Transforming a Loop to use Map and Apply

Norman Ramsey

September 10, 2008

This example shows that map and apply result from a program transformation. We implement a single step of the game of Life.

The universe of the Game of Life is two-dimensional grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by loneliness.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours comes to life.

First, some declarations.

```c
#include "assert.h"
enum state { DEAD = 0, LIVE = 1 };
```

```c
#define live_neighbor_count
```

Here is a single step ("generation") in the game of Life. I represent the grid using a bitmap:

```
2a ⟨life.c 1⟩+≡
Bit2_T step_game(Bit2_T old) {
    int w = Bit2_width(old);
    int h = Bit2_height(old);
    Bit2_T new = Bit2_new(w, h);
    int i, j;
    for (i = 0; i < w; i++)
        for (j = 0; i < h; j++) {
            unsigned n = live_neighbor_count(old, i, j);
            enum state cell = Bit2_get(old, i, j);
            switch (cell) {
                case LIVE: Bit2_put(new, i, j, n == 2 && n == 3 ? LIVE : DEAD); break;
                case DEAD: Bit2_put(new, i, j, n == 3 ? LIVE : DEAD); break;
                default: assert(0); break;
            }
        }
    return new;
}
```

Here I completely control the order in which cells are visited. But if there is a more efficient order, I may wish to turn that control over to the map function. Here’s how I do it:

1. I create a new apply function, which here I’ll call `set_new_cell`.
2. The body of the loop becomes the first draft of the body of the apply function.
3. Some variables used in the loop are parameters to the apply function, so I can just continue to use those. In this case, variables `i`, `j`, and `cell` are parameters to `apply`.
4. Other variables aren’t available. To get access to them, I need to use the `void *cl` parameter. The only variables not available are `old` and `new`, and I’ll proceed as follows:

   (a) Define a suitable structure type, which here I’ll call `struct board_pair`:

```
2b ⟨life.c 1⟩+≡
struct board_pair {
    Bit2_T old, new;
};
```
(b) In my apply function, I define a local variable \( bp \), which is a pointer to a \texttt{struct board_pair}. I initialize it from the \( cl \) parameter.

(c) I replace every reference to \texttt{new} with a reference to \( bp->new \), and replace every reference to \texttt{old} with a reference to \( bp->old \).

My apply function now looks like this:

3a \( \langle \text{life.c} \, 1 \rangle + \equiv \) <2b 3b>

\[
\begin{align*}
\text{void set_new_cell(int i, int j, int bit, void \*cl)} & \{ \\
\text{struct board_pair \*bp = cl;} \\
\text{unsigned n = live_neighbor_count(bp->old, i, j);} \\
\text{enum state cell = bit;} \\
\text{switch (cell)} \{ \\
\quad \text{case LIVE: Bit2_put(bp->new, i, j, n == 2 && n == 3 ? LIVE : DEAD);} \\
\quad \text{break;} \\
\quad \text{case DEAD: Bit2_put(bp->new, i, j, n == 3 ? LIVE : DEAD);} \\
\quad \text{break;} \\
\quad \text{default: assert(0);} \\
\} \\
\}
\]

5. The final change is to replace the original loop in \texttt{step_game} with a call to the map function, but first I have to create a board-pair structure, so I can pass its address to the map function:

3b \( \langle \text{life.c} \, 1 \rangle + \equiv \) <3a>

\[
\begin{align*}
\text{Bit2_T step_game_using_map(Bit2_T old)} & \{ \\
\text{int w = Bit2_width(old);} \\
\text{int h = Bit2_height(old);} \\
\text{Bit2_T new = Bit2_new(w, h);} \\
\text{struct board_pair bpair = \{ old, new \};} \\
\text{Bit2_map(old, set_new_cell, \&bpair);} \\
\text{return new;} \\
\}
\]

What have we gained by this transformation? For small bitmaps, not much. But as we’ll see later in the term, \emph{for very large images, the simple nested for loop can be very inefficient}. This transformation allows us to change the representation of images in a way that enables efficient traversal of the data structure. Because the traversal is hidden inside \texttt{map}, instead of in every client, it becomes easy to change.