from the top-right corner.

We can represent such a shape by a sequence of integers (v_i, h_i) for $0 \leqslant i \leqslant k$ for some k. The value v_0 is the drop from the top of the board to the top step (so $v_0 \geqslant 0$), and $h_0 > 0$ is the width of step number 0. Then $v_i > 0$ and $h_i > 0$ give the drop to and the width of step number i, and finally $v_k > 0$ and $h_k \geqslant 0$ is the final drop and the distance to the bottom-right corner. For a chessboard the starting step is $[(8, 8)]$ and the final step is $[(0, 8), (8, 0)]$. Here is a picture of the step shape $[(1, 1), (2, 1), (2, 2), (2, 2), (1, 2)]$:



Set: There is another way to represent your step shape, namely by a sequence

$$SESSESSEESSEESEE$$

of moves, where S denotes a move South, and E a move East. The inital shape is S^8E^8 and the final shape is E^8S^8. Perhaps we should keep this alternative in mind as well.

We still have to prove that the step shape is safe. I agree that both the empty and full tilings are step shapes, but that is not enough. Can we add a domino to a non-full step shape and leave a step shape?

Let's see. Suppose we cannot add a domino to the top step. This will happen if either $v_0 = 0$, when there is no room, or $(v_0, h_0) = (1, 1)$. In the second case only a horizontal domino would fit but it would overhang the remaining steps. But if we can't add a domino to the second step as well, then we also have $(v_1, h_1) = (1, 1)$. Etcetera, until at the end we have either $(v_k, h_k) = (1, 1)$ or $h_k = 0$. So it seems that the one kind of step shape we can't add a domino to is a "real" set of steps with each step having a unit drop and width! We therefore have to show that a real set of steps can't be built by tiling according to our rules.

Oh, but this is obvious, isn't it? Every step in a real set of steps has the same colour, either Black or White. It follows that a real set of steps covers more of one colour than the other. Since each domino covers both a Black and a White square, a real set of steps cannot be covered by dominoes. I suppose Rel would like to see this proof conducted through a formal calculation with dots and relations, and not a variable in sight.

Rel: Yes I would: calculation is the sincerest form of proof. But I don't want to spend time on it as your argument is quite convincing. By my previous reasoning, your proof is enough to show that we can always remove a domino from a non-empty step shape and leave a step shape. In fact, we can see directly that a real set of steps is the only kind we can't remove dominoes from. Anyway, I am itching to represent these placements of dominoes symbolically.

Fun: Before you do, there is a problem. I agree that systematically placing dominoes to maintain a step shape is a good idea, but if we do it in all possible ways we will surely create exactly the same tiling many times over. Somehow we have got to either generate each tiling just once, or find some way of filtering out duplicates. The problem with Data's simple representation is that two different tilings may end up having the same step shape, so the information provided by the step shape alone is not going to be sufficient to identify duplicates.

Set: That's true. But Data's representation is so neat that I would like to stick with it. That means finding a way to generate each possible tiling just once. (*Pause.*)

Dili: While you lot have been chattering away I have been diligently working out the 4×4 case by hand. There are 36 ways to tile a 4×4 board. There are five ways to tile a 4×2 board, so putting two of these side by side gives 25 ways. Another nine are obtained by putting two 2×4 boards above one another – not counting those tilings that fall into the first case. Finally, there are two ways to tile a board that cannot be split into two subtilings side by side or above one another. And $25 + 9 + 2 = 36$. Would you like to see them?

Data: No thanks, but that information will be a useful check when we come to construct the program. Returning to the problem, think of the tree that represents the possible safe tilings. The root of the tree is the empty board and we want all the leaves to be at depth N labelled with the distinct full tilings. Each edge represents the action of one *putDomino*. Now, we have to ensure that each safe partial tiling appears exactly once in the tree, so two distinct tilings mustn't give the same result by adding dominoes. Then we would get a directed graph rather than a tree. More precisely, suppose T_1 and T_2 are distinct tilings, and P_1 and P_2 are two instances of *putDomino*; then we want to forbid $T_1 + P_1 = T_2 + P_2$.

Well, obviously, $P_1 \neq P_2$, otherwise T_1 and T_2 would be the same tiling, and P_1 and P_2 can't be vertical and horizontal dominoes placed on the same step because they wouldn't result in the same step shape. So P_1 and P_2 have to be dominoes placed on different steps. (*Pause.*)

Rel: Can't we solve the problem by only putting dominoes on the highest steps that they fit? The troublesome situation arises because one can add two dominoes P_1 and P_2, placed on different steps, to the same step shape T. Then $T_1 = T + P_2$ and $T_2 = T + P_1$. If we follow my strategy, then one of T_1 and T_2 will never be generated, so the problem doesn't arise.

Fun: No, that doesn't quite work. Consider the "Stonehenge" shape of two vertical dominoes with a horizontal one across the top. If we follow your strategy then, after placing the leftmost vertical domino, we have to put the next domino vertically on top of it, so we can never build Stonehenge!

Set: Interesting. One *can* adopt the stratgey of placing a horizontal domino only at the highest place it will fit, because either a vertical domino can also be placed there, or can be placed on the square in some other part of the tree when a vertical wall of sufficient height has been built on the left. The problem is with vertical dominoes only.

I think I can see the way forward now. There is no problem if P_1 and P_2 have different orientations but are placed on the same step, nor if both P_1 and P_2 are horizontal dominoes. So suppose that P_1 is a vertical domino placed on a higher step than than domino P_2. Let us allow the move P_1 now, but prevent it from happening in some other descendant of the step shape by leaving a little pebble on the step on which P_1 sits, meaning that in other parts of the tree only a horizontal domino can be placed there (if it can, of course).

Data: I see. So we want to represent a step not by a pair of integers but by a triple (p, v, h) in which p is either *Free* or *Pebble*. A step $(Free, v, h)$ means we are free to place either a vertical or horizontal domino on the step (if either or both are possible), but a value $(Pebble, v, h)$ means that only a horizontal domino may be placed there if at all possible.

Here is how we process the step shape $[(p_0, v_0, h_0), \ldots, (p_k, v_k, h_k)]$. We pass over steps (p, v, h) for which $v * h \leqslant 1$ because no dominoes can be placed on them, so suppose we are now considering (p_i, v_i, h_i) where $v_i * h_i > 1$. Suppose first that $p_i = Free$. If $v_i \geqslant 2$ and $h_i \geqslant 2$, then we can place either a vertical or a horizontal domino on the step, so we do both. If $v_i \geqslant 2$ and $h_i = 1$, then we place a vertical domino on the step, and, in searching further down the list, record the fact that we have done so by leaving the step $(Pebble, v_i, h_i)$ behind. In the remaining case $v_i = 1$ and $h_i \geqslant 2$ we simply place a horizontal domino.

Fun: We might as well do this as a program! I propose defining the types

```
> data State  = Free | Pebble
> type Step   = (State,Int,Int)
> type Shape  = [Step]
```

Agreed, *Shape* records only the step shape and not the tiling that leads to it, but our problem is only to count the number of tilings, not to enumerate them. I am going to represent *putDomino* by a function that takes a *Shape* and returns a list of *Shape* without duplicates. Here is the definition, which follows the scheme proposed by Rel, Set and Data:

```
> putDomino :: Shape -> [Shape]
> putDomino [] = []
> putDomino ((Free,v,h):steps)
>    | v>=2 && h>=2   = [horizontal v h steps, vertical v h steps]
>    | v>=2 && h==1   = [vertical v h steps] ++
>                       map (cons (Pebble,v,h)) (putDomino steps)
>    | v==1 && h>=2   = [horizontal v h steps]
>    | otherwise      = map (cons (Free,v,h)) (putDomino steps)

> putDomino ((Pebble,v,h):steps)
>    | v>=1 && h>=2   = [horizontal v h steps]
>    | otherwise      = map (cons (Pebble,v,h)) (putDomino steps)
```

We still have to define `horizontal` and `vertical`, but I think the main structure is clear. The definition of `cons x xs` is, of course, just `x:xs`.

Set: Can't we just program `horizontal` and `vertical` by

```
> horizontal v h steps = (Free,v-1,2):(Free,1,h-2):steps
> vertical v h steps   = (Free,v-2,1):(Free,2,h-1):steps
```

Placing either kind of domino introduces a new step; in the horizontal case the height of v is reduced by 1 and the step has width 2. The new step has height 1 and width $h - 2$. Similar remarks apply to the vertical case. Ah, not quite. These definitions can introduce “virtual” steps with height or width 0.

Rel: I don't think that is a big problem: virtual steps can be merged with real steps above or below them. You want to change the definitions to read

```
> horizontal :: Int -> Int -> Shape -> Shape
> horizontal v h steps
>  | h>2 || null steps = (Free,v-1,2):(Free,1,h-2):steps
>  | otherwise         = (Free,v-1,2):(p,v'+1,h'):steps'
>                        where (p,v',h'):steps' = steps

> vertical :: Int -> Int -> Shape -> Shape
> vertical v h steps
>   | h>1 || null steps = (Free,v-2,1):(Free,2,h-1):steps
>   | otherwise         = (Free,v-2,1):(p,v'+2,h'):steps'
>                         where (p,v',h'):steps' = steps
```

Fun: Well, that doesn't get rid of virtual steps at the head of the list. I see now that the way to do that is to modify my definition of `cons` to read

```
> cons :: Step -> Shape -> Shape
> cons (p,v,h) ((Free,0,h'):steps) = (p,v,h+h'):steps
> cons (p,v,h) steps               = (p,v,h):steps
```

I think we are almost done now. We can define

```
> tilings :: (Int,Int) -> [Shape]
> tilings (m,n) = putnDominos d [[(Free,m,n)]]
>                 where d = (m*n) `div` 2

> putnDominos :: Int -> [Shape] -> [Shape]
> putnDominos 0 = id
> putnDominos d = putnDominos (d-1) . concat . map putDomino

> main = print (length (tilings (8,8)))
```

I agree now that all that relational stuff was useful in the beginning, and it certainly led Data to think about step shapes, but we were always heading for a functional program.

Dili: For square boards you can double the speed of your program by always starting off with a horizontal tile in the bottom-left corner. Then by symmetry you get exactly half the possible tilings, so double the final answer.

Teacher: Thank you all, I think you have collaborated very well indeed. There is another way to program the problem though. Go back to Set's alternative representation of step shapes as sequences of S and E moves. Consider the grammar

$$\begin{array}{rcl} SSE & \rightarrow & ESS \mid SXE \\ SEE & \rightarrow & EES \\ XEE & \rightarrow & EES \end{array}$$

The first rule corresponds to either adding a vertical domino or leaving a pebble on the step. The second rule is interpreted as adding a horizontal domino, and the third rule as replacing the pebble by a horizontal domino. Given a starting value of S^8E^8, the problem is to count the number of leftmost derivations that lead to a final shape E^8S^8. A leftmost derivation corresponds to adding a domino on the highest step. There is a fairly simple closure algorithm to implement the problem but I won't go into details since I suspect that the advantages and disadvantages of the second method (no arithmetic, but a less compact representation of shapes) is counter-balanced by the advantages and disadvantages of yours.

You may like to know that there are 12,988,816 possible tilings of the chessboard. In fact, there is a formula for the number of tilings of an $m \times n$ rectangle. It occurs in *Concrete Mathematics*, by Graham, Knuth and Patashnik, as Bonus problem 51 in Chapter 7. The formula is

$$2^{mn/2} \prod_{\substack{1 \leqslant j \leqslant m \\ 1 \leqslant k \leqslant n}} \left(\cos^2 \frac{j\pi}{m+1} + \cos^2 \frac{k\pi}{n+1} \right)^{1/4}$$

(The exponent 1/4 is missing in early printings, but was corrected in the ninth printing.) According to the authors, the proof is not easy and "really beyond the scope of this book".

Acknowledgements

This pearl was inspired in part by Edsger Dijkstra's note "Nondeterministic construction of an arbitrary witness" (EWD 1229, January 1996). Dijkstra used the S and E representation, although he was concerned only with the problem of generating every tiling, not with the trickier problem of generating every tiling without duplicates.

I showed an earlier draft of this pearl to Don Knuth when he visited Oxford. He made several suggestions for improvement and pointed out the existence of the formula above. If I had known about it, and I certainly should have, then perhaps the pearl would not have been written. Nevertheless, I am deeply grateful to him for taking the time to make comments. I am also grateful to four anonymous referees who made a number of constructive comments, most of which have been incorporated into the dialogue.