Hetris

An exploration of literate programming in Haskell

Ian Lynagh <igloo@earth.li>

Date October 24, 2002

Contents

1 Introduction

Dear reader,

As we start to make our way into the 21st century, so too a new paradigm begins to show it's face. Born in 1992, Knuth's "literate programming" threatens to revolutionise programming, as structured programming did before it. But what is it all about?

A literate program can be compiled not only to executable code but also to documentation. In the same file is both the source code for (part of) the executable program and the source for the technical documentation describing this code. This documentation describes, in English prose, how the code works; for example, it describes the datatype invariants, what the arguments and result of functions are and how the algorithms used work. It is hoped that, by being kept close to the code it describes, the documentation is more likely to be kept up-to-date than stand-alone documentation, leading to a more maintainable package. Historians will remember that this is one of the goals of structured programming; literate programming aims to get even closer to the holy grail of easily maintainable code.

Another advantage of literate programs is that it is harder to make bad design choices when you are explaining the justification for your choice to the reader as you write the code. If you find you cannot justify a decision then the chances are you have made the wrong one and you should think it through again.

However, one cannot write literate programs in a language not designed, or at least adapted, for them. The executable compiler must know how to extract the program source and the documentation compiler must know how to tell the documentation and the source apart. Depending on the system the documentation compiler may strip all or some of the code; any code left in is often pretty-printed—for example, keywords may be typeset in bold.

Haskell is a mixed blessing in this regard. To its advantage the language specification allows for programs to be written in a literate style—program fragments can be written in blocks beginning "`\begin{code}`" and ending "`\end{code}`" which makes it easy to write documentation in any of the TeX family of languages around the code; in general, due to the standard practise in the circles in which Haskell is used, this will tend to be LaTeX.

On the down side, while LaTeX provides a reasonable environment for manually typesetting Haskell code, it is not designed to be an easy target for automatic typesetters. This, combined with the significant white space of Haskell, makes writing tools that produce LaTeX for good looking, validly laid out representations of arbitrary Haskell input difficult, and as such the tools are scarce.

Here we target a tool, HaskellλLaTeX, which makes a reasonable attempt at typesetting the code within a literate program for LaTeX. However, in theory, with a minimum of alteration, any decent tool should yield good results.

Another shortcoming of Haskell is that the user doesn't supply an interface file describing the interface for modules they write but the information is just taken from the export list of the module. This means that an abstract description of a module cannot be checked by the compiler and as such is at risk of falling into a state of disrepair. We still describe the abstract interfaces here, we just need to be careful to remember to keep them in sync with the concrete implementations.

Despite the obstacles in our path we will nevertheless forge onwards. As a vehicle for exploring the literate programming paradigm we will work through the design and implementation of Hetris, a (slightly cut-down) Tetris-like game written in Haskell.

We continue now with a discussion of terminology in Section 2 followed by a general overview in Section 3. Next we consider each module in turn, first looking at the abstract interface where appropriate followed by an example concrete implementation.

2 Terminology

Attempting to conduct an exposition of a system without first agreeing on some terminology with which to refer to its components would likely be a painful affair. Before proceeding we will therefore take a minute to agree on how we will describe the user's view of the game. As mentioned earlier the name of the game is "Hetris"—a name that gives a clue as to the style of game but is hopefully different enough from the names of similar games that it will make us immune to evil lawyers.

The objects that the user controls will be referred to as "pieces". We will also refer to the "active piece" when talking about the piece currently being controlled by the user. Each piece is made up of a number, traditionally 4, "blocks". The rectangular area in which the user manipulates the pieces will be called the "playing area" and is made up of a number of "squares" in a grid. The term "board" will refer to the entire user interface of the application.

There are a couple more terms we will need to talk about, but these will be introduced as we come across them. We are now ready to start thinking about our design.

3 Plan

Before we start writing code, or even designing abstract specifications of modules, it will be helpful to have a feeling for what modules will be in the system and how they will interact. We should also note at this point that we are currently looking at a slightly cut down version of the game: blocks are not coloured but merely exist or do not; squares in the playing area are similarly either occupied or empty; the next piece is not visible to the user; scoring is not implemented. It should be easy, however, to add this functionality to produce a more complete game if required.

There are many ways we may choose to represent a piece—Section 5 has a brief discussion of just some of them. Each has its own set of advantages and disadvantages so we would like to design the rest of the game so that it can be used unchanged regardless of which representation we happen to decide to use. We will therefore separate the code for manipulating the representation of pieces into a separate module and only allow the rest of the game to know about a minimal interface; in other words we will define an abstract module for pieces and how the concrete implementation works will not be relevant to the design or implementation of any other module. In our design of the abstract module we must be careful to think about what we are representing abstractly and not to export functions that only make sense for a particular set of concrete implementations, even if we currently intend to use one such implementation.

A similar story applies to the representations of the board. For much the same reasons we therefore also create a module for this, using the abstraction technique to hide the concrete details.

Another good candidate for an abstract module is the user interface. The rest of the code can be exactly the same regardless of whether the user interface used is based on curses, Gtk or the MS Windows toolkit, so by making the user interface an abstract module we make porting the game to different platforms and toolkits significantly easier.

The three modules we have considered so far are all concerned with carrying out the various bits of mechanism needed. Policy, i.e., which manipulations of the abstract datatypes and user interface should be performed when, has not yet been considered. The game is conceptually very simple so the amount of policy should be very small and there is no obvious way to separate a significant chunk of it off; it therefore seems reasonable that it can all be handled by a single module. As this will be at the top level this can also serve as Main.

In summary, at the top level we will break Hetris down into 4 modules: Main, Pieces, Board and UI. For each of the last three we will agree on an abstract specification of the module and the other three may only use it via its public interface. The Main module will need to know about, i.e., import, each of the other three. Additionally the Board module will need to know about the Pieces module as it will need to find out how to draw pieces in the playing area. None of the other modules should need to know about any of the others, so we can easily see that these dependencies form a directed acyclic graph; this is important as current Haskell implementations deal poorly with cycles in dependencies.

4 Global datatypes

There are many points at which we could begin our design of the program. For example, we could start with the user interface and work down to the logic of the game, working through the modules as we explore deeper; another possibility would be to start at the deep logic and work outwards. However, it seems logical to instead start with what one might call global datatypes.

The problem we are trying to solve is how information is passed from one module to another, either as an argument to a function or as the result of one. Hiding the details with data abstraction is not the effect we want here—we are trying to *share* the actual information, not simply allow other modules to pass it around.

We could, with a minimum of legerdemain, make any of these types "owned" by the most appropriate module. However, we would not be being honest with ourselves if we did this—these types really belong, conceptually speaking, to the channels by which modules communicate.

There are four types that come into this category; as they just require definitions it does not make seem worth the hassle to split them off into four tiny modules, so instead we bundle them together into this single module. The module export information is shown below, followed by an explanation of each of the four types.

```
module Data (Delay, Vector, Event (..), Change (..))
    where
```

4.1 Delay

At the very heart of the game is a clock ticking away. On each tick either the active piece is moved down or, if this is not possible, its component blocks are added to the board and a new piece is made active. The time between these clock ticks is a policy decision—as far as the mechanism modules are concerned it need not even necessarily be constant—so it should be set by the Main module. However, the user interface will need to stop waiting for input after this time has elapsed, so it needs to know the value too. We therefore make it a globally known type. We will allow modules using the type to assume that it is an instance of the Integral class, counting the time until the next tick in milliseconds; this means the Int type should be sufficiently wide to hold all the values we care about.

```
type Delay = Int
```

4.2 Vector

We need to talk about positions, widths and heights on and of the playing area all over the place. For example, the policy module Main will need to agree on a size, i.e., width and height, that the mechanism module UI can display.

For an example of when positions on the playing area need to be passed around consider what happens when the playing area is altered and the user interface needs to be updated accordingly. We could always pass around lists of lists, say, to describe the current state of the board to the user interface, but it is more efficient to pass around a list of changes which describe a change of a particular square, i.e., a position, on the playing area.

We can use the same type for talking about both positions and the width and height of the board, so we would like a name that conveys the impression that its value may be either a length or position; for lack of a better word we choose Vector. Again it makes sense if we allow ourselves to assume that the type is an instance of class Integral, and again Int should be easily wide enough for our purposes.

```
type Vector = Int
```

A value of 0 refers to the the uppermost or leftmost cell as appropriate if the Vector is referring to a position.

4.3 Event

The user interface will need to communicate with the policy module to inform it of events that have happened. We don't want to pass low level things like what key was pressed around, not least because this precludes interfaces that don't work in this way, e.g., mouse driven interfaces. Instead we use an abstract datatype Event where each constructor corresponds to one of the possible events that can occur.

```
data Event = RotL
           | RotR
           | MDown
           | MLeft
           | MRight
           | Drop
           | Tick
           | Quit
           | None
    deriving Eq
```

We derive Eq as it will allow us to use slightly simpler code later on.

The meaning of each constructor is as follows:

RotL, RotR, MDown, MLeft, MRight
: These correspond to requests to rotate the active piece left or right or move it down, left or right respectively.

Drop
: Corresponds to a request to drop the piece as far down as possible, i.e., equivalent to multiple MDown events.

Tick
: This event occurs when the time until the next clock tick hits zero.

Quit
: The user has requested the program to quit.

None
: This is not a real event; it will be created when, for example, a user presses a key that is not bound to any real event. Its purpose is simply to make things more convenient for us in some circumstances.

4.4 Change

As we briefly mentioned earlier, changes in the playing area need to be sent to the user interface module. In this simplified specification of the game there are three things we will want to be able to do. First we may want to turn a given square on the board on. Second we may want to turn a square off. Finally, after deleting a complete row we may want to pause briefly for the user to be able to see what has happened; the amount of time we should wait is a look-and-feel issue, so we leave it up to the user interface to decide. These map fairly directly to an abstract datatype as shown:

```
data Change = On Vector Vector
            | Off Vector Vector
            | Delay
```

The two Vectors used by the On and Off constructors are x and y coordinates respectively.

5 Pieces: Abstract specification

We now need to think carefully about what a piece is and how we can represent it abstractly. When a piece is on the board it takes up a number of squares. Rather than thinking about it occupying those square it is easier to think of it being at a particular square on the board—we will call this the "key point"—and also occupying a number of squares around it. The position of a piece is not part of its internal state, so a piece can be moved around just by altering this position without having to go via this module. Note that there is no need for the square at the position of the piece to actually have a block on it, although using 4 connected blocks to make a piece it is not possible to make one where there is not a sensible block to put on the key point.

Now we know what we are modelling we will continue by working through the design of an abstract interface to it. From this interface we will need to be able to extract all the information about pieces we will need in other parts of the game.

The Pieces module exports the type Piece, but not its constructors. This allows functions dealing with pieces to give a type signature but means they can't alter the internal state of a piece; this enforces the abstraction required. Also exported are various functions for manipulating pieces; we will go into more detail below.

module Pieces (Piece, *rot_left*, *rot_right*, *blocks*, *extent_down*, *extent_left*, *extent_right*, *pieces*)
where

As we will see, many of the functions use types defined in the Data module. Any implementation will therefore need to import it.

import Data

As mentioned earlier, position of a piece on the board need not be part of the piece's state. However, rotation is more complex.

If rotation was just rotation about the key point then we would have more freedom and be able to produce nicer solutions; however, some pieces are not rotated in this way in Tetris and this is a detail we would rather match faithfully. Rotating a piece left or right is therefore really just taking the next or previous form of the piece.

If rotation was done in a mathematically pure way then we might be tempted to store the current rotation outside of the piece along with the current position. However, in the mathematically more clumsy, but more elegant looking, traditional way we need a function that, given a piece, returns the piece rotated left or right.

rot_left, *rot_right* :: Piece → Piece

Now that we know how we are going to move and rotate pieces we need to be able to draw them in the correct place at the correct rotation on the user interface. We want to export a function that enable this information to be extracted while making as little assumption as possible about the internal structure of the Pieces type. No matter what the definition of the type, it is representing a list of blocks that are part of the piece; therefore it seems logical that it is this list that is available.

We are still left with a choice, however. We could just return a list of positions of the blocks relative to the key position of the piece or we could take an x and y coordinate representing the key point and return the absolute positions of the blocks on the board. Neither choice jumps out as being obviously correct so we choose the first alternative in the belief it will result in slightly less work later on. The *blocks* function will return the list of relative positions of blocks within a piece you pass it.

$blocks$ :: Piece $\rightarrow$ [(Vector, Vector)]

When we try to move pieces left or right or rotate them, as well as making sure they aren't going to end up on top of other block on the board we also need to make sure they don't go off the sides or bottom. While we could work this out by finding the extreme left, right and bottom blocks in the output of *blocks*, the concrete implementation may be able to do it more efficiently; in particular it may store the extreme points for each piece. We therefore provide functions for finding the extremities of a piece. In each case a positive number means that the piece extends by that amount from the key point in the relevant direction.

This is a good example of a case where OOP's inheritance and overriding would be useful—we could define an implementation of the extent functions using the output of *blocks* which concrete implementations could override if they could do it more efficiently.

$extent\_down$, $extent\_left$, $extent\_right$ :: Piece $\rightarrow$ Vector

Last, but by no means least, we need to be able to get new pieces. The Pieces module just exports a list of all of the pieces; how the next one is chosen is left to the user of the module.

$pieces$ :: [Piece]

It is important to note that, while the external interface has been defined, there is still a lot of flexibility left for the internal representation. The blocks could be stored in a binary search tree, a BSP tree, an array, a list of lists, encoded as a number in some base and so on. In each case we may choose to store the extents or derive them from the blocks representation. Some choices will result in more elegant code, some are purer solutions than others, some will be more time efficient, some will be more space efficient; what is important is that we do not need to concern ourselves with these decisions now.

6 Pieces: Concrete implementation

The header is, as it always is for a concrete implementation of an abstract module, the same as that of the abstract specification:

```
module Pieces (Piece, blocks, extent_down, extent_left, extent_right, rot_left,
               rot_right, pieces)
    where

import Data
```

Tetris is not a particularly challenging task for today's CPUs, and the amount of RAM required is not likely to cause a problem on a modern machine.

Our motivation here then is to pick a representation such that the implementation can be easily understood.

At the core of a representation is a list of coordinates of blocks relative to the key square of the piece. We store an infinite list of these corresponding to the blocks used by a piece after successive left rotations.

```
data Piece = Piece [[(Vector, Vector)]]
```

This makes the implementation of *blocks* simple—we just return the first list.

```
blocks :: Piece → [(Vector, Vector)]
blocks (Piece xss) = head xss
```

The code to calculate the maximum extents is based on the output of *blocks* in the way hinted at in Section 5.

```
extent_down :: Piece → Vector
extent_down p = maximum $ map snd $ blocks p

extent_left :: Piece → Vector
extent_left p = negate $ minimum $ map fst $ blocks p

extent_right :: Piece → Vector
extent_right p = maximum $ map fst $ blocks p
```

To rotate a piece left we just remove the first element of the list; the correctness of this follows from the definition of the list above. Rotating right is the same as rotating left 3 times so we drop the first 3 elements of the list.

```
rot_left :: Piece → Piece
rot_left (Piece xss) = Piece (tail xss)

rot_right :: Piece → Piece
rot_right (Piece xss) = Piece (drop 3 xss)
```

The list of pieces is just that—the second part of the name is intended to be descriptive of the shape of the piece, but this is more successful in some cases than others.

```
pieces :: [Piece]
pieces = [piece_I, piece_L, piece_J, piece_T, piece_2, piece_5, piece_O]
```

We conclude with the actual definitions of the pieces.

```
piece_I :: Piece
piece_I = Piece (cycle [p1, p2])
    where p1 = [(0, −1), (0, 0), (0, 1), (0, 2)]
          p2 = [(−1, 0), (0, 0), (1, 0), (2, 0)]

piece_L :: Piece
piece_L = Piece (cycle [p1, p2, p3, p4])
    where p1 = [(0, −1), (0, 0), (0, 1), (1, 1)]
          p2 = [(−1, 0), (0, 0), (1, 0), (1, −1)]
          p3 = [(−1, −1), (0, −1), (0, 0), (0, 1)]
          p4 = [(−1, 1), (−1, 0), (0, 0), (1, 0)]
```

```
piece_J :: Piece
piece_J = Piece (cycle [p1, p2, p3, p4])
    where p1 = [(0, −1), (0, 0), (0, 1), (−1, 1)]
          p2 = [(−1, 0), (0, 0), (1, 0), (1, 1)]
          p3 = [(1, −1), (0, −1), (0, 0), (0, 1)]
          p4 = [(−1, −1), (−1, 0), (0, 0), (1, 0)]

piece_T :: Piece
piece_T = Piece (cycle [p1, p2, p3, p4])
    where p1 = [(−1, 0), (0, 0), (1, 0), (0, 1)]
          p2 = [(0, −1), (0, 0), (0, 1), (1, 0)]
          p3 = [(−1, 0), (0, 0), (1, 0), (0, −1)]
          p4 = [(0, −1), (0, 0), (0, 1), (−1, 0)]

piece_2 :: Piece
piece_2 = Piece (cycle [p1, p2])
    where p1 = [(1, 0), (0, 0), (0, −1), (−1, −1)]
          p2 = [(0, 0), (0, −1), (−1, 0), (−1, 1)]

piece_5 :: Piece
piece_5 = Piece (cycle [p1, p2])
    where p1 = [(−1, 0), (0, 0), (0, −1), (1, −1)]
          p2 = [(0, 0), (0, −1), (1, 0), (1, 1)]

piece_O :: Piece
piece_O = Piece (repeat [(0, 0), (0, 1), (1, 0), (1, 1)])
```

7 Board: Abstract specification

Back in abstract mode now, we need to find a suitable abstract representation for the board. The board needs to keep track of the squares occupied by blocks of pieces that have come to rest, the active piece and its location in the playing area.

As with Pieces, we export the type Board and various functions that manipulate boards.

```
module Board (Board, create_board, get_changes, can_down, next_piece)
    where
```

As we will see, functions in this module will use types and functions from Data and Pieces. Any implementation will therefore need to import them.

```
import Data
import Pieces
```

At the beginning of the game we will need to create a new board. For this we define a function *create_board* which, when passed a width followed by a height followed by a piece, will return a board of the requested dimensions and the list of changes needed to go from a blank board to one with the requested piece visible. It is positioned such that it's key point is at the middle of the top row.

One may argue that the piece shouldn't be passed at the time of the creation of the board, but a separate function should add it later. However, this would just complicate the internal state of concrete implementations of the module while giving no real benefit.

create_board :: Vector → Vector → Piece → (Board, [Change])

Most of the time the board will be handling events. The *get_changes* function takes a board followed by an event and returns the board as it is after this event has been performed and the list of changes the user interface needs to perform to get to this state.

get_changes :: Board → Event → (Board, [Change])

We want the next piece to be chosen by a policy module rather than this mechanism module. This means that we need to be able to tell whether the active piece can be moved down when we get a Tick event. The *can_down* function returns True if the active piece on the Board passed to it can move down one row without lying on top of other blocks or sticking out of the bottom of the board.

can_down :: Board → Bool

Once we have decided that a new piece should be added we need a function that augments a board with one. It should first break the current active piece down into its constituent blocks and fix them in place on the board. Then it should remove any completed rows, from the bottom up, before finally adding the new active piece.

For this we need a function similar to *get_changes*, but this one must take a Piece rather than an Event. The result type is also slightly different—a new Board will only be returned if the new active piece fits in the board without overlapping with blocks left on the board from previous; otherwise Nothing is returned instead. In either case a list of Changes is returned—the player will want to see their rows disappear and the new piece appear even if it signals the end of the game!

next_piece :: Board → Piece → (Maybe Board, [Change])

8 Board: Concrete implementation

Again, and as always, the header is the same as that of the abstract specification:

```
module Board (Board, create_board, get_changes, can_down, next_piece)
    where

import Data
import Pieces
```

The concrete implementation we will use here will be based around Haskell's Array type. This will allow us to write clear, simple code to get and overwrite the state of a board. We thus also need to include the Array module.

```
import Array
```

In this simplified variant of the game each square of the board either contains a block or it doesn't; therefore we can use a Bool to represent whether a square has a block in it or not. The playing area is then represented by an array of these, indexed by 2-d coordinates of Vectors. The actual Board type also keeps track of the current active piece as well as its coordinates. In both cases the x coordinate is the first Vector and $(0, 0)$ is the upper left corner as usual.

```
type Block = Bool
type PlayingArea = Array (Vector, Vector) Block
data Board = Board PlayingArea Piece Vector Vector
```

We will start the implementation with some utility functions. First we provide an augmented version of the *blocks* function exported by the Pieces module. This takes the relative block positions of the piece as returned by *blocks* and adds them to a position which it also takes as arguments; this gives a list of absolute positions of blocks. It then filters the list to extract only the coordinates that are in the range of the playing area array passed; this means that if a piece is not entirely within the playing area, and remember that when a piece first appears it may legitimately be off the top of the playing area, then we won't ask the user interface to draw blocks outside of the area it has allocated for the playing area.

Next we define a function *alter_blocks* that builds on this. It takes the same arguments and additionally a (curried) function that takes an x and y coordinate and returns a Change; it then applies this function to all of the coordinate pairs to produce a list of Changes. It is no coincidence that the On and Off constructors have this type, and we further define functions *on* and *off* which can be used to create the list of changes needed to turn all the blocks of a given piece at a given location on and off respectively.

```
restricted_blocks :: PlayingArea → Piece → Vector → Vector → [(Vector, Vector)]
restricted_blocks a p x y = filter (inRange (bounds a))
                                [(x + off_x, y + off_y) | (off_x, off_y) ← blocks p]

alter_blocks :: (Vector → Vector → Change) → PlayingArea → Piece → Vector → Vector → [Change]
alter_blocks f a p x y = map (uncurry f) (restricted_blocks a p x y)

on :: PlayingArea → Piece → Vector → Vector → [Change]
on = alter_blocks On

off :: PlayingArea → Piece → Vector → Vector → [Change]
off = alter_blocks Off
```

A playing area of width w and height h has squares indexed from $(0, 0)$ up to $(w-1, h-1)$, and initially none of these contain a block. The *create_board* function creates an array with the appropriate range of indices all containing False. The first component of the result tuple, the Board, is then built from this Array and the piece that was passed, which is placed at the middle of the top row as required by the abstract specification.

The second component of the result tuple, the list of Changes the user interface will have to perform, is the result of turning on all of the blocks used by the piece in its initial position.

```
create_board :: Vector -> Vector -> Piece -> (Board, [Change])
create_board width height p = (b, on a p (width `div` 2) 0)
    where a = listArray ((0, 0), (width - 1, height - 1)) (repeat False)
          b = Board a p (width `div` 2) 0
```

The *get_changes* function performs different tasks depending on what Event it is passed. Probably the simplest cases are those where the active piece is just moved one square down, left or right. In these cases we first check, using functions we will define shortly, that we can move in the specified direction. If we can then the changes needed in the user interface are to turn off all the blocks of the piece where it currently is and then turn on all the blocks where it moves to. The new Board returned is the same as the one passed but with the coordinates of the piece suitably updated. If a Tick event gets this far then it must correspond to the active piece moving down as it would have been caught earlier if it signals the next piece.

```
get_changes :: Board -> Event -> (Board, [Change])
get_changes b@(Board a p x y) MDown
    | can_down b = (Board a p x (y + 1), off a p x y ++ on a p x (y + 1))
get_changes b@(Board a p x y) MLeft
    | can_left b = (Board a p (x - 1) y, off a p x y ++ on a p (x - 1) y)
get_changes b@(Board a p x y) MRight
    | can_right b = (Board a p (x + 1) y, off a p x y ++ on a p (x + 1) y)
get_changes b Tick                     = get_changes b MDown
```

We can handle Drop by, if we can move the piece down, first acting as if we had been given a MDown event. We then take the Board this returns and recursively consider what happens if we deal with a Drop event on it. We return the Board returned and both lists of changes concatenated in order.

```
get_changes b Drop
    | can_down b = (b'', cs1 ++ cs2)
    where (b', cs1) = get_changes b MDown
          (b'', cs2) = get_changes b' Drop
```

The code for rotating both left and right is very similar so is best handled by a generic function; we therefore define *rotate*, as shown shortly, which we pass a function which manipulates a piece in the appropriate way, i.e., rotates it left or right, and the Board we were passed.

```
get_changes b RotL = rotate rot_left b
get_changes b RotR = rotate rot_right b
```

We have handled every case where the board change or changes need to be generated for the user interface. Therefore for any other event we just return the same board we were passed and the empty list of changes.

```
get_changes b _ = (b, [])
```

We now have some promises to fulfil; let us start with the definition of *rotate*. We pass it a function that manipulates a piece in the desired way followed by a Board. If the Piece in the Board when acted upon by the rotate function 'fits', as defined by a function we, if you'll forgive the nested promise, will define in just a few lines, then we return the board with the piece in its new orientation and the changes list turns off the blocks used by the piece in its previous position and turns on those corresponding to its new position; all in all it is very similar to the movement events except the piece also changes.

If it doesn't fit then we do nothing, as *get_changes* does.

```
rotate :: (Piece → Piece) → Board → (Board, [Change])
rotate f (Board a p x y)
    | fits b' = (b', off a p x y ++ on a p' x y)
    where p' = f p
          b' = Board a p' x y
rotate _ b                = (b, [])
```

As promised, we continue with a definition of *fits*. We take a Board and, in essence, return a Bool indicating whether or not the Piece in the Board 'fits'; that is to say, we return True if it doesn't lie on top of any blocks already in the playing area and it doesn't stick out of the top, left or right of the playing area. We allow it to stick out of the top. The astute reader will have noticed that we need to make a yet deeper nested promise, this time to define *not_collides* that checks that the Piece in a Board doesn't overlap any blocks in the Board's playing area.

```
fits :: Board → Bool
fits b@(Board a p x y) = not_collides b && y + extent_down p ≤
                         (snd $ snd $ bounds a) && x − extent_left p ≥
                         (fst $ fst $ bounds a) && x + extent_right p ≤
                         (fst $ snd $ bounds a)
```

For this latest promise we take the blocks occupied by the Piece, restricted to the playing area, and take the value of the array at each coordinate. If any of them are True then there is a collision so we return False; otherwise we return True.

```
not_collides :: Board → Bool
not_collides (Board a p x y) = not $ or $ map (a !) $ restricted_blocks a p x y
```

Having completed this chain of promises we still have one outstanding—to define the functions to test whether the active piece can be moved down, left and right. If you've been keeping a close eye on things then you'll have noticed that one of these is exactly what is exported to decide which is the applicable behaviour upon getting a Tick event.

```
can_down, can_left, can_right :: Board → Bool

can_down (Board a p x y) = fits (Board a p x (y + 1))

can_left (Board a p x y) = fits (Board a p (x − 1) y)

can_right (Board a p x y) = fits (Board a p (x + 1) y)
```

This leaves one exported function remaining—the one that deals with one piece coming to rest, completed lines being removed and the new active piece being added in.

The first step is to update the playing area with the blocks of the piece that is coming to rest. To do this we use the (//) operator, zipping the absolute position of the blocks within the playing area with an infinite list of Trues to produce the list of new associations to add.

Second we we use the *drop_complete_lines*, that (you guessed it!) we will define next, to produce a tuple of the array after complete lines have been removed and the changes the user interface will need to show the user this.

For the returned board we take this final array and put the piece on with its key point at the initial square. If it doesn't overlap with any existing blocks in this position then we wrap it with Just and return it; otherwise we return Nothing. The second component of the result, the list of changes, is composed of the changes needed to drop the complete lines followed by the changes needed to turn on the blocks of the new active piece.

$$
\begin{aligned}
&\mathit{next\_piece} :: \mathsf{Board} \rightarrow \mathsf{Piece} \rightarrow (\mathsf{Maybe\ Board}, [\mathsf{Change}])\\
&\mathit{next\_piece}\ (\mathsf{Board}\ a\ p\ x\ y)\ p' = \begin{pmatrix} \textbf{if}\ \mathit{not\_collides}\ b' & , cs \mathbin{+\!\!+} \mathit{on}\ a''\ p'\ x'\ y' \\ \quad \textbf{then}\ \mathsf{Just}\ b' & \\ \quad \textbf{else}\ \ \mathsf{Nothing} & \end{pmatrix}\\
&\quad \textbf{where}\ a' = a\ //\ \mathit{zip}\ (\mathit{restricted\_blocks}\ a\ p\ x\ y)\ (\mathit{repeat}\ \mathsf{True})\\
&\qquad\quad (a'', cs) = \mathit{drop\_complete\_lines}\ a'\\
&\qquad\quad b' = \mathsf{Board}\ a''\ p'\ x'\ y'\\
&\qquad\quad ((\mathit{xmin}, \mathit{ymin}), (\mathit{xmax}, \_)) = \mathit{bounds}\ a\\
&\qquad\quad x' = (\mathit{xmin} + \mathit{xmax})\ `\mathit{div}`\ 2\\
&\qquad\quad y' = \mathit{ymin}
\end{aligned}
$$

All that is left is for us to define *drop_complete_lines*. This is really just a header function for *drop_complete_lines'* which does the hard work; all we do here is to extract the range of x values we will have to check for each row and the list of y values corresponding to rows to be checked. We reverse the second list as we want to drop rows from the bottom up.

$$
\begin{aligned}
&\mathit{drop\_complete\_lines} :: \mathsf{PlayingArea} \rightarrow (\mathsf{PlayingArea}, [\mathsf{Change}])\\
&\mathit{drop\_complete\_lines}\ a = \mathit{drop\_complete\_lines'}\ \mathit{xs}\ (\mathit{reverse}\ \mathit{ys})\ a\\
&\quad \textbf{where}\ ((\mathit{xmin}, \mathit{ymin}), (\mathit{xmax}, \mathit{ymax})) = \mathit{bounds}\ a\\
&\qquad\quad \mathit{xs} = \mathit{range}\ (\mathit{xmin}, \mathit{xmax})\\
&\qquad\quad \mathit{ys} = \mathit{range}\ (\mathit{ymin}, \mathit{ymax})
\end{aligned}
$$

Then *drop_complete_lines'* recurses down the list of rows to be checked. If the list is empty then trivially the array is unchanged and the user interface need perform no changes. Otherwise we first consider the first row in the list. If for each x value the array has True for this y value then we need to remove this row; otherwise we continue with a recursive call on the rest of the list.

To remove row y we first turn off all of the squares on that row and then pause for the user to appreciate what has happened. Then we move all the rows above that row down a row and make the top row empty, passing the active playing area along and collecting up the changes lists. Then there is another delay before finally we recursively call ourselves; note that we need to check row y again as it now contains what was the row above.

```
drop_complete_lines' :: [Vector] → [Vector] → PlayingArea → (PlayingArea, [Change])
drop_complete_lines' _ [] a       = (a, [])
drop_complete_lines' xs (y:ys) a = if and [a ! (x, y) | x ← xs]
                                   then (a"', cs1 ++ [Delay] ++ cs2 ++ cs3 ++ [Delay] ++ cs4)
                                   else  drop_complete_lines' xs ys a
    where cs1 = [Off x y | x ← xs]
          (a', cs2) = move_down a xs ys
          (a", cs3) = empty_top_row a' xs
          (a"', cs4) = drop_complete_lines' xs (y:ys) a"
```

There are two helper functions left undefined; the first, *move_down*, is intended to move a region of squares down one row. The list of changes needed for this is build by considering each square in the region and generating an on or off event depending on whether or not it has a block in it; the event acts on the square below, i.e., with y value on greater, though. The array is overriden in an analogous way.

```
move_down :: PlayingArea → [Vector] → [Vector] → (PlayingArea, [Change])
move_down a xs ys = (a', cs)
    where cs = [ (if a ! (x, y)
                  then On
                  else  Off) x (y + 1) | y ← ys, x ← xs]
          a' = a // [((x, y + 1), a ! (x, y)) | y ← ys, x ← xs]
```

The final function to define simple sets the top row to be empty of all blocks. Thus for each x value it sets the corresponding square in the top row to be off and updates the array similarly.

```
empty_top_row :: PlayingArea → [Vector] → (PlayingArea, [Change])
empty_top_row a xs = (a', cs)
    where cs = [Off x 0 | x ← xs]
          a' = a // [((x, 0), False) | x ← xs]
```

9 User interface: Abstract specification

Back in abstract description mode now, let us consider the user interface. For obvious reasons the result type of every function that deals with the user interface is gong to be in the IO monad.

There are no types that need to be defined by this module, but we do use some of the types defined in the Data module, so that needs to be imported. It will come as no surprise to you that various functions are exported, and we will now go into them in more detail.

```
module UI (init_ui, shutdown_ui, make_board, get_event, do_changes)
    where

import Data
```

First we will need to export a function which initialises the user interface. Similarly we need a function which tells the user interface to clean up after itself and shut down cleanly. Calls to *init_ui* and *shutdown_ui* must strictly alternate with a call to *init_ui* first. Calls to any other user interface functions must be after a call to *init_ui* and before the next call to *shutdown_ui*.

It is the policy module that should get to decide how large the board should be. However, the user interface is unlikely to be able to show a board of unlimited size. The initialisation function therefore returns the maximum size of board it can display as a tuple containing the width and height.

$init\_ui$ :: IO (Vector, Vector)

$shutdown\_ui$ :: IO ()

After initialising the user interface, the next thing to do is to create a board. The *make_board* function takes a width followed by a height and creates a board if that size; a call to this function must not pass values larger than those returned by the last call to *init_ui*.

$make\_board$ :: Vector $\rightarrow$ Vector $\rightarrow$ IO ()

The *get_event* function takes a (positive) Delay and waits up to that amount of time for the user to do something; in general it is likely that this will be pressing a key. If the user does something then it returns the corresponding Event. Otherwise Tick is the Event returned. The elapsed time while waiting for an event is also returned.

get_event :: Delay $\rightarrow$ IO (Event, Delay)

The final task the user interface must be able to perform is to apply a list of changes. The end result must be the same as applying each change in the order they were given.

$do\_changes$ :: [Change] $\rightarrow$ IO ()

10 The user interface module (UI)

The user interface is the most obvious case where multiple implementations are sensible. User interfaces are one of the less well developed areas in the Haskell community, but the new FFI allows us to easily and portably define a Haskell interface to curses. We don't show the actual interface here, but it should be easy to understand from the C documentation; the Curses module exports it.

While as far as the game is concerned the user interface is a single idea, the input and output aspects are really completely separate in curses. We therefore split the functionality off into separate modules which are both imported by UI. The initialisation and shutdown functions come under neither category so we leave them in the main module.

module UI (*init_ui*, *shutdown_ui*, *make_board*, *get_event*, *do_changes*)
    where

import Data
import Curses
import Input
import Output

The actual initialisation work is standard curses startup stuff: we initialise the curses system, enter cbreak mode (disable line buffering etc) and turn off echoing in common with the vast majority of curses programs. We also enable the keypad mode so that we can use the arrow keys to control the movement of the active piece. Finally we ask for the cursor to be invisible.

The maximum board size depends on how the board is drawn, and the logic for that is in the Output module. We therefore have that module export a function that gives this maximum size and we return that value.

```
init_ui :: IO (Vector, Vector)
init_ui = do w ← initscr
             errI $ cbreak
             errI $ noecho
             errI $ keypad w cTRUE
             errI $ curs_set 0
             max_size
```

Shutting the interface down is rather simpler—we just call the curses shutdown function.

```
shutdown_ui :: IO ()
shutdown_ui = do _ ← endwin
                 return ()
```

11 Input: Concrete curses implementation

For the input side of the user interface we will need a number of modules. It will come as no surprise that we need to import the Curses and Data modules. In order to give types to all of the functions we will need to be able to refer to the C types returned by the FFI functions, so CTypes also needs to be imported. The Time module is used to measure the elapsed time and the Char module is used to convert between characters and Ints.

There is only one input function exported by the UI abstract module, namely *get_event*, so that is all we export here.

```
module Input (get_event)
   where

import Curses
import Data
import CTypes
import Time
import Char
```

The specification does not allow the delay to be less than or equal to zero, so we give an error if this is the case. Note that we check the value of the delay after it has been converted to a CInt as the conversion process may not preserve the value.

Otherwise we set the timeout to what was requested, make a note of the current time, and call *getch* to wait for a key to be pressed. When this happens, or it times out, we record the time again. Finally we return a tuple with the event corresponding to the key pressed (which will be *cERR* if a timeout occurred) and the time elapsed between the two times recorded; for both components an additional function is used—these are described below.

```
get_event :: Delay → IO (Event, Delay)
get_event delay
    | delay' ≤ 0 = error "Input.get_event: delay <= 0"
    | otherwise  = do timeout delay'
                      start ← getClockTime
                      c ← getch
                      end ← getClockTime
                      return (key_to_event c, elapsed_time start end)
    where delay' = fromIntegral delay
```

Some keys, which are represented as CInts by the Curses library, have events tupled with them in a lookup list *key_events* suitable for use with *lookup*. If the key we are passed is not mapped to anything then we return the event None instead.

```
key_to_event :: CInt → Event
key_to_event k = maybe None id (lookup k key_events)
```

The construction of the lookup list is uninteresting. We just built it piece by piece and concatenate the pieces together.

```
key_events :: [(CInt, Event)]
key_events = [(cERR, Tick)] ++ movement ++ rotations ++ control
    where to_event e = map (λ c → (c, e))
          conv_char = fromIntegral ∘ ord
          movement = lefts ++ rights ++ downs ++ drops
          rotations = rot_lefts ++ rot_rights
          control = quits
          lefts = to_event MLeft [conv_char 'j', fromIntegral cKEY_LEFT]
          rights = to_event MRight
                     [conv_char 'l', fromIntegral cKEY_RIGHT]
          downs = to_event MDown
                     [conv_char 'k', fromIntegral cKEY_DOWN]
          drops = to_event Drop [conv_char ' ']
          rot_lefts = to_event RotR [conv_char 'u', fromIntegral cKEY_UP]
          rot_rights = to_event RotR [conv_char 'i']
          quits = to_event Quit [conv_char 'q']
```

Sadly Haskell doesn't provide an easy way to measure the time between two points in time. The best we can get from the standard libraries is a TimeDiff. We assume that less than a minute passes between the two times—a reasonable assumption in this context!

```
elapsed_time :: ClockTime → ClockTime → Delay
elapsed_time start end = t `max` 0
    where t = case diffClockTimes end start of
                (TimeDiff 0 0 0 0 0 secs psecs) → let secs' = 1000 × fromIntegral secs
                                                      psecs' = fromIntegral
                                                                 (psecs `div` 1000000000)
                                                  in secs' + psecs'
                td → error ("Input.elapsed_time: " ++ show td)
```

12 Output: Concrete curses implementation

If we were to use a single character for each square of the playing area, and indeed the rest of the board, then the squares would be significantly taller than they are wide. To counteract this we will treat each pair of characters on a line as a single entity as far as drawing the board is concerned; this will give them a roughly square appearance.

The Output module is responsible for the other two functions exported by the UI module, and also needs to provide the concrete implementation of this module with a function giving the largest size the interface can accommodate.

Unsurprisingly we import the Data module, as well as the Curses module to provide types for the functions. We also use the Char module to convert characters into their numeric ASCII values.

XXX Storable

```
module Output (max_size, make_board, do_changes)
    where

import Data
import Curses
import Char
import Storable
```

Continuing our practise of separating constants out of the main code, we define values representing the minimum amount of white space we require around the playing area. A border of one empty character around one solid character, a total width of 2, all the way round is quite aesthetically pleasing so we go with that.

```
border_width, border_height :: Vector

border_width = 2

border_height = 2
```

Our first commitment to the outside world is to provide a function that returns the maximum size of user interface we can draw. We first use the curses *getmaxyx* function to find the height and width of the window passed (XXX while this can't use stdscr is getmaxyx really width and height?). As we are treating characters in pairs along the horizontal axis we need to divide this width by 2 (rounding down), and then we use *fromIntegral* to convert the coordinates into CInts. Finally we need to subtract twice the appropriate border sizes from the dimensions, once for the top/left and again for the bottom/right. Note that width is the first component of the result.

```
max_size :: IO (Vector, Vector)
max_size = do w <- peek stdscr
              (height, width) <- getmaxyx w
              let width' = fromIntegral (width `div` 2)
                  height' = fromIntegral height
              return (width' - 2 * border_width, height' - 2 * border_height)
```

Our second commitment is to provide a function to allow the user of the module to draw a new board. We make a list of screen coordinates comprising the border[1] and convert 'X' to a ChType (XXX should be ACS_BLOCK). then we use the *write* function to write an 'X' to each of these coordinates.

```
make_board :: Vector → Vector → IO ()
make_board width height = do let c = fromIntegral $ ord 'X'
                             mapM_ (flip (uncurry write) c) border
    where border = [(x, border_height − 1) | x ← xs] ++
                   [(x, border_height + height) | x ← xs] ++
                   [(border_width − 1, y) | y ← ys] ++
                   [(border_width + width, y) | y ← ys]
          xs = [border_width − 1 .. border_width + width]
          ys = [border_height .. border_height − 1 + height]
```

Our final external commitment is a function that performs a list of changes to update the screen. To do this we use *mapM_* with a function that performs a single change and finish up by calling the curses *refresh* function to make sure the changes are reflected on the screen.

```
do_changes :: [Change] → IO ()
do_changes cs = do mapM_ do_change cs
                   errI refresh
                   return ()
```

Performing a single change is a simple case analysis. To turn a square on we paint a '#' in it; to turn a square off we paint a blank space in it. For a delay we set the timeout to 500ms and call *getch*. This could be cut short by the user pressing a key, but the effort required to work around this cannot be justified—think of it as a feature.

```
do_change :: Change → IO ()
do_change (On x y)  = paint_square_c x y '#'
do_change (Off x y) = paint_square_c x y ' '
do_change Delay     = do timeout 500
                         _ ← getch
                         return ()
```

We still have a couple of functions left to tidy up. First let us look at *write*. This takes an x and y coordinate and a ChType and writes it at the corresponding pair of screen coordinates.

```
write :: Vector → Vector → ChType → IO ()
write x y c = do mvaddch y' x' c
                 mvaddch y' (x' + 1) c
    where y' = fromIntegral y
          x' = fromIntegral $ 2 × x
```

The *paint_square_c* function is similar; it writes a Char at given playing area coordinates. The hard work is done by *paint_square*, with the harder work being done by *write*.

```
paint_square_c :: Vector → Vector → Char → IO ()
paint_square_c x y c = paint_square x y (fromIntegral $ ord c)
```

[1] Technically they are not screen coordinates; the coordinate (x, y) maps to both $(2x, y)$ and $(2x + 1, y)$ in real screen coordinates

$paint\_square$ :: Vector $\rightarrow$ Vector $\rightarrow$ ChType $\rightarrow$ IO ()
$paint\_square\ x\ y\ c = write\ (x + border\_width)\ (y + border\_height)\ c$

13 The heart of the game

All the modules dealing with the various pieces of the game are now complete leaving only the central policy module, the very heart of the game, left to write. This will serve as Main so the header is essentially already fixed for us.

```
module Main (main)
    where
```

We pull together all of the abstract modules here, so we start by importing them all. We also import Random as we are going to want to be able to select a random piece to become the new active piece.

```
import Data
import Pieces
import Board
import UI
import Random
```

In this simplified variant the time between ticks is a constant, but in a more sophisticated variant it might be a function on factors such as the score. In either case it makes sense to separate this functionality out into a function so it can easily be tweaked for good playability.

$start\_delay$ :: Delay
$start\_delay = 1000$

Similarly we separate out the desired width and height.

$desired\_dimensions$ :: (Vector, Vector)
$desired\_dimensions = (9, 12)$

We are going to need to be able to get a random new piece both when we create the Board and when we find we need to add a new piece on a Tick event. Thus it makes sense to split the code for doing so off into a separate function.

We use the random number generator in the IO monad so we return an IO Piece rather than just a Piece. We pick a random number in the range of the elements of the *pieces* list, exported by Pieces, and return the element at that position in the list.

get_new_piece :: IO Piece
$get\_new\_piece = randomRIO\ (0, length\ pieces - 1) >\!\!>\!\!= (return \circ (pieces\ !!))$

The *main* function first performs an initialisation phase. The first task is to initialise the user interface, noting the maximum width and height board it can cope with. It then gets a new piece and makes a board, as large as possible while not more than the desired dimensions, with this as the initial piece. The user interface is then asked to perform the relevant changes.

The next phase is performed by a function roughly equivalent to an event loop. It takes the current representation of the board and the time until the next tick event—in this case the time between ticks—and deals with events as they happen.

When the loop finishes we enter the final phase; we tell the user interface to shut down and then the program terminates.

```
main :: IO ()
main = do (width_ui, height_ui) ← init_ui
          let (width_des, height_des) = desired_dimensions
          let width = width_ui `min` width_des
              height = height_ui `min` height_des
          make_board width height
          piece ← get_new_piece
          let (b, cs) = create_board width height piece
          do_changes cs
          event_loop b start_delay
          shutdown_ui
          return ()
```

The actual event loop is complicated mainly by special cases. It starts by getting the next event, passing *get_event* the time until the next tick event is due. The event that occurred and the time that elapsed are returned. If the event was Quit then the loop terminates. Otherwise more complex handling is needed.

If the elapsed time is less than the time until the next tick event and the event wasn't Tick then we subtract the elapsed time from the time until the next tick event to get the new time until the next tick and leave the event unchanged. Otherwise we have either overrun our allocated time (XXX could lose events here) or we have received a Tick event; in either case we reset the time until the next tick and continue as if we had received a Tick event.

If we are dealing with a Tick event and the current piece can't be moved down we get a new piece and use the *next_piece* function to add it to the board. If this succeeds then we call ourselves recursively—the next iteration of the event loop. Otherwise the game is over so we leave the loop. Otherwise we can just use *get_changes* and *do_changes* to work out and apply the changes needed respectively. Then we continue with the next iteration of the event loop.

```
event_loop :: Board → Delay → IO ()
event_loop b d = do (e, elapsed) ← get_event d
                    if e == Quit
                      then return ()
                      else do let (d', e') = if elapsed < d && e ≠ Tick
                                                then (d − elapsed, e)
                                                else (start_delay, Tick)
                              if e' == Tick && not (can_down b)
                                then do piece ← get_new_piece
                                        let (m_b', cs) = next_piece b piece
                                        do_changes cs
                                        case m_b' of
                                             Just b' → event_loop b' d'
                                             Nothing → return ()
                                else do let (b', cs) = get_changes b e'
                                        do_changes cs
                                        event_loop b' d'
```

14 Conclusion

Congratulations, you've made it to the end! We have now finished our design, implementation and (technical) documentation of Hetris, a Tetris-like game.

The experience has been an interesting one for me. It has taken longer to write than a normal, structured implementation would have done. However, it has the advantage of having good documentation built in—perhaps when I come back to look at it in a few years the time will be repaid by the ease with which I can understand how it all works! Perhaps also the result is a better designed program—it is hard to objectively tell.

After a moments thought, taking all the factors into account, I think I would classify the experiment as a success. Not only have I enjoyed writing the surrounding literature, but I believe that a higher quality product is the result. If you have enjoyed reading this, or even been inspired to experiment with literate programming yourself, then it has been even more of a success.

I am interested to hear your views, whether on your own experiences with literate programming, comments on mine or anything else—do drop me a line!

Happy hacking!

Ian Lynagh
igloo@earth.li