The Polar Game in Haskell: An Accidental Online Chautauqua

Paul R. Potts

June-July 2013 and February 2025

In the summer of 2013, while looking for my next job, I wrote a series of blog posts describing my experiments in Objective-C, Scala, Dylan, and Haskell. I boiled those posts about Dylan down into a straightforward discussion of my finished design and code. The posts about Haskell, though, show me learning as I went along.

Initially I was working mostly from books, but then a wonderful thing happened — a number of different Blogger readers found the blog posts and started leaving comments, including detailed suggestions and code snippets. I'd riff off these suggestions in my next installment, and use the suggestions as best I could, and then the commenters came back with further suggestions based on their own experiments with the code.

In other words, it became a sort of seminar, or "Chautauqua" — a shared inquiry. I first discovered the term decades ago when I read Robert Pirsig's book Zen and the Art of Motorcycle Maintenance *and it felt right to me, because I grew up near the Chautauqua Institution in Yew York, and sometimes attended programs there with my family.*

So, this article is really just a cleaned-up version of all these posts, including the comments. As such, it includes snippets of my code, as well as other people's code, that were rough drafts, notes from our "lab notebooks" if you will, and which contain known bugs (and, most likely, unknown bugs as well). The code snippets do not necessarily represent the best practices in Haskell programming at the time the code was written, and almost certainly don't represent the best practices in 2025.

So, I don't necessarily recommend using this code as an example for your Haskell programming, but I certainly do *recommend learning while in dialogue with a community of fellow programmers, like I did; don't be afraid of posting work-in-progress in a public forum; it demonstrates your bravery and your ability to learn. That's the experience I wanted to capture in this article.*

Many thanks to Blogger readers Twan van Laarhoven, Jeff Liquia, BMeph, Roland, Matt Walton, and others who left comments and contributed code to *make this serendipitous occasion happen!*

My Code

The code can be found in my GitHub repository here. Note that I have not touched this code in a dozen years, and I don't know if the latest GCH will even compile and run it correctly.

Introductory Notes about Polar

Many years ago there existed on old-school MacOS a small game called "Polar." It was a very simple game, written by a guy (Go Endo) who was probably a student at the time, but I was fond of it — fond enough to save it for 23 years, with the intention of studying its design and re-implementing it in the future. (I'm a bit of a digital pack-rat, and have saved a lot of old bits and bobs like this.) Years ago, I made notes of how to beat the first 3 levels (it was one of those "incredibly simple but maddeningly difficult" games), drew out the levels, made notes on how the objects behaved, etc. I haven't been able to run that game for a long time, but today I just got it working under the MacOS emulator SheepShaver. Here's what level 1 looks like (blown up a bit):

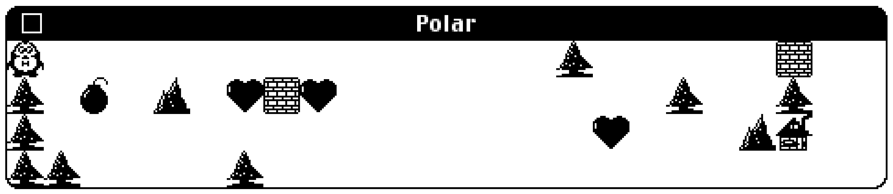

Figure 1: "Polar Level 1"

The penguin is your avatar. The rest of the objects are ice blocks, trees, hearts, bombs, mountains, and houses. The world is a sheet of ice. You can walk around on the ice. Some objects (trees, mountains, and houses) can't be moved, while bombs, hearts, and ice blocks move without friction — if you push them, they will keep going until they hit the edge of the world or another object. Trees don't block the movements of the penguin avatar — it can walk over them as if they were just painted on the ground — but trees will block the movement of other objects. If you *slide* an ice block, and, while sliding, it hits another object, it will stop sliding. If you *push* an ice block directly against another object, the ice block is crushed, and disappears. If you slide a bomb towards a mountain, it will blow up the mountain — both the bomb and the mountain will disappear. Any other object will stop a sliding bomb and it won't blow up.

The goal of the game is to slide all the hearts on the board into the house. Cute, huh? But because the ice is frictionless to everything except the penguin avatar, it's incredibly easy to get objects stuck against walls or corners where you can no longer move them the way you need to, because there are no longer open spaces available for you to get the penguin avatar *into*, and to slide the objects *onto* (remember, you can't *pull* anything, only *push* things by walking into them). So, you have to carefully plan out your moves. If you get stuck, there's an option to start the level over. You might wind up having to use that option a lot.

I should mention that the original game had a copyright notice (1990), and was shareware ($2.00). I can't remember if I ever sent the author $2.00. I'm not sure how he would feel about me taking apart and trying to re-implement his game, or whether he'd try to assert that copyright prevented me from doing so, but I'll assume he's a nice guy and wouldn't care as long as I don't charge for it, and go ahead, on the theory that easier to ask forgiveness than permission. I was not able to find him online — maybe "Go Endo" was a pseudonym?

Day 1

So if you've been following recent posts, you know I've been messing with the logic for a simple sliding-tile game. In my last post I took some designs refined via a side trip into Dylan and brought them back into Objective-C, making them a little more idiomatic by pruning my tile classes that didn't hold their weight, and compensating for Objective-C's very limited method dispatch options.

But in addition to learning Objective-C, and Apple's APIs for writing an app, I'm also trying to further my knowledge of Haskell, which is somewhere just beyond "utter newbie." So I'm going to try to implement the game logic in Haskell, too. Since the game is decidedly stateful, there is a certain impedance mismatch here, at least with respect to the early chapters in most of the tutorials and guides. But I'm told that Haskell also makes a great imperative programming language, so let's give it a shot. And along the way I can try to mold my understanding of stateful versus imperative a little more.

For day one, which was a shorter-than-usual day, I did not get into the state monad or how to model mutation of a 2-D array yet. I wanted to consider whether I could model the tile classes the way I could in Dylan, and do something useful in them. It occurred to me that each move of the penguin, and all the subsequent actions including possibly pushing an object, possibly a collision, possibly an object sliding frictionlessly as long as it can and then another collision, actually takes place in a 1-dimensional vector, not a 2-dimensional array. So it might be interesting to handle a penguin move by extracting a vector (in the form of a list) from the array, and replacing it with an updated list.

I haven't worked that all out yet but here is the bare beginning of my experimentation. There's a way to represent tiles:

```haskell
data Tile = Empty | Tree | Mountain | House | Ice_Block |
    Bomb | Heart | Edge
    deriving (Show)
```

Part of the confusion of learning Haskell is that, semantically, this isn't quite the equivalent of a set of enumerations, or of a set of class declarations. From what I can tell, this is more like a list of singleton factories — constructors, where I've also derived them from **Show**, sort of the equivalent of mixing in a base class. But this is all an approximation, and Haskell is *quite* different than the other languages I'm most familiar with.

My next thought was that I wanted to be able to declare "base classes" so that, for example, I could have a **Walkable** class that comprised **Empty** and **Tree**. In Dylan I would do this by using classes, but there is different way: declaring a **type-union** of singletons. I think that this Haskell solution is more like the **type-union**. I looked in vain for an explicit type union. Instead I found **class** (which, in Haskell, does not correspond to a class in the sense that I'm used to, of a template for a run-time object that consists of data members and methods to operate on it, but a *typeclass*, something I clearly need to study more):

```
class Walkable a where
    walkable :: a -> Bool
```

And then this: which boils down to, I think, a function to determine whether a Tile is an instance of a Walkable typeclass:

```
instance Walkable Tile where
    walkable Empty = True
    walkable Tree = True
    walkable _ = False
```

Now I can write something like this (just a vague thought-in-progress at the moment):

```
slide :: [Tile] -> [Tile]
slide [] = error "slide empty list!"
slide (t) = error "single item list!"
slide (Empty:ts) = ts ++ slide ts

collide :: [Tile] -> [Tile]
collide [] = error "traverse empty list!"
collide [Edge] = [Edge]
collide (Empty:ts) = ts
collide (Bomb:Mountain:ts) = [Empty, Empty] ++ ts
collide (Heart:House:ts) = [Empty, House] ++ ts

step :: [Tile] -> Bool
step [] = error "step: empty list!"
step (t:_) = if walkable t then True else False
```

Then after sticking in a dummy **main**, I can load this into GHCI and interact with it a little:

```
*Main> :t Tree
Tree :: Tile
```

```
*Main> step [Mountain, Empty, Empty, Tree, Edge]
False
*Main> step [Tree, Empty, Empty, Tree, Edge]
True
*Main> collide [Heart, Mountain]
*** Exception: arctic-slide.hs:(22,1)-(26,47):
    Non-exhaustive patterns in function collide
```

(Um, yeah, OK, I have work to do there…)

```
*Main> collide [Heart, House]
[Empty,House]
*Main> slide [Empty, Empty, Empty, Empty, Mountain]
*** Exception: single item list!
```

(Whoops…)

Anyway, that's not exactly what I want to do — really, I want the functions to actually return a new list of the same length, so I'll have to build it up as I recurse down the list — maybe in the **List** monad? But it's a start on the whole basic concept of matching on the "types" of my tiles in a "vector," such as it is. That whole bit with **Walkable** — which I admit I don't quite understand yet — seems like far too much conditional logic when I really just want to pattern-match on a type union of **Tile**. In other words, I want to write something like this (not valid Haskell):

```
    union Walkable = Empty | Tree
    step (Walkable:_) = True
```

That's a small example, but I have several other type union classes I need to use to categorize the tiles, so I have an incentive to make that as clear and simple as possible. It seems like I'm still fighting with Haskell's idioms here. Clearly, as they say, more research is needed…

Day 1 Comments

Blogger reader Twan van Laarhoven wrote:

> Some tips:

```
slide (t) = error "single item list!"
```

> The pattern **(t)** matches any list; you are looking for [**t**].
>
> Instead of using **if, then, else**, use guards:

```
step :: [Tile] -> Bool
step [] = error "step: empty list!"
step (t:_) | walkable t = True
step (t:_) | otherwise = False
```

Or just

```
step (t:_) = walkable t
```

or even

```
step = walkable . head
```

If you really want to write this as a pattern, there are two options. Use view patterns:

```
{-# LANGUAGE ViewPatterns #-}
step (walkable -> True : _) = True
```

or change the datatype:

```
data WalkableTile = Empty | Tree
data UnwalkableTile =
    Mountain | House | Ice_Block | Bomb | Heart | Edge
data Tile = Walkable WalkableTile | Unwalkable UnwalkableTile
```

Also, your **Walkable** class doesn't add much. It could just be a standalone function **walkable :: Tile -> Bool**.

Finally, you should not unnecessarly raise errors. For example, why isn't **slide** [] just []?

I responded:

T v. L, thanks for much for your help. It is embarrassing being such a beginner but I am very happy to get pointers. The books and tutorials are frustrating because there seems to be a gap between very simple program tailor-made for Haskell implementations, that they cover, and slightly more complicated program, without a good bridge between the two other than trying to read the code for much bigger published Haskell programs.

I am interested in learning more about **ViewPatterns** — I have not heard of those. Today I will be seeing if I can get the **List** monad working to collect up the altered lists.

You are right about the **Walkable** class. There are some other classes that have more members, but they also overlap membership. That's probably a very un-Haskellish way to do it since it introduces multiple possible matches, although I know Haskell evaluates the patterns in strict order. See the earlier Dylan code for what I mean. In Dylan I'm relying on the runtime to check for matches in order from less specific to more specific, and that seems to work OK, but

> I don't have a "catch everything" case deliberately — since if all the methods don't match, I want a run-time error, as it indicates a design error upstream.

Blogger reader BMeph replied:

> @Paul: I think the "more Haskellish" way to indicate a design error would be to have a compile-time error, not a run-time one. Of course, that only works when you are knowingly "doing it wrong," but still, there it is.
>
> @Twan v. L: The second part of the **step** definition that starts **step (t:_) |...** should be all blanks up to that | part.

Day 2

Another short day since I had several phone interviews. Thanks to the folks who left comments!

I got a little further today; I feel like I'm starting to understand Haskell's data handling a little bit better. It's a cliché but I think the hard part is un-learning, and understanding what something like this *doesn't* do. So here's where it stands now — not finished by any means, but coming along, with painful slowness as I continue to learn:

```haskell
data Dir = North | East | South | West
    deriving (Show, Eq)

data Pos y x = Pos Int Int
    deriving (Show, Eq)

-- N.B.: capitalization of initial letters in posY, posX is
-- semantically important!

posY ( Pos y x ) = y
posX ( Pos y x ) = x

data Tile = Empty | Tree | Mountain | House | Ice_Block |
    Bomb | Heart | Edge deriving (Show, Eq)

-- Different types of tiles have different properties in
-- different interaction contexts:
-- The penguin can walk through empty tiles or trees (forest)
walkable :: Tile -> Bool
walkable t = ( t == Empty ) || ( t == Tree )
```

```
-- But everything except empty tiles will block sliding objects
blocking :: Tile -> Bool
blocking t = ( t /= Empty )

-- A subset of tiles are movable (and will slide until blocked)
movable :: Tile -> Bool
movable t = ( t == Bomb ) || ( t == Heart ) || ( t == Ice_Block )

-- A subset of tiles aren't movable; note that this set
-- overlaps blocking and that Tree is both walkable and fixed
fixed :: Tile -> Bool
fixed t = ( t == House ) || ( t == Mountain ) || ( t == Edge )
```

That all should be fairly non-controversial, I think. The predicate approach to classifying tiles in different contexts may actually make more sense in Haskell, given that I can then use these predicates as guards. The replacement for a simple struct, **Pos**, still feels awkward — I haven't really dug into whether it could be improved with record syntax, or some other technique. For now it's there because it works.

All the beginner tutorials say "don't use arrays, don't use arrays, don't use arrays!" At least not until I reach the stage where I need to optimize the implementation. So I'll try that. Let's try a list, and I'll extract "slices" from it, lists starting at a given **Pos** going in one of four different directions. Eventually I want the slice function to terminate the slices with **Edge** tiles that aren't actually stored in the list. So… I have to think about this some more, but here's a single case, sort of taken care of:

```
type Board = [[Tile]]
slice :: Board -> Pos y x -> Dir -> [Tile]
slice board pos East = drop ( posX pos )
    $ head $ drop ( posY pos ) board
slice _ _ _ = error "slice: not handled yet!"
```

I don't have **slide** finished, but here's a version of **collide** that works, at least a little:

```
collide :: [Tile] -> [Tile]
collide (t:(Empty:ts)) | movable t =
    [Empty] ++ collide (t:ts)
collide (Bomb:(Mountain:ts)) = [Empty, Empty] ++ ts
collide (Heart:House:ts) = [Empty, House] ++ ts
collide (_) = error "collide: unexpected case!"
```

The nested pattern **(Bomb:(Mountain:ts))** was sort of a flash of inspiration — but it appears that maybe both this version and the **(Heart:House:ts)** version work the same — I think — so perhaps it's kind of pointless. It seemed to go along with the "destructure it the way you would structure it" idea, although I would normally not build a list out of cons cells unless it was irregular in some way.

Here's the penguin step function, returning True if the penguin can move onto the tile at the head of the list:

```
step :: [Tile] -> ( Bool, [Tile] )
step [] = error "step: empty list!"
step ts = if walkable (head ts) then ( True, ts )
                                else ( False, collide ts )
```

And there's **move**, which "absorbs" the case where the penguin is turned to face a different direction. It's not really done; the idea is that it will give back the board, basically generating a new world. For now we kind of punt on the question of how to rebuild the board out of the existing board and the modified "slice" — and so the I just return a list as the first element of the tuple. In the first case where the penguin hasn't moved, that doesn't actually make sense, but it satisfies GHC for now (wow, she's kind of a harsh mistress, but you've got to love those thigh-high black leather boots!)

```
move :: Board -> Pos y x -> Dir -> Dir ->
    ( [Tile], Pos y x, Dir, Dir )

move board pos move_dir penguin_dir =
    if move_dir /= penguin_dir
    then ( head board, pos, move_dir, move_dir )
    else ( collide $ slice board (Pos 1 0) penguin_dir,
        pos, penguin_dir, penguin_dir )
```

Boy, that's tuple-icious... not sure I like it, but it's a start. So:

```
*Main> walkable Tree
True
*Main> :t Pos
Pos :: Int -> Int -> Pos y x
*Main> let slice = [Heart, House]
*Main> collide slice
[Empty,House]
*Main> let slice = [Bomb, Empty, Mountain]
*Main> collide slice
[Empty,House]
*Main> let board = [[Empty, Tree, Empty, Edge],
    [Bomb, Empty, Mountain, Edge]]
*Main> move board (Pos 1 0) West East
([Empty,Tree,Empty,Edge],Pos 1 0,West,West)
*Main> move board (Pos 1 0) East East
([Empty,Empty,Empty,Edge],Pos 1 0,East,East)
```

More tomorrow if I can manage it!

Day 2 Comments

Blogger reader BMeph wrote:

> I'm pretty sure that the **data Pos y x = ...** line doesn't do what you think it does. Unless you don't mind people using **Pos String Double** types in their code.
>
> I like watching the process of a programmer getting a grip on what Haskell can, can't, does, and doesn't do — please, keep it up!

I replied:

> Thanks, BMeph. I am still wrestling with what **data** actually does. There are lots of examples that show using it, but the only thing I've seen that strictly describes what it does is the formal language definition, and I struggle with the formal semantics a little.

Blogger reader Noah (I think it was Noah; some comments have been lost and users deleted over the years) wrote:

> I would suggest **data Pos = Pos { posX :: Int, posY :: Int}** instead. This constructs the **posX** and **posY** for you automatically, and removes the two type parameters **x** and **y**, which like BMeph said, you don't really want.

I replied:

> Thanks, Noah. I had moved to **Pos Int Int**. I'd like to be rid of having to write getters. If you have a minute, do you think you could explain what the type parameters are for/do? Not explain like I'm five, but like I'm more accustomed to C, Java, Dylan, or Scheme. I was just reading this which talks about the different namespaces that the parts of the **data** declaration use, but that doesn't seem to tell the whole story, and I'm still looking for that whole story on **data**. If I can collect up enough pieces and get to the point where I feel I really understand all the options, I am volunteering to write "**data** for noobs" or some such.

I did not see a reply from Noah, but Blogger reader Ykarin-chan wrote:

> I am just going to describe the default Haskell '98 syntax, GADTs and records and empty data types change things, but aren't important at first. (Though in some ways the syntax for GADTs is clearer. I often find myself using it even when it is unnecessary.)

With a **data** declaration you are defining a "type constructor" on the left hand side of the = and 1 or more "data constructors" on the right hand side. So the simplest possible data declaration is:

```
data T = U
```

Defining a constant "type constructor" **T**, and a constant "data constructor." Note that the type constructor and the data constructor are in different namespaces, so they can, but do not have to be, the same lexicographically.

In fact the Haskell type **()**, the 0-tuple, is defined as:

```
data () = ()
```

The next complication is that you can have multiple **data** constructors for the same type:

```
data Bool = True | False
```

The | is meant to be reminiscent of "or," since a **Bool** can be either **True** or **False**.

Now the next complication is that "data constructors" can be "higher-typed," like your **Pos** example:

```
data Pos = Pos Int Int
```

This means that the data constructor **Pos** has type **Int -> Int -> Pos**. So in order to construct an element of the type **Pos** (in the type namespace), we use the **Pos** (in the data namespace) on two Ints:

```
Pos 3 4 :: Pos
```

Or the type of Peano natural numbers, which are hideously inefficient but mathematically simple.

```
data Peano = Zero | Succ Peano
```

So **Zero** is a constant data constructor, but **Succ** is a function from **Peano** to **Peano**.

Now you can also make the type constructor a function and not a constant, but the syntax is a little different. We put (type) variable names after the type constructor, and the compiler figures out their "type" (really, their "kind") for us. For example lets look at **Maybe**:

```
data Maybe a = Nothing | Just a
```

This means that the **Maybe** type constructor is not usable as a type by its self. Its "higher-kind-ed" in that it takes a type, and gives you a new type. So **Maybe** isn't a type, but **Maybe Int** is. The compiler sees that you are using **a** directly as a type, and so gives it **Kind ***, which means **Maybe** has kind *** -> ***. (***** is just the kind notation for the kind of a actual type, (as opposed to a type function).

> Now type variables, despite their name, do not actually have to have kind *.
>
> So if you took out the record syntax sugar, **StateT**'s **data** declaration would be writen:

```
data StateT s m a = StateT (m (a, s))
```

> where **s** and a have kind *, but **m** has kind * **->** *.
>
> Does that help at all?

I replied:

> Yukarin-chan, that helps a lot! It seems to put together in one place bits and pieces I've seen in several different places. I will study on that for a while — it usually takes me a couple times, and then actually using it more than once, to really understand something in Haskell. But I'm getting there!

Blogger reader Jedaï wrote:

> Since you're familiar with Java, Maybe a is analog to generic types in Java. This is in fact very common in Haskell though we call it polymorphism.

Day 3

More phone interviews, more coding. On my laptop, amidst a gaggle of fighting children, during a thunderstorm, with our basement flooding, with the kind assistance of some friendly commentors, a little more progress. Let's change **Pos**:

```
data Pos = Pos { posY :: Int, posX :: Int }
    deriving (Show, Eq)
```

And define a game world:

```
data World = World { board :: Board, penguinPos :: Pos,
                          penguinDir :: Dir,
                          heartCount :: Int } deriving (Show)
```

It was painful, took an embarrassingly long time, and this can't possibly be how I want to keep it indefinitely, but I finished **slice** which treats a list of lists of tiles like a 2-dimensional array and gives us what the penguin sees before him, looking in a given direction:

```
slice :: Board -> Pos -> Dir -> [Tile]
slice board pos East = ( drop ( posX pos ) $
    board !! ( posY pos ) ) ++ [Edge]
slice board pos South = ( drop ( posY pos ) $
    ( transpose board ) !! ( posX pos ) ) ++ [Edge]
slice board pos West = ( reverse $ take ( posX pos + 1 ) $
    board !! ( posY pos ) ) ++ [Edge]
slice board pos North = ( reverse $ take ( posY pos + 1 ) $
    ( transpose board ) !! ( posX pos ) ) ++ [Edge]
```

Let's just leave that as it is for now and use it, with the intent of replacing it with a real array of some sort later on. I still have to figure out how to merge a modified penguin track with an unmodified board to create the next state of the entire board… that's not going to be pretty, but it's doable.

So, one of the things I really love about Haskell is that once you get these pieces, they really do start come together nicely. Let's go ahead and define the first board. I could make it from the strings or a run-length encoding or something, but for now let's just bite the bullet and build the list the hard way:

```
    get_initial_board :: [[Tile]]
    get_initial_board = [[Tree,Empty,Empty,Empty,Empty,Empty,
                          Empty,Empty,Empty,Empty,Empty,Empty,
                          Empty,Empty,Empty,Tree,Empty,Empty,
                          Empty,Empty,Empty,Ice_Block,Empty,Empty],
                         [Tree,Empty,Bomb,Empty,Mountain,Empty,
                          Heart,Ice_Block,Heart,Empty,Empty,Empty,
                          Empty,Empty,Empty,Empty,Empty,Empty,
                          Tree,Empty,Empty,Tree,Empty,Empty],
                         [Tree,Empty,Empty,Empty,Empty,Empty,
                          Empty,Empty,Empty,Empty,Empty,Empty,
                          Empty,Empty,Empty,Empty,Heart,Empty,
                          Empty,Empty,Mountain,House,Empty,Empty],
                         [Tree,Tree,Empty,Empty,Empty,Empty,
                          Tree,Empty,Empty,Empty,Empty,Empty,
                          Empty,Empty,Empty,Empty,Empty,Empty,
                          Empty,Empty,Empty,Empty,Empty,Empty]]

    penguin_view :: Board -> Pos -> Dir -> [Tile]
    penguin_view board pos dir = drop 1 $ slice board pos dir
```

So now we can actually start doing stuff with this. Here's what's in front of the penguin, from different vantage points, facing in different directions:

```
*Main> penguin_view get_initial_board (Pos 0 0) East
[Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,
Empty,Empty,Empty,Empty,Empty,Tree,Empty,Empty,Empty,Empty,
Empty,Ice_Block,Empty,Empty,Edge]

*Main> penguin_view get_initial_board (Pos 0 0) South
[Tree,Tree,Tree,Edge]
```

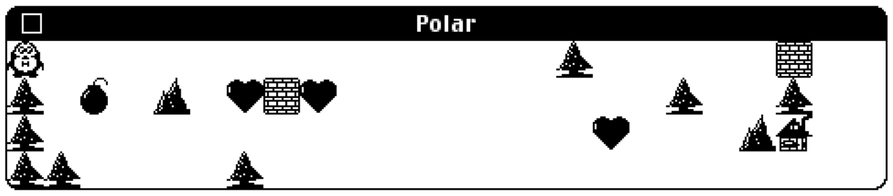

Figure 2: "Polar Level 1"

```
*Main> penguin_view get_initial_board (Pos 0 0) West
[Edge]

*Main> penguin_view get_initial_board (Pos 0 0) North
[Edge]

*Main> penguin_view get_initial_board (Pos 3 21) North
[House,Tree,Ice_Block,Edge]
```

Fun! Tomorrow, if I can manage it… an updated world.

Day 3 Comments

No comments were left on the Day 3 post.

Day 4

OK, things are getting meaty: I've made some minor modifications to **World**:

```
data World = World { wBoard :: Board, wPenguinPos :: Pos,
                     wPenguinDir :: Dir, wHeartCount :: Int }
                     deriving (Show)
```

This extracts the sequence of tiles in front of the penguin, for various directions, from a nested list representation of the board:

```
view :: Board -> Pos -> Dir -> [Tile]
view board pos East = ( drop ( posX pos + 1 ) $
    board !! ( posY pos ) ) ++ [Edge]
view board pos South = ( drop ( posY pos + 1 ) $
    ( transpose board ) !! ( posX pos ) ) ++ [Edge]
view board pos West = ( reverse $ take ( posX pos ) $
    board !! ( posY pos ) ) ++ [Edge]
```

```
view board pos North = ( reverse $ take ( posY pos ) $
    ( transpose board ) !! ( posX pos ) ) ++ [Edge]
```

I have fleshed out **slide** and **collide** after some testing; I haven't tested all my known cases yet. Maybe tomorrow. Here is how I create the initial world:

```
init_world :: World
init_world = ( World init_board ( Pos 0 0 ) South 3 )
```

South because in the south-facing representation, the penguin's face is visible (although of course I don't have a GUI yet).

A little utility function for clarity:

```
nest :: [a] -> [[a]]
nest xs = [xs]
```

And now, deep breath, the logic to build the next board out of the current board combined with a replaced list of tiles that may have been changed due to object interaction. It gets pretty ugly here when we're undoing the appending of **Edge** with init, and undoing the reversing that view has done when looking **North** and **West**, and working with the transposed board for **North** and **South**. There are some extra line breaks in there that are not in the working code. I have an issue with my **let** clauses not compiling correctly if I break the lines. I'm sure there's a prettier workaround, and I will look that up, but after going down a rabbit hole of Haskell syntax, I have timed out for today and right now I'm just happy it runs:

```
next_board :: Board -> Pos -> Dir -> ( Bool, Board )
next_board board pos East =
    let ( penguin_could_move, updated_view ) =
        step $ view board pos East
    in (
        penguin_could_move,
        take ( posY pos ) board ++
        nest (
            ( take ( posX pos + 1 )
                ( board !! ( posY pos ) ) ) ++
            ( init updated_view ) ) ++
        drop ( posY pos + 1 ) board )
next_board board pos South =
    let ( penguin_could_move, updated_view ) =
        step $ view board pos South
    in (
        penguin_could_move,
        transpose (
            take ( posX pos ) ( transpose board ) ++
            nest (
                ( take ( posY pos + 1 )
                        ( ( transpose board ) !! ( posX pos ) ) ) ++
                    ( init updated_view ) ) ++
            drop ( posX pos + 1 ) ( transpose board ) ) )

next_board board pos West =
    let ( penguin_could_move, updated_view ) =
        step $ view board pos West
    in (
        penguin_could_move,
        take ( posY pos ) board ++
        nest (
            ( reverse ( init updated_view ) ) ++
            ( drop ( posX pos )
                ( board !! ( posY pos ) ) ) ) ++
        drop ( posY pos + 1 ) board )

next_board board pos North =
    let ( penguin_could_move, updated_view ) =
        step $ view board pos North
    in (
        penguin_could_move,
            transpose (
            take ( posX pos ) ( transpose board ) ++
            nest (
                ( reverse ( init updated_view ) ) ++
                ( drop ( posY pos )
                    ( ( transpose board ) !! ( posX pos ) ) ) ) ) ++
            drop ( posX pos + 1 ) ( transpose board ) ) )
```

That… seems like way too much code, and I would like to kill it in favor of using a real array type — soon. The tutorials were pretty insistent that I try to use lists. I'm pretty sure this is not what they meant. I will say that I was really impressed, writing this, how much of it worked the first time, as soon as I got it past the compiler. But that doesn't necessarily mean this is the best possible design for this code.

Anyway, updating penguin pos:

```
next_ppos :: Pos -> Dir -> Pos
next_ppos pos East = ( Pos ( posY pos ) ( posX pos + 1 ) )
next_ppos pos South = ( Pos ( posY pos + 1 ) ( posX pos ) )
next_ppos pos West = ( Pos ( posY pos ) ( posX pos - 1 ) )
next_ppos pos North = ( Pos ( posY pos - 1 ) ( posX pos ) )
```

And, updating the world. I had a similar problem with the line-broken **let** clause here:

```
next_world :: World -> Dir-> World
next_world old_world move_dir =
    let ( can_move, board ) = next_board ( wBoard old_world )
        ( wPenguinPos old_world ) ( wPenguinDir old_world )
    in
        if ( move_dir /= wPenguinDir old_world )
        then ( World ( wBoard old_world )
                     ( wPenguinPos old_world )
                   move_dir ( wHeartCount old_world ) )
        else ( World board
                   ( next_ppos ( wPenguinPos old_world )
                               ( wPenguinDir old_world ) )
                   ( wPenguinDir old_world )
                   ( wHeartCount old_world ) )
```

Now, some pretty-printing, since it gets pretty tedious to visualize the board from reading the dumped-out list in GHCI:

```
pretty_tiles :: [Tile] -> String
pretty_tiles [] = "\n"
pretty_tiles (t:ts) = case t of
                 Empty     -> "___ "
                 Mountain  -> "mtn "
                 House     -> "hou "
                 Ice_Block -> "ice "
                 Heart     -> "hea "
                 Bomb      -> "bom "
                 Tree      -> "tre "
                 Edge      -> "### "
             ++ pretty_tiles ts

pretty_board :: Board -> String
pretty_board [] = ""
pretty_board (ts:tss) = pretty_tiles ts ++ pretty_board tss

pretty_world :: World -> String
pretty_world world =
    "penguin @: " ++ show ( wPenguinPos world ) ++
    ", facing: "  ++ show ( wPenguinDir world ) ++
    ", hearts: "  ++ show ( wHeartCount world ) ++
    "\n" ++ pretty_board ( wBoard world )
```

And here's where the rubber meets the road — or, rather, fails to. I need state, at least simulated state. I messed with state monads for a while but I'm not quite ready. I will tackle that another day. I messed with trying to capture a list in a closure and append a series of successive worlds to it but while that would work fine in Scheme, Lisp, or Dylan I realized that in Haskell I was just fighting the entire language design. So I gave in and did this stupid thing for now, just so I could see my world updating and start to validate that all the tile interactions on the board work:

```
main :: IO ()
main = do
    putStrLn "ArcticSlide start"
    let world0 = init_world
    putStrLn $ pretty_world world0

    -- 21 East
    let world5 =
        next_world ( next_world ( next_world ( next_world (
            next_world world0  East ) East ) East ) East ) East
    let world10 =
        next_world ( next_world ( next_world ( next_world (
            next_world world5  East ) East ) East ) East ) East
    let world15 =
        next_world ( next_world ( next_world ( next_world (
            next_world world10 East ) East ) East ) East ) East
    let world20 =
        next_world ( next_world ( next_world ( next_world (
            next_world world15 East ) East ) East ) East ) East
    let world21 = next_world world20 East
    putStrLn $ pretty_world world21
    -- 2 South
    let world23 = next_world ( next_world world21 South ) South
    putStrLn $ pretty_world world23
    -- 3 East
    let world26 = next_world ( next_world (
        next_world world23 East ) East ) East
    putStrLn $ pretty_world world26
    -- 2 North
    let world28 = next_world ( next_world world26 North ) North
    putStrLn $ pretty_world world28
    -- 2 West
    let world30 = next_world ( next_world world28 West ) West
    putStrLn $ pretty_world world30
```

That is far from what I'd like to be doing eventually with managing game moves, and I still haven't put in any handling for the heart count, but it works:

Apologies for the formatting with my current web page template; this looked OK in Blogger. I need to update the template to support wider monospaced text blocks, but I am not a CSS guru by any means.

```
ArcticSlide start
penguin @: Pos {posY = 0, posX = 0}, facing: South, hearts: 3
tr __________________________________________tr _______________ic ______
tr ___bo ___mt ___he ic he ___________________________tr ______tr ______
tr _____________________________________________he _________mt ho ______
tr tr ____________tr ___________________________________________________

...

penguin @: Pos {posY = 0, posX = 22}, facing: North, hearts: 3
tr __________________________________________tr _______________ic ______
tr ___bo ___mt ___he ic he ___________________________tr ______tr ______
tr _____________________________________________he _________mt ho ______
tr tr ____________tr ___________________________________________________

penguin @: Pos {posY = 0, posX = 21}, facing: West, hearts: 3
tr __________________________________________tr ic _____________________
tr ___bo ___mt ___he ic he ___________________________tr ______tr ______
tr _____________________________________________he _________mt ho ______
tr tr ____________tr ___________________________________________________
```

Aaaand… the penguin has pushed the ice block in the upper right to the west, and it has slid west and become blocked by the tree. That's… good, right? My brain is a little fried. All that to update a game board. I need a break, and maybe a stiff drink. I'm going to have to fortify myself before I tackle the state monad. But I am determined!

Day 4 Comments

Blogger reader Roland wrote:

> Hi Paul, I like your post about this Polar game. Some remarks about **Edge**.
>
> I suggest you factor out ++ [**Edge**] from the view function as:

```haskell
view :: Board -> Pos -> Dir -> [Tile]
view board pos dir = (view' board pos dir) ++ [Edge]
where
view' board pos East = ( drop ( posX pos + 1 ) $ ...
```

> Do you really want to add the **Edge** tile every time you construct a slice? An alternative would be to have an **addEdge** function, that adds all the edges to a board after you create it with **get_initial_board**.
>
> Do you really need the **Edge** tile? If the penguin sees an empty list of tiles, it's at the edge of his nice small world. You could avoid all the runtime errors like:

```
collide [] = error "traverse empty list!" or
step [] = error "step: empty list!"
```

> If you have no runtime errors, the chances that your program fails are much smaller!
>
> Thanks and regards,
>
> Roland

He left a second comment:

> Hi Paul,
>
> **next_ppos** could be written as:

```
next_ppos :: Pos -> Dir -> Pos
next_ppos pos dir = Pos (posY pos + fst step) (posX pos + snd step)
  where
    step = delta dir
    delta East = (0, 1)
    delta South = (1, 0)
    delta West = (0, -1)
    delta North = (-1, 0)
```

> So you split the adding from the direction stuff.
>
> Regards
>
> R.

I replied:

> Hi Roland,
>
> Yeah, it has occurred to me that if I'm actually using lists where I can detect the end case, there isn't really a need for the **Edge** tile. I could just delete it entirely. It is a hold-over from the Objective-C version where I use it as a flag to avoid dereferencing array elements past the bounds.
>
> Thanks for the simplification of **next_ppos** — that's a good idea.
>
> It also occurred to me as I ate dinner last night that I can use a fold on a list of moves to collect up the updated boards. I don't think in the fully functional mind-set yet. But that should be easy to implement — I'll do that for next time, which should allow me to get the rest of the boards up and running quickly and validate all the cases.

Day 4 and a Half: Folding a Penguin

So, just a quick update today. While I was cooking bacon this morning I looked at comments and tried to implement an idea I had last night. Roland suggested I could get rid of **Edge**. I had already been asking myself this. Using a special flag value for the edge-of-board case came from the Objective-C version where I wanted to avoid reading tiles outside the bounds of the board array. When using lists there is a built-in termination condition, so Edge is gone completely.

Roland also suggested a simplified next_ppos, like so:

```
next_ppos :: Pos -> Dir -> Pos
next_ppos pos dir = Pos ( posY pos + fst step )
                        ( posX pos + snd step )
    where step = delta dir
          delta East = ( 0, 1 )
          delta South = ( 1, 0 )
          delta West = ( 0, -1 )
          delta North = ( -1, 0 )
```

So that's in there now. Thanks, Roland!

The next thing I wanted to do is get rid of that ugly test code with all the nested calls to **next_world**. I was re-reading *Learn You a Haskell* and it occurred to me that this sort of thing — distilling a list — is what *folds* are for. And then, a minute later, that I don't actually want to *fold* the worlds down to one final world — I want to capture all the intermediate worlds as we process a list of moves. And that's what a *scan* is for. So we're conducting surveillance on the penguin as he goes about his business. GHCI tells me that the type of **scanl** is **(a -> b -> a) -> a -> [b] -> [a]**. So I'm calling it with a function that takes a **World** and a **Dir** and returns a **World**. That's the **(a -> b -> a)** part. Then it gets an initial **World**, that's the **a**, and a list of elements of type **Dir**, that's the **[b]**, and returns a list of elements of type **World**, that's **[a]**.

```
moves_to_dirs :: [(Dir, Int)] -> [Dir]
moves_to_dirs [] = []
moves_to_dirs (m:ms) =
    replicate ( snd m ) ( fst m ) ++ moves_to_dirs ms

moves_board_1 = [(East,21),(South,2), (East,3),(North,2),(West,2)]

move_sequence :: [(Dir,Int)] -> [World]
move_sequence repeats = scanl next_world init_world steps
    where steps = moves_to_dirs repeats

main :: IO ()
```

```
    main = do
        mapM_ putStrLn pretty_worlds
        where worlds = move_sequence moves_board_1
```

And that gives me the whole shebang, ending in:

```
penguin @: Pos {posY = 0, posX = 22}, facing: West, hearts: 3
tr __________________________________________tr _______________ic ______
tr ___bo ___mt ___he ic he ___________________________tr ______tr ______
tr _____________________________________________he _________mt ho ______
tr tr ____________tr ___________________________________________________

penguin @: Pos {posY = 0, posX = 21}, facing: West, hearts: 3
tr __________________________________________tr ic _____________________
tr ___bo ___mt ___he ic he ___________________________tr ______tr ______
tr _____________________________________________he _________mt ho ______
tr tr ____________tr ___________________________________________________
```

Oh, if you just want to see the final result, **foldl** will work here. Their types are almost identical, except that **foldl** returns a single **a** (in this case, a **World**) instead of a list of elements of type **World**. So a function to make use of that just returns a single **World**, but everything else is the same. Like so:

```
    move_sequence' :: [(Dir,Int)] -> World
    move_sequence' repeats = foldl next_world init_world steps
        where steps = moves_to_dirs repeats
```

And then I can display both:

```
    main :: IO ()
    main = do
        mapM_ putStrLn pretty_worlds
        putStrLn pretty_final_world
        where worlds = move_sequence moves_board_1
              final_world = move_sequence' moves_board_1
              pretty_worlds = map pretty_world worlds
```

I like it — examples of fold and scan that are a little more complex than the usual textbook examples. Personally I'd rather read more of those and less about how we can implement some simple math operation that can be trivially implemented in a dozen other, more readable ways.

Day 4 and a Half Comments

Blogger reader Jeff Licquia wrote:

> Hi! Fellow Haskell newbie here, so please don't take this as "advice from the sages" as much as "blatherings from the potentially confused".

I was fascinated by the series of blog posts, mostly because your **next_board** function bothered me. It seemed to me like it did too much, and could be factored down a bit more. In my experience, Haskell is about splitting actions down to little nubs, and re-composing them in fancy ways, and it seemed that **next_board** could do that.

So first I factored out the multiple-return-value part:

```
next_board :: Board -> Pos -> Dir -> ( Bool, Board )
next_board board pos dir =
    let ( penguin_could_move, updated_view ) =
        step $ view board pos dir
        in ( penguin_could_move, update_board_from_view board
             pos dir updated_view )
```

After cutting-and-pasting my way to a working **update_board_from_view**, I recognized that there are basically two operations: applying a **view** to a row forwards, and applying a **view** to a row backwards. That looked like this:

```
apply_view_to_row :: [Tile] -> Int -> Bool -> [Tile] -> [Tile]
apply_view_to_row orig pos True update =
    take ( pos + 1 ) orig ++ ( init update )
apply_view_to_row orig pos False update =
    ( reverse ( init update ) ) ++ ( drop pos orig )
```

That takes care of moving **East** and **West**. For **North** and **South**, I noticed that the operations were exactly the same once you ignored the transposes and swapped the **X** and **Y** parameters. So:

```
apply_view_to_rows :: Board -> Int -> Int -> Bool -> [Tile] -> Board
apply_view_to_rows orig row pos is_forward update =
    take row orig ++
    nest ( apply_view_to_row ( orig !! row ) pos is_forward update )
        ++ drop ( row + 1) orig

update_board_from_view :: Board -> Pos -> Dir -> [Tile] -> Board
update_board_from_view board pos dir updated_view
| is_eastwest = apply_view_to_rows board ( posY pos ) ( posX pos )
    is_forward updated_view
| otherwise = transpose ( apply_view_to_rows ( transpose board )
    ( posX pos ) ( posY pos ) is_forward updated_view )
    where is_forward = elem dir [East, South]
          is_eastwest = elem dir [East, West]
```

There's probably more interesting refactorings that could be done, but this looks a lot better to me.

> Thanks for the brain teaser!

I replied:

> Thanks, Jeff! Some good ideas there. I was not really happy with the original functions. I had it in mind that I wanted to come back and work on them later, but once it worked I wanted to push on first to get to the point where I could exercise the game play. I am hoping there will be a major simplification possible by moving to an array type. It seems like there should be a way to express extracting or replacing a whole (or partial) row or column from an a 2-D array in one line of code. If so then all the game logic outside of pretty-printing and maybe a GUI ought to fit on a page, and read as simply as the concept is.

Blogger user Unknown wrote:

> Since you're implementing this game at least partly as a learning exercise, you may not be interested in using any other libraries. However, I do have a library for working with all kinds of grids/tiles for board games. You can install it using the command **cabal install grid**.

The comment included this GitHub link and a second link to the User Guide.

I am not able to determine the date this comment was left, but I think it may have been left quite a long time after the original post. I don't recall investigating the Grid library, but if I get back to this project, I should take a look.

Day 5: Array v. List

So, a little more progress in learning me a Haskell: I've managed to implement the board using an immutable array. There's good news and bad news here. If you're an old hand at functional programming, you probably know all this and more, but I needed to do a little thinking on purely functional data structures. I have not really been satisfied with the amount of code necessary to manage my 2-D board in a list. I spent some time doodling some possible alternative implementation before concluding that purely functional data structures — in which nodes are never mutated — are hard. Anything I might be accustomed to doing with double or multiply-linked lists is pretty much a washout, since you can't ever share structure. In fact, I think one data structure I came up with might not be constructible at all without being able to mutate links between nodes. So I'm starting to understand why the tutorials all advise me to stick with lists.

Nevertheless, this is a small problem, and efficiency is not my biggest concern, at least not in the learning phase. I wanted to figure out how to use an immutable array. The tutorials have not been very satisfying. They seem to assume that anything this trivial is too trivial to demonstrate. But here's what I did.

First, the type of an array in Haskell encodes the number of dimensions and the node type, but not the size. You set that when you call the constructor. Here's a 2-D array type for my board:

```
    type BoardArray = Array ( Int, Int ) Tile
```

I specified some bounds:

```
    max_row :: Int
    max_row = 3

    max_col :: Int
    max_col = 23
```

And I should point out one of the fundamental problems with using arrays: it's very easy to kill your program by exceeding the array bounds. There is a similar problem with **head**, but when writing functions with pattern-matching and guards there are pretty accepted conventions for dealing with empty lists. I suppose one could use guard patterns on all array accesses, but it starts to seem a little silly.

The next thing is that a given array works with some auxiliary types. The // operator takes an array and a list of tuples and builds a new array with updated content. The type of that list of tuples is this:

```
    type TileAssocList = [ ( ( Int, Int ), Tile ) ]
```

For accessing multiple items in an array, the **range** method builds lists of indexing tuples. The syntax to range requires tuples of tuples, with the parentheses piling up, so I wrapped it up in a function:

```
    make_2d_range :: Int -> Int -> Int -> Int -> [ ( Int, Int ) ]
    make_2d_range y0 x0 y1 x1 = range ( ( y0, x0 ), ( y1, x1 ) )
```

So how does that work? It just iterates coordinates, permuting from higher indices to lower, like so:

```
*Main> make_range 0 0 0 1
[(0,0),(0,1)]

*Main> make_range 0 0 1 3
[(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3)]
```

For this problem domain, I need to know how reversed ranges work. For example, when the penguin is facing west, I want to build a range and a list of tiles in reverse index order. Can range do that for me?

```
*Main> make_range 0 23 0 0
[]
```

Ah... no. I guess that would have been too easy. So I'll have to account for those cases specially. Here's a function to get the penguin's view out of a 2-D array of tiles, in the form of a tile association list I can use to create a freshly created "modified" array (it's not really modified, but a new one is created with the updates from that list applied):

```
view_array :: BoardArray -> Pos -> Dir -> TileAssocList
view_array board pos dir =
    let row = ( posY pos )
        col = ( posX pos )
        coord_list = case dir of
            East  -> if ( col == max_col )
                     then []
                     else make_2d_range row
                        ( col + 1 ) row max_col
            South -> if ( row == max_row )
                     then []
                     else make_2d_range
                        ( row + 1 ) col max_row col
            West ->  if ( col == 0 )
                     then []
                     else make_2d_range
                        row 0 row ( col - 1 )
            North -> if ( row == 0 )
                     then []
                     else make_2d_range
                        0 col ( row - 1 ) col
        tile_assoc = zip coord_list
            ( map ( (!) board ) coord_list )
    in case dir of
        East -> tile_assoc
        South -> tile_assoc
        West -> reverse tile_assoc
        North -> reverse tile_assoc
```

That's not so bad. The key to this function is the **!** operator — this gets a tuple and an array and returns an element — and I **zip** the elements up with their coordinate tuples. Note that a lot of the bulk of this function is handling the edge cases, because we don't want to apply an out-of-range coordinate tuple to **!**. There may still be a shorter, clearer implementation possible. By comparison, here's a list-of-lists version factored a bit using currying to make it as self-documenting as I could get it — note the use of **id** to let me return a general function as **orient**. I'm sure it doesn't impress FP whizzes, but I'm kinda proud of it — I feel like I'm starting to use Haskell a little more idiomatically:

```
view_list :: BoardList -> Pos -> Dir -> [Tile]
view_list board pos dir =
    let row = ( posY pos )
        col = ( posX pos )
        transposed = elem dir [ South, North ]
        reversed = elem dir [ West, North ]
        orient | reversed = reverse
               | otherwise = id
        trim = case dir of
            East -> drop ( col + 1 )
            South -> drop ( row + 1 )
            West -> take col
            North -> take row
        extract | transposed = ( transpose board ) !! col
                | otherwise = board !! row
    in orient $ trim $ extract
```

Testing **view_list**:

```
*Main> view_list init_board_list (Pos 0 0) East
[Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,
Empty,Empty,Tree,Empty,Empty,Empty,Empty,Empty,Ice_Block,Empty,Empty]

*Main> view_array init_board_array (Pos 0 0) East
[((0,1),Empty),((0,2),Empty),((0,3),Empty),((0,4),Empty),
((0,5),Empty),((0,6),Empty),((0,7),Empty),((0,8),Empty),
((0,9),Empty),((0,10),Empty),((0,11),Empty),((0,12),Empty),
((0,13),Empty),((0,14),Empty),((0,15),Tree),((0,16),Empty),
((0,17),Empty),((0,18),Empty),((0,19),Empty),((0,20),Empty),
((0,21),Ice_Block),((0,22),Empty),((0,23),Empty)]
```

Now we can write **step**. Here's the list version I've presented before:

```
    step_list :: [Tile] -> ( Bool, [Tile] )
    step_list [] = ( False, [] )
    step_list ts = if walkable (head ts) then ( True, ts )
                                         else ( False, collide ts )
```

The array version is a little more complicated, because I want to strip the list I pass to **collide** down to just a list of tiles, in order to retain that clean logic for dealing with just a list of tiles. So I unzip my coordinate tuples from my tiles, get a potentially updated tile list, and zip it back together. That complicates it a bit, like so:

```
    step_array :: TileAssocList -> ( Bool, TileAssocList )
    step_array [] = ( False, [] )
    step_array tile_assoc = if ( walkable $ head tile_list )
                            then ( True, tile_assoc )
                            else ( False, zip coord_list
                                          ( collide tile_list ) )
        where ( coord_list, tile_list ) = unzip tile_assoc
```

I'm going to have to uglify my nice **collide** method a bit because I need to return at least one additional value — indicating whether **collide** consumed a heart, so that we can keep score of the game.

Next up, you can see the array and list solutions start to diverge hugely. It's hard to merge the list-based board back together with the potentially updated tile list to create the next immutable list-based board. My original method was pretty hideous. With Jeff's refactoring it's still a lot of code. (Note: I don't have this completely working yet; I'm getting a run-time error about bad patterns I haven't quite figured out yet):

```
next_board_list :: BoardList -> Pos -> Dir ->
    ( Bool, BoardList )
next_board_list board pos dir =
    let ( penguin_could_move, updated_view_list ) =
        step_list $ view_list board pos dir
    in ( penguin_could_move, update_board_from_view_list
         board pos dir updated_view_list )

apply_view_list_to_row :: [Tile] -> Int -> Bool ->
    [Tile] -> [Tile]
apply_view_list_to_row orig pos True update =
    take ( pos + 1 ) orig ++ ( init update )
apply_view_to_row orig pos False update =
    ( reverse ( init update ) ) ++ ( drop pos orig )

apply_view_list_to_rows :: BoardList -> Int -> Int ->
    Bool -> [Tile] -> BoardList
apply_view_list_to_rows orig row pos is_forward update =
    take row orig ++
    nest ( apply_view_to_row
        ( orig !! row ) pos is_forward update ) ++
    drop ( row + 1 ) orig

update_board_from_view_list :: BoardList -> Pos -> Dir ->
    [Tile] -> BoardList

update_board_from_view_list board pos dir updated_view_list
    | is_eastwest = apply_view_list_to_rows board
                      ( posY pos ) ( posX pos )
                      is_forward updated_view_list
    | otherwise = transpose ( apply_view_list_to_rows
                            ( transpose board )
                            ( posX pos ) ( posY pos )
                            is_forward updated_view_list )
    where is_forward = elem dir [ East, South ]
          is_eastwest = elem dir [ East, West ]
```

By comparison, the array is much more suited to create an updated version of itself, given a list of elements to update. This is handled by the // function, in this simple function to create the next board in array form, called from **step_array**:

```
next_board_array :: BoardArray -> Pos -> Dir ->
    ( Bool, BoardArray )
next_board_array board pos dir =
    let ( penguin_could_move, updated_view ) =
        step_array $ view_array board pos dir
    in ( penguin_could_move, board // updated_view )
```

I like that — it looks like we're working with the data structure rather than against it, although the overhead to manage the ranges and lists still feels to me more complicated than it should be. That complexity carries over elsewhere: for example, pretty-printing the array requires that range logic again. In fact I wind up just wrapping up and re-using the logic to pretty-print the list, so you can see how much additional code I needed:

```
pretty_tiles :: [Tile] -> String
pretty_tiles [] = "\n"
pretty_tiles (t:ts) = case t of
                 Empty     -> "___"
                 Mountain  -> "mt "
                 House     -> "ho "
                 Ice_Block -> "ic "
                 Heart     -> "he "
                 Bomb      -> "bo "
                 Tree      -> "tr "
             ++ pretty_tiles ts

pretty_board_list :: BoardList -> String
pretty_board_list [] = ""
pretty_board_list (ts:tss) =
    pretty_tiles ts ++ pretty_board_list tss

split_tile_list :: [ Tile ] -> [ [ Tile ] ]
split_tile_list [] = []
split_tile_list ts = [ take tiles_in_row ts ] ++
                     ( split_tile_list $
                         ( drop tiles_in_row ) ts )
    where tiles_in_row = max_col + 1

pretty_board_array :: BoardArray -> String
pretty_board_array board = pretty_board_list split_tiles
    where full_range = make_2d_range 0 0 max_row max_col
          all_tiles = map ( (!) board ) full_range
```

```
split_tiles = split_tile_list all_tiles
```

As an aside, it seems like there ought to be at least one standard list split function, but it looks like folks don't really agree on how it should work.

So there it is — the array is kind of a mixed blessing here. I haven't done any large-scale profiling on it, to determine if the need to generate a whole new array each pass is a big loss, compared to the potential structure-sharing in the list implementation. It simplifies some of the code dramatically, while adding a layer of dealing with ranges and lists of tuples everywhere — as soon as we want to pull items out of the array, or merge them back in to a new array, we're dealing with lists again. Still, given the ugliness of the list merge code, it seems like the more natural choice for this kind of small game board data structure.

Day 5: Array v. List Comments

Blogger reader Jeff Licquia wrote:

> Took care of a little insomnia by fixing your list implementation. There are two problems.
>
> First: the bad patterns error comes from two places where **apply_view_to_row** wasn't renamed to account for the "_list_" implementation. A quick search-and-replace fixed that right up.
>
> After that, the list version "steals" tiles; each eastward move at the beginning chops a tile off the end, and eventually the moves collide with the shortened list and things blow up. I remembered you used to have **Edge** tiles, but decided to get rid of them, so I guessed that the "init update" calls in **apply_view_list_to_row** were for chopping off the **Edge** tile that's no longer there, so I just took out the "init" call. Either my hunch was right, or I'm just lucky. :-)
>
> After those two fixes, the list and array implementations seem to produce identical output. I know you're more a fan of the array version, but I want to see the all-important performance evaluation later, which won't happen with broken lists.
>
> I really like the new view functions. I still tend to be leery of the "case dir of…" thing happening over and over, but am getting tired again (yay! die, insomnia!), so not seeing the problem clearly.
>
> Also, the pretty printer stuff is screaming for something like a **fold** or **map**, but I'm not seeing how that improves things at the moment. Maybe some sleep will bring clarity.

Blogger reader Michael Alan Dorman said:

> I think (as something of a Haskell newbie myself, so take this with a grain of salt) what you want to be looking at is lenses to simplify your board manipulation code. http://www.youtube.com/watch?v=cefnmjtAolY is a presentation from Edward Kmett about them.

I replied to Jeff:

> Thanks, Jeff. Sorry to hear about the insomnia! But I'm happy for the help.
>
> Yes, when I got rid of Edge it broke a lot of things including my collide and slide methods, and that was not immediately apparent. I had to rewrite that extensively. I'm keeping the GitHub version up to date. But I'm not going to go back and revise all the blog posts as I go — that way lies madness :)

I replied to Michael:

> Michael, I am vaguely aware of lenses as something I would like to master… and I've still got to get a better handle on monads. I feel like I understand them when explained, but attempting to use one in my own code outside of **IO** has been a complete failure. I will take a look at that video.

Jeff added another comment:

> I know this is your learning exercise, not mine… but I ended up implementing scorekeeping with the **Writer** monad.
>
> Are you interested in seeing it, or do you not want the fun spoiled?

I replied to Jeff:

> Jeff, I'd love to see it! The feedback and comments are what makes this really valuable for me to blog about, rather than just writing it and keeping the code to myself. Maybe it will be the moment when I achieve monad enlightenment! If you like, send me an e-mail and I'll make it a real entry. Or actually if you have a Blogger account, I could enable you as a contributor on Blogger, at least I think I could.

I wrote:

> I am taking a sanity break from glowing screens over the Fourth of July Weekend — I'll get back to looking at this next week.

Blogger reader Cake wrote:

> Wow, I'm so happy I found your blog. I was trying to model a game in haskell recently (I thought it would be a good training), and I stumble on a lot of things you did, like Arrays.

I'll be sure to keep an eye on your articles now :)

Day 5 and a Half: Refactoring with a Monad

The job search has eaten my brain for the last few days — have I mentioned yet that I need a job? Oh, yes, I believe I may have — but I'm taking some time to press on with my Haskell larnin', especially since I've been getting great, helpful feedback.

The first thing I did was make some minor fixes to the list implementation, as suggested by Jeff. It's working now and my version looks like this:

```
next_board_list :: BoardList -> Pos -> Dir ->
    ( Bool, BoardList )

next_board_list board pos dir =
    let ( penguin_moved, updated_view_list ) =
        step_list $ view_list board pos dir
    in ( penguin_moved, update_board_from_view_list
         board pos dir updated_view_list )

apply_view_list_to_row :: [ Tile ] -> Int -> Bool ->
    [ Tile ] -> [Tile]

apply_view_list_to_row orig pos True update =
    take ( pos + 1 ) orig ++ update

apply_view_list_to_row orig pos False update =
    ( reverse update ) ++ ( drop pos orig )

apply_view_list_to_rows :: BoardList -> Int -> Int ->
    Bool -> [ Tile ]
    -> BoardList

apply_view_list_to_rows orig row pos is_forward update =
    take row orig ++
    nest ( apply_view_list_to_row ( orig !! row ) pos
           is_forward update ) ++
    drop ( row + 1 ) orig
    where nest xs = [xs]

update_board_from_view_list :: BoardList -> Pos ->
    Dir -> [ Tile ]
    -> BoardList
```

```
update_board_from_view_list board pos dir updated_view_list
    | is_eastwest = apply_view_list_to_rows board
                        ( posY pos ) ( posX pos )
                        is_forward updated_view_list
    | otherwise = transpose ( apply_view_list_to_rows
                                ( transpose board )
                                ( posX pos ) ( posY pos )
                                is_forward updated_view_list )
    where is_forward = elem dir [ East, South ]
          is_eastwest = elem dir [ East, West ]
```

Now, it turns out that Jeff did more than suggest a refactoring — he actually did something I haven't quite gotten my head around yet, which is to refactor my code to use a monad for managing some of this task. He forked my code in his own GitHub repo here and sent me some notes to share on my blog. Here's part of what he said:

> The way I got my head wrapped around monads was to think of them as "important stuff to do, but not the point." You need to do some housekeeping that's important, but it's not the reason you're writing this function. The classic example is division. You're writing a math library, and you need to implement division. Division by zero is something you need to deal with sanely, but it's not the point; you're writing the function because you want to divide by things that aren't zero. So, to handle the zero case, you return a **Maybe** instead of a simple number. Only now you can't just add numbers together with division, because you're dealing with Maybes, not numbers. So you end up implementing addition with Maybes, except that makes no sense, as adding never fails, and people using your math library get annoyed because now *they* have to deal with division-by-zero errors even when they're not dividing, and it's a mess. Except — **Maybe** is a monad. So you skip all that mess, implement division with a **Maybe**, everything else without, and use the cool monad and functor features of the language to bridge the gaps. The same pattern exists with scorekeeping. A lot of the functions in your code need to keep track of the score and occasionally award points, but scores aren't "the point" of, say, **collide**. And when you start thinking about all the places you need to worry about scores, you start seeing scorekeeping infect all kinds of weird places in your code. I think you even mentioned having to "uglify" your code with scorekeeping in your blog post.

Yes, yes, yes — mainly the chain of function invocations that handle generating the next board, down to the **collide** calls. Because it's only at the point where a heart disappears that we can decrement the heart count. Without state, I can't make this a global state. In a purely function form, I have to "thread" the indication that the heart count should be decreased through the whole chain of function signatures, which now all have to return an extra thing.

> So, minimize the ugly with monads. Just do what you need to do to pass around the score, and deal with it when it's appropriate. (In my implementation, that was in **next__world**). The **Writer** monad is perfect for the job. It uses a monoid, which is a fancy ways of saying "something that knows how to grow". Lists are monoids, because you can append to them. Numbers are monoids, because you can add and multiply them. And so on. What the **Writer** monad does is take care of the adding part. You just return the thing you're working with, and the monad tacks it on using the monoid. Specifically, with scorekeeping, you just note how many points each individual action takes, and the monad does the adding together. When you finally deal with the score in **next__world**, you get all the accumulated points in one tidy variable.

OK, cool… let's see what he came up with!

```
import Control.Monad.Writer

...

-- Keep track of the score with a writer monad
type ScoreTracker = Writer ( Sum Int )
```

OK, let me pause there and see if I can make sense of that. *Learn You a Haskell* says

> Whereas **Maybe** is for values with an added context of failure and the list is for non-deterministic values, the **Writer** monad is for values that have another value attached that acts as a sort of log value. **Writer** allows us to do computations while making sure that all the log values are combined into one log value that then gets attached to the result.

OK, I think I get that — in *Learn You* it is used for implementing logging, not scoring of a game, but it seems like it could be generalizable. The example given does this just kind of thing I was mentioning — makes a simple function return a tuple to pass both the actual interesting return value and the log string, or in our case I think we want a score. *Learn You* continues:

> When we were exploring the **Maybe** monad, we made a function **applyMaybe**, which took a **Maybe** a value and a function of type **a -> Maybe b** and fed that **Maybe** a value into the function, even though the function takes a normal a instead of a **Maybe a**. It did this by minding the context that comes with **Maybe a** values, which is that they are values with possible failure. But inside the a **->** **Maybe b** function, we were able to treat that value as just a normal

value, because **applyMaybe** (which later became »=) took care of checking if it was a **Nothing** or a **Just** value. In the same vein, let's make a function that takes a value with an attached log, that is, an **(a, String)** value and a function of type **a -> (b, String)** and feeds that value into the function. We'll call it **applyLog**. But because an **(a, String)** value doesn't carry with it a context of possible failure, but rather a context of an additional log value, **applyLog** is going to make sure that the log of the original value isn't lost, but is joined together with the log of the value that results from the function.

Oooh, again, that sounds very promising. So I'm convinced that **Writer** is the right abstraction here. The values that **Writer** gets are **Sum** and **Int** — **Sum** is our monoid, **Int** is a type we're going to use to accumulate the updated score. (To go along with the Polar game logic, I think there really should ultimately be two scores — one should be the heart count for a given board, which decrements, and gets tested against zero to indicate board completion, and the other should be a level, which increments as the player moves through the levels, but never mind that for now).

Jeff then came up with:

```
noscore :: a -> ScoreTracker a
noscore x = writer (x, Sum 0)

score :: a -> ScoreTracker a
score x = writer (x, Sum 1)
```

Two functions, **noscore** and **score**. I think these are both monadic **return** — injecting a value, passing it to the next step while applying the sum operation. So let's see how he uses it. here's my **slide** function:

```
slide :: [ Tile ] -> [ Tile ]
slide ( Ice_Block : ts ) | ( null ts ) ||
    ( blocking $ head ts ) = ( Ice_Block : ts )

slide ( t : Empty : ts ) =
    ( Empty : ( slide ( t : ts ) ) )

slide ( t : ts ) | ( null ts ) ||
    ( blocking $ head ts ) = collide ( t : ts )
```

I'm not going to take Jeff's current version, because he's restructured it a bit using guards, which obscures just the differences due to the use of the ScoreTracker, but here's a version that does the same thing. We don't have to explictly construct the return tuples:

```
slide' :: [ Tile ] -> ScoreTracker [ Tile ]

slide' ( Ice_Block : ts ) | ( null ts ) ||
        ( blocking $ head ts ) = noscore ( Ice_Block : ts )

    slide' ( t : Empty : ts ) =
        noscore ( Empty : ( slide ( t : ts ) ) )

    slide' ( t : ts ) | ( null ts ) || ( blocking $ head ts ) =
        collide ( t : ts )
```

And this doesn't actually compile. Note that **collide** doesn't handle the monad — the compiler warns us as Jeff described:

```
    Couldn't match expected type `ScoreTracker [Tile]'
                with actual type `[Tile]'
    In the return type of a call of `collide'
    In the expression: collide (t : ts)
    In an equation for slide':
        slide' (t : ts)
          | (null ts) || (blocking $ head ts) = collide (t : ts)
```

That seems pretty clear — so I have to fix it up the same way:

```
    collide' :: [ Tile ] -> ScoreTracker [ Tile ]

    collide' [] = noscore []

    collide' ( t : ts ) | fixed t =
        noscore ( t : ts )

    collide' ( Bomb : Mountain : ts) =
        noscore ( [ Empty, Empty ] ++ ts )

    collide' ( Heart : House : ts ) =
        score ( [ Empty, House ] ++ ts )

    collide' ( Ice_Block : ts ) | ( null ts ) ||
        ( blocking $ head ts ) = noscore ( Empty : ts )

    collide' ( t : ts ) | ( movable t ) && ( ( null ts ) ||
        ( blocking $ head ts ) ) = noscore ( t : ts )

    collide' ( t : Empty : ts ) | movable t =
        noscore ( Empty : ( slide( t : ts ) ) )
```

And **slide'** should call **collide'** instead of **collide**, of course. So once this is compiled and loaded into GHCI, we can play with it and compare it to the original **collide**:

```
*Main> :t collide'
collide' :: [Tile] -> ScoreTracker [Tile]
```

```
*Main> :t collide
collide :: [Tile] -> [Tile]
*Main> collide [ Bomb, Mountain ]
[Empty,Empty]
*Main> collide [ Heart, House ]
[Empty,House]
*Main> collide' [ Heart, House ]

:23:1:
    No instance for (Show (ScoreTracker [Tile]))
      arising from a use of `print'
    Possible fix:
      add an instance declaration for (Show (ScoreTracker [Tile]))
    In a stmt of an interactive GHCi command: print it
```

Er, yeah. The result is not printable, but can we see its type?

```
*Main> :t ( collide' [ Heart, House ] )
( collide' [ Heart, House ] ) :: ScoreTracker [Tile]
```

In fact, we can. So there might be an easy way to make the monadic type printable — **deriving (Show)** doesn't work — but first, how do we extract the values? Well, we get back the return value of the whole chain from **runWriter**:

```
*Main> runWriter $ collide' [Heart, House]
([Empty,House],Sum {getSum = 1})
```

What's the type? It's just a tuple:

```
*Main> :t ( runWriter $ collide' [Heart, House] )
( runWriter $ collide' [Heart, House] ) :: ([Tile], Sum Int)
*Main> fst $ runWriter $ collide' [Heart, House]
[Empty,House]
*Main> snd $ runWriter $ collide' [Heart, House]
Sum {getSum = 1}
```

Anyway, I think my mind is blown enough for today. I'm going to stop there. Jeff has made some other modifications to my code here and there — modifications that improve the clarity — but I'll have to get back to those. I'm off to read the monad tutorials again, and maybe understand them better this time!

Day 5 and a Half: Refactoring with a Monad Comments

Blogger reader Jeff Licquia wrote:

> Good luck with the job search. I hope something pans out soon.
>
> It occurs to me that I was a bit too clever in my code. When people talk about the **Maybe** monad, they usually talk about **Maybe a**, where **a** is something else. **Writer** is like that, but it takes two things:

> the monoid, and the "something else" like what **Maybe** takes. So, defining **ScoreTracker** might make more sense defined like this:

```
type ScoreTracker a = Writer (Sum Int) a
```

> That corresponds more closely to the **Maybe a** stuff that other tutorials talk about.
>
> I think you got there eventually, but it's worth keeping consistent, and maybe someone else won't make the leap.
>
> Ultimately, I think using **runWriter** is the best way to get the result at the **ghci** command line. In theory, we could use "instance" syntax to make **ScoreTracker** an instance of **Show**, but I couldn't get it to work, and for just doodling at the command line the **runWriter** form works fine.
>
> I'm actually teaching myself the **State** monad now. It's got its own level of weird beyond all the monadic goodness. I think we can make **Writer** work for the heart counter, too, but you mentioning "state" made me think of how to do this with the **State** monad. But that just makes my brain hurt, so we'll skip it for now. :-)

I replied:

> Jeff, **State** was the first one I tried to understand after how to use the basic **IO** monad, and I had a lot of trouble. This one seems easier! I am going over functors and monoids in *Learn You a Haskell*. I'm thinking about tricks that would show the progress of the **ScoreTracker** better on the command line. Please keep me posted with what you learn about **State**. I'm getting there — being able to just about understand how **Writer** works was very encouraging!

Blogger reader Matt Walton wrote:

> I always felt the **State** monad was a bit like cheating. "Oh we're in a language that doesn't have mutable state, so let's just pretend that we have it." You might as well run in **IO** and use actual mutables, says the part of my brain that conveniently ignores the real differences between pure monads and actual effects for the sake of being able to make dramatic statements.
>
> I never figured out **Writer**, but that was probably because I didn't need to.
>
> The last serious thing I wrote that used **State** was for my BSc dissertation, and **State** seemed highly appropriate at the time, because I was writing a compiler, and needed to carry a symbol table and other

such information around, and **State** made that very convenient.

Thus, one then presumes that using **State** to model, well, game state, makes a lot of sense at some level, even if it's not perhaps the most… hmm… Haskelly thing to do. Whatever 'Haskelly' means.

Day 5 and Three-quarters: a Bug Fix and liftM

Jeff Licquia has been playing further with the code and so have I. He discovered a bug in the version I posted in yesterday's installment (my bad). In slide' I neglected to call slide' in the recursive version of slide' but called the existing non-monadic slide. In other words, I had:

```
    slide' ( t : Empty : ts ) =
        noscore ( Empty : ( slide ( t : ts ) ) )
```

The problem here is that we'll add nothing to the accumulated score, and proceed into a chain of function invocations that handle ordinary lists. So the score increase that should happen at that point never happens:

```
*Main> runWriter $ collide' [Heart, House]
([Empty,House],Sum {getSum = 1})
*Main> runWriter $ collide' [Heart, Empty, House]
([Empty,Empty,House],Sum {getSum = 0})
```

Oops. Yeah, that's a bug. Note that the compiler can't catch this because it's doing what I've asked; there are not actually any type conflicts. The monadic **slide'** returns the type it is supposed to return, but in building up the "payload," the [**Tile**] part of **ScoreTracker** [**Tile**], the code fails to continue to build up the state. Let this be a lesson to me — leaving around the previous version of a function, when I'm testing a new one can be hazardous!

So, we can just fix that by calling **slide'**, right?

```
    slide' ( t : Empty : ts ) =
        noscore ( Empty : ( slide' ( t : ts ) ) )
```

Um, not so much:

```
arctic-slide.hs:52:49:
    Couldn't match expected type `[Tile]'
                with actual type `ScoreTracker [Tile]'
    In the return type of a call of slide'
    In the second argument of `(:)', namely `(slide' (t : ts))'
    In the first argument of `noscore', namely
      `(Empty : (slide' (t : ts)))'
```

Oh… yeah. There's that. We want to continue building up the monadic version of the list, but the **(:)** just takes a regular list. Now it's all complicated! But really there's a simple solution. I'll quote Jeff for a while, since he explained it so well to me. I have not quoted his code *exactly*, but the ideas are the same:

> ...the straightforward fix fails, because **slide'** returns a **ScoreTracker**, not a [**Tile**]. So the fix is a little more complicated. Since **slide'** returns pretty much exactly what we need, we can start with just that:

```
slide' ( t : Empty : ts ) = slide' ( t : ts )
```

> That's not quite right; we just dropped a tile. To get it back, remember that everything is a function, including the **:** operator *[and so it is easily composed with other functions — PRP]*. That means we can create a function that prepends an **Empty** element to a list...

```
prefix_empty :: [ Tile ] -> [ Tile ]
prefix_empty ts = Empty : ts
```

> So why would we do this? Because we need to take **ScoreTracker** into account. Here Haskell provides a function called **liftM**, which takes a normal function and "lifts" it into a monad. So:

```
prefix_empty_st :: ScoreTracker [ Tile ] -> ScoreTracker [ Tile ]
prefix_empty_st = liftM prefix_empty
```

> will give us a function with the type **ScoreTracker** [**Tile**] **->** **ScoreTracker** [**Tile**], which is what we want. (Technically, that's not true; it gives us **Monad m => m** [**Tile**] **-> m** [**Tile**]. But that's just a generic version of what we want, which works with **ScoreTracker**, **Maybe**, or lots of other monads).

So now we have this:

```
slide' ( t : Empty : ts ) = prefix_empty_st $ slide' ( t : ts )
```

Which doesn't use **score** or **noscore** — it just builds up the list, still in a monadic context, preserving whatever score changes might be applied by the function invocations it makes. And actually since we're not going to use the prefix functions elsewhere, they don't really earn their keep, and we can just write:

```
slide' ( t : Empty : ts ) = liftM ( Empty : ) $ slide' ( t : ts )
```

Note the partial application of **(:)** by binding it to only one parameter before we pass it to **liftM** — we're creating a new version of **(:)** that only takes one argument instead of two.

Jeff went on to identify a second bug, basically caused by the same problem in a **collide'** function also calling **slide** instead of **slide'**. A quick fix is to make that **collide'** function look like the **slide'** function we just fixed. But then, why not define one in terms of the other?

```
collide' ( t : Empty : ts ) | movable t =
    slide' ( t : Empty : ts )
```

Let's go back a bit and reconsider — when I was using a special value for **Edge**, the logic for **slide** and **collide** was considerably simpler (although it did not work right). Here it is today:

```
slide' :: [ Tile ] -> ScoreTracker [ Tile ]
slide' ( Ice_Block : ts ) | ( null ts ) ||
    ( blocking $ head ts ) = noscore ( Ice_Block : ts )
slide' ( t : Empty : ts ) =
    liftM ( Empty : ) $ slide' ( t : ts )
slide' ( t : ts ) | ( null ts ) || ( blocking $ head ts ) =
    collide' ( t : ts )

collide' :: [ Tile ] -> ScoreTracker [ Tile ]
collide' [] = noscore []
collide' ( t : ts ) | fixed t = noscore ( t : ts )
collide' ( Bomb : Mountain : ts) =
    noscore ( [ Empty, Empty ] ++ ts )
collide' ( Heart : House : ts ) =
    score ( [ Empty, House ] ++ ts )
collide' ( Ice_Block : ts ) | ( null ts ) ||
    ( blocking $ head ts ) = noscore ( Empty : ts )
collide' ( t : ts ) | ( movable t ) && ( ( null ts ) ||
    ( blocking $ head ts ) ) = noscore ( t : ts )
collide' ( t : Empty : ts ) | movable t =
    slide' ( t : Empty : ts )
```

Erm. I'd no longer call that elegant, beautiful code. For one thing, I have to wrap it brutally to fit into my Blogger text window. That's not just annoying when dealing with Blogger — it suggests that the lines are too long for easy reading even if they aren't wrapped. And here's what Jeff's version looks like today — he's implemented his own way to structure the code with guards:

```
slide :: [ Tile ] -> ScoreTracker [ Tile ]
slide [] = noscore []
slide ( t1 : t2 : ts )
  | t1 == Ice_Block && blocking t2 = noscore ( t1 : t2 : ts )
  | blocking t2 = collide ( t1 : t2 : ts )
  | otherwise = do
                  ts' <- slide ( t1 : ts )
                  return ( Empty : ts' )
slide ( t : ts )
  | t == Ice_Block = noscore ( t : ts )
  | otherwise = collide ( t : ts )

collide :: [ Tile ] -> ScoreTracker [ Tile ]
collide [] = noscore []

collide ( t1 : t2 : ts )
  | ( t1, t2 ) == ( Bomb, Mountain ) =
      noscore ( Empty : Empty : ts )
  | ( t1, t2 ) == ( Heart, House ) =
      score ( Empty : House : ts )
  | t1 == Ice_Block && blocking t2 =
      noscore ( Empty : t2 : ts )
  | movable t1 && blocking t2 =
      noscore ( t1 : t2 : ts )
  | movable t1 = do
                   ts' <- slide ( t1 : ts )
                   return ( Empty : ts' )
  | otherwise = noscore ( t1 : t2 : ts )

collide ( t : ts )
  | t == Ice_Block = noscore ( Empty : ts )
  | otherwise = noscore ( t : ts )
```

And I like that — using the separate functions for both slide and collide only to handle the **structurally** different versions — empty list, list with at least two items, list with at least one item — and the *guards* to handle when we differ by value. It is, I think, more readable than mine. I was a little freaked out by the use of **do** and **<-** in the middle of a function outside of main, but I'll think on that some more. I have not quite satisfied myself that it is perfectly correct, but then, I haven't really convinced myself that mine is correct either. So I have more to do on that front!

Day 5 and Three-quarters: a Bug Fix and liftM Comments

Blogger reader Jeff Licquia wrote:

> Yup, the **do** notation takes some getting used to. It's unfortunate that you have to learn it right away just to get things done, because it's easy to get the concepts wrong in the beginning and then have to "unlearn" them.
>
> We can convert those out of **do** notation. The first block in slide would look like this:

```
| otherwise = slide ( t1 : ts ) >>= \ts' -> return ( Empty : ts' )
```

> Or, we could do this:

```
| otherwise = liftM ( Empty : ) $ slide ( t1 : ts )
```

Look familiar? :-)

The first form is a direct translation from the **do** block, just how it's implemented internally. Thinking about how that works is a good way to start breaking out of the "**do** notation is for **IO** and **main**" rut. In particular, the idea that:

```
do
    x <- foo
    bar x
```

is just another way to say:

```
foo >>= \x ->
bar x
```

Essentially, that **<-** is creating a little anonymous function to be inserted into the chain of binds. Even though it looks like variable assignment in other languages, it's not.

Day 6: Towards a GUI

So, I have some time today to program and I want to see how far I can get in starting to develop a GUI for my game, incomplete as it is. Can I get a window displayed and reacting to mouse or keyboard events, and drive the game logic with it?

I came across the paper FranTk — A Declarative GUI Language for Haskell (PDF file link) by Meurig Sage and it looked interesting, so I considered trying FranTk. However, that led to broken links. Moving on…

Let's see if I can get somewhere with **FG**. That needs **gtk2hs**. Hmmm… **cabal update**, **cabal install gtk2hs-buildtools**, **cabal install gtk**.

```
[1 of 2] Compiling Gtk2HsSetup
    ( Gtk2HsSetup.hs, dist/setup-wrapper/Gtk2HsSetup.o )
[2 of 2] Compiling Main
    ( SetupMain.hs, dist/setup-wrapper/Main.o )
Linking dist/setup-wrapper/setup ...
Configuring cairo-0.12.4...
setup: The program pkg-config version >=0.9.0 is required but it
could not be found.
Failed to install cairo-0.12.4
```

I tried downloading pkg-config-0.28 source from here and that got me as far as running **./configure –prefix=/usr/local/** and seeing:

```
configure: error: Either a previously installed
pkg-config or "glib-2.0 >= 2.16" could not be found.
```

```
Please set GLIB_CFLAGS and GLIB_LIBS to the correct
values or pass --with-internal-glib to configure to use
the bundled copy.
```

So I tried **./configure –prefix=/usr/local/ –with-internal-glib** and that seemed to go OK; I was able to do **make**, **make check** — one failure out of 25 tests in "check-path" — and **sudo make install**. Back to **cabal install gtk** and… nope.

```
Configuring cairo-0.12.4...
setup: The pkg-config package cairo-pdf is required but it
could not be found.
Failed to install cairo-0.12.4

Configuring glib-0.12.4...
setup: The pkg-config package glib-2.0 is required but it
could not be found.
Failed to install glib-0.12.4
cabal: Error: some packages failed to install:
cairo-0.12.4 failed during the configure step. The exception was:
ExitFailure 1
gio-0.12.4 depends on glib-0.12.4 which failed to install.
glib-0.12.4 failed during the configure step. The exception was:
ExitFailure 1
gtk-0.12.4 depends on glib-0.12.4 which failed to install.
pango-0.12.4 depends on glib-0.12.4 which failed to install.
```

OK… so I guess it's time to install MacPorts because the Cairo page suggests using it to install cairo. I know there are competing tools — fink and Homebrew and I've used both of them at some point, years ago, but removed them, for reasons I can no longer remember… I think it had something to do with the way they insisted on installing things under **/opt** and it was clear to me if they would interfere with each other. But anyway, I'll try the MacPorts installer for 2.13 for Mountain Lion… and then sudo port install cairo… oh, wow, it's installing the universe… **bzip2**, **zlib**, **libpng**, **freetype**, **perl5**, **python27**… oh, the humanity…

OK, where are we… oh, **cabal install gtk** again. "The pkg-config package cairo-pdif is required but it could not be found." Let's try **glib** again.

```
Pauls-Mac-Pro:Gitit Wiki paul$ sudo port install glib2
--->  Computing dependencies for glib2
--->  Cleaning glib2
--->  Scanning binaries for linking errors: 100.0%
--->  No broken files found.
```

But **cabal install gtk** is still broken. Is there a MacPorts version of **gtk2**? Yes, apparently OH GOD IT'S BUILDING THE WHOLE WORLD…

(Musical interlude…)

But then **cabal install gtk** seems to go OK. A lot of deprecated function warnings. Another twenty minutes go by… what was I doing again? You know, I'm getting all confused, why don't I start with **gtk2hs** because *Real World Haskell* uses it… I need to **sudo port install glade3**… and OH GOD IT'S BUILDING THE WHOLE WORLD AGAIN… aaand welcome to hour three of "The Polar Game in Haskell, Day 6: Towards a GUI…"

OK, **glade** and **glade3** don't have any executables in my path. Oh, it's **glade-3**, how silly of me, even though the port is called **glade3**. And it says **Gtk-WARNING **: cannot open display:**. Oh yeah, it's X-Windows JUST SHOOT ME IN THE GODDAMN FACE… oh, I mean now I will happily go down another rabbit hole, thank you sir may I have another? So… the older X server is not supported in Mountain Lion anymore but there's something called XQuartz. **XQuartz-2.7.4.dmg**… "you need to log out and log back in to make XQuartz your default X11 server." Oh, thanks, I'll just close these FOURTEEN browser tabs, SEVEN **bash** terminal sessions, and other apps… you know, it's time for a food break anyway…

…aaand we're back. It launches, but I get "an error occurred while loading or saving configuration information for glade-3. Some of your configuration settings may not work properly." There's a "Details" button:

```
Failed to contact configuration server; the most common
cause is a missing or misconfigured D-Bus session bus daemon.
See http://projects.gnome.org/gconf/ for information. (Details -
1: Failed to get connection to session: Session D-Bus not running.
Try running `launchctl load -w
/Library/LaunchAgents/org.freedesktop.dbus-session.plist'.)
Failed to contact configuration server; the most common cause
is a missing or misconfigured D-Bus session bus daemon. See
http://projects.gnome.org/gconf/ for information. (Details -
1: Failed to get connection to session: Session D-Bus not running.
Try running `launchctl load -w
/Library/LaunchAgents/org.freedesktop.dbus-session.plist'.)
Failed to contact configuration server; the most common cause
is a missing or misconfigured D-Bus session bus daemon. See
http://projects.gnome.org/gconf/ for information. (Details -
1: Failed to get connection to session: Session D-Bus not running.
Try running `launchctl load -w
/Library/LaunchAgents/org.freedesktop.dbus-session.plist'.)
Failed to contact configuration server; the most common cause
is a missing or misconfigured D-Bus session bus daemon. See
http://projects.gnome.org/gconf/ for information. (Details -
1: Failed to get connection to session: Session D-Bus not running.
Try running `launchctl load -w
/Library/LaunchAgents/org.freedesktop.dbus-session.plist'.)
```

```
Failed to contact configuration server; the most common cause
is a missing or misconfigured D-Bus session bus daemon. See
http://projects.gnome.org/gconf/ for information. (Details -
1: Failed to get connection to session: Session D-Bus not running.
Try running `launchctl load -w
/Library/LaunchAgents/org.freedesktop.dbus-session.plist'.)
```

Gaaah! Well, OK, I can do that… and I'm able to edit a little file. Now to look at some tutorials. I get 404s on http://www.haskell.org/gtk2hs/docs/tutorial/glade/ and also http://dmwit.com/gtk2hs/%7C — ooof. My first attempt at adapting a little code from *Real World Haskell* — not going so well. This tutorial is still available: http://www.haskell.org/haskellwiki/Gtk2Hs/Tutorials/ThreadedGUIs but as to how useful it is… I'm gonna have to get back to you on that. There's also this tutorial: http://home.telfort.nl/sp969709/gtk2hs/chap2.html so I can create a little GTK GUI entirely in code rather than using a Glade file. Something like this:

```haskell
import qualified Graphics.UI.Gtk

main :: IO ()
main = do
    Graphics.UI.Gtk.initGUI
    window <- Graphics.UI.Gtk.windowNew
    Graphics.UI.Gtk.widgetShowAll window
    Graphics.UI.Gtk.mainGUI
```

Aaand I get an immediate segmentation fault. Hmmm. I think I read about running with "-threaded…"

```
Pauls-Mac-Pro:arctic-slide-haskell paul$ ghci -threaded
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Warning: -debug, -threaded and -ticky are ignored by GHCi
```

OK, how about GHC?

```
Pauls-Mac-Pro:arctic-slide-haskell paul$ ghc basic-gui.hs -
threaded
[1 of 1] Compiling Main             ( basic-gui.hs, basic-gui.o )
Linking basic-gui ...
Undefined symbols for architecture x86_64:
  "_iconv", referenced from:
      _hs_iconv in libHSbase-4.5.0.0.a(iconv.o)
     (maybe you meant: _hs_iconv_open,
_base_GHCziIOziEncodingziIconv_iconvEncoding6_info ,
_hs_iconv , _base_GHCziIOziEncodingziIconv_iconvEncoding4_closure , _base_GHCziIOziEncodingz
_base_GHCziIOziEncodingziIconv_iconvEncoding5_closure ,
_base_GHCziIOziEncodingziIconv_iconvEncoding6_closure ,
_base_GHCziIOziEncodingziIconv_iconvEncoding3_closure ,
_base_GHCziIOziEncodingziIconv_iconvEncoding2_closure ,
_base_GHCziIOziEncodingziIconv_iconvEncoding2_info ,
_base_GHCziIOziEncodingziIconv_iconvEncoding5_info ,
_base_GHCziIOziEncodingziIconv_iconvEncoding7_closure ,
_hs_iconv_close ,
_base_GHCziIOziEncodingziIconv_iconvEncoding7_info )
  "_iconv_close", referenced from:
      _hs_iconv_close in libHSbase-4.5.0.0.a(iconv.o)
     (maybe you meant: _hs_iconv_close)
  "_iconv_open", referenced from:
      _hs_iconv_open in libHSbase-4.5.0.0.a(iconv.o)
     (maybe you meant: _hs_iconv_open)
  "_locale_charset", referenced from:
      _localeEncoding in libHSbase-4.5.0.0.a(PrelIOUtils.o)
ld: symbol(s) not found for architecture x86_64
collect2: ld returned 1 exit status
```

Hmmm. It seems like this might take longer than I thought…

Day 6: Towards a GUI Comments

No comments were left on the Day 6: Towards a GUI post.

Day 6 and a Half: Towards a GUI, Continued

OK, so when I left off last time, I was running into a gruesome link error. I found this Stack Overflow thread and the first accepted answer fixed the problem. However, it seems that the answer may be to avoid MacPorts versions of the libraries I need. So I'm going to attempt to clean that all out and use Homebrew. So, first:

```
sudo port -fp uninstall --follow-dependents installed
```

And then I'm manually cleaning out some of the stuff mentioned in this article.

Next, I'm removing this from my **.profile** (hey, I'm so pleased that it is clearly marked!

```
# MacPorts Installer addition on 2013-07-16_at_11:57:13:
adding an appropriate PATH variable for use with MacPorts.
export PATH=/opt/local/bin:/opt/local/sbin:$PATH
# Finished adapting your PATH environment variable for
use with MacPorts.
```

Now to install Homebrew:

```
ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"
```

I ran **brew doctor** and wow, I have a mountain of warnings. I got rid of most of them except for several about "unbrewed" things — static libraries, **.la** files, and dylibs in **/usr/local/lib**. I get a warning about MacGPG2 but that seems to be fixed by upgrading to the current version. So now I'm trying **cabal install –reinstall gtk**, and I get:

```
Configuring gtk-0.12.4...
setup: The pkg-config package gthread-2.0 is required but it could not be
found.
```

And so, attempting to follow the directions here:

```
brew install glib cairo gtk gettext fontconfig
```

...and that actually crashes. I get "confest cannot be opened because of a problem." In the console log:

```
Process:         conftest [12844]
Path:            /private/tmp/*/conftest
Identifier:      conftest
Version:         0
Code Type:       X86-64 (Native)
Parent Process:  sh [12843]
User ID:         501

Date/Time:       2013-07-16 15:42:18.117 -0400
OS Version:      Mac OS X 10.8.4 (12E55)
Report Version:  10

Crashed Thread:  0

Exception Type:  EXC_BREAKPOINT (SIGTRAP)
Exception Codes: 0x0000000000000002, 0x0000000000000000

Application Specific Information:
dyld: launch, loading dependent libraries

Dyld Error Message:
  Library not loaded: /usr/local/lib/libintl.8.dylib
  Referenced from: /private/tmp/*/conftest
  Reason: no suitable image found.  Did find:
 /usr/local/lib/libintl.8.dylib:
           no matching architecture in universal wrapper
 /usr/local/lib/libintl.8.dylib:
           no matching architecture in universal wrapper
```

And I get an error about "GLib requires a 64 bit type." I also had to do some manual clean-out of some files that had the wrong permissions and were interfering with installing pkgconfig. I found a number of people reporting this problem, but none of the solutions they outlined seemed to work for me. So... what else can I try?

There's this:

http://www.haskell.org/haskellwiki/Gtk2Hs/Mac#GTK.2B_OS_X_Framewo
rk

OK! Deep sigh… let's try this!

```
Pauls-Mac-Pro:Downloads paul$ sh ./gtk-osx-build-setup.sh
Checking out jhbuild (07b5a7d) from git...
Cloning into 'jhbuild'...
remote: Counting objects: 37027, done.
remote: Compressing objects: 100% (14715/14715), done.
remote: Total 37027 (delta 28610), reused 28612 (delta 22178)
Receiving objects: 100% (37027/37027), 7.27 MiB | 2.27 MiB/s, done.
Resolving deltas: 100% (28610/28610), done.
Switched to a new branch 'stable'
Patch is empty.  Was it split wrong?
If you would prefer to skip this patch, instead run "git am --
skip".
To restore the original branch and stop patching run "git am --
abort".
Installing jhbuild...
gnome-autogen.sh not available
yelp-tools not available
Configuring jhbuild without autotools
Now type `make' to compile jhbuild
Installing jhbuild configuration...
Installing gtk-osx moduleset files...
PATH does not contain /Users/paul/.local/bin, it is recommended that you add that.

Done.
```

Ummm… OK, wow, that installed source in my home directory build tools in a *hidden* directory (prefaced with a period) under my home directory. There are warning notes about how the build process conflicts with MacPorts and fink. There's also a note that says "Note: jhbuild requires Python 2.5 to unpack tar files" (of course it does… that's the simplest and most system-compatible way to unpack tar files, right?) Ugh. Anyway… in ~**/Source/jhbuild** I type ~**/.local/bin/jhbuild bootstrap** and it builds about a bazillion things including **cmake**. (Talk amongst yourselves, this is going to take a while… time for another snack…)

That seemed to work. And so: ~**/.local/bin/jhbuild build meta-gtk-osx-bootstrap** and ~**/.local/bin/jhbuild build meta-gtk-osx-core**. Somewhat to my shock, everything succeeded! I tried to build gimp, but that failed with "we require Pango with the optional support for Cairo compiled in," and I don't want to go too far down that rabbit hole, so I gave up on that. So let's see if I can make that work with GHC. The next step is **package-config**. Which requires **glib**.

Ummm, wait a minute… oh, crap. That's still broken with homebrew. Ummm. What about ___package-confi___g from MacPorts, which the instructions for GTK OSX warned me about? Sure, let's try it, what the hell… after all, I've wasted nearly a full day already… so, **sudo port selfupdate**, **sudo port install pkg-config**… that seemed to work. So then we download the **Gtk2HS** tarball… ummm, the link from the instructions is broken. Ummm… from SourceForge here… but that version is looking much older than the one described here. I'm getting a bad feeling about this. But anyway… 0.10.1 it is! Configure away!

```
checking for pkg-config... /opt/local/bin/pkg-config
checking pkg-config is at least version 0.9.0... yes
checking for GLIB... no
configure: error:

The development files for the glib-2.x library were not found.
Perhaps you need to install glib or glib-devel
```

Huh. Well. It's just about the end of my work day; I've got to go downstairs and help my wife get dinner ready. Ummm. So! I hope you've enjoyed this tutorial on how to use the GTK GUI library in Haskell! Please join me next time when I perform brain surgery on myself using a hacksaw, a folding mirror, and a bottle of Scotch!

In the original blog post, I included this YouTube video of Robyn Hitchcock performing "Fix You" live, which Blogger embedded in the blog post page. I'm not sure how to do that in my own blog, so follow the link if you want to see Robyn, one of my favorite musicians, who I got to see live at The Ark in Ann Arbor in the late nineties.

Day 6 and a Half: Towards a GUI, Continued Comments

Blogger reader Jeff Licquia wrote:

> Hmm…

```
sudo apt-get install libghc-gtk-dev
cat > test-gtk.hs
ghc test-gtk.hs
./test-gtk
```

> And there's a blank window on my screen.
>
> Sorry… but it's not often that a Linux fanboy gets to wag his finger at Apple for not getting something right, so we have to enjoy our moments of glory when we find them. :-)
>
> Seriously, a few options:
>
> - Not Haskell-specific, but of the GUI toolkits I saw with Haskell stuff,

> something based on wxWidgets is likely your best bet. It will support native Mac OS X look-n-feel, and also port easily to Windows and Linux via GTK+. (And you can do GTK+ on Mac, but for some reason I don't suspect you're eager to try that option.)
>
> - Qt would be a very close second. It's got a proven pedigree on Mac, too. The one drawback: things in the Qt world are in flux, which means that slightly older Qt code may require a little attention to get working with the latest shiny. wxWidgets is likely to have a slight edge in that department.
>
> - After that, Tk has a very long reputation as a cross-platform GUI toolkit. The first GUI systems written for Python and Perl were based on Tk. That said, the link from the Haskell wiki to HTk seems dead, so maybe not.
>
> - Something to consider: if you've got something like Parallels or VMWare or VirtualBox on your Mac, the path of least resistance might be to fire up an Ubuntu VM and do the tinkering in Linux. It probably could have saved you time today. OTOH, you could grow a big pain-point when it comes time to transfer your tinkerings back.

Looking back from 2025, Jeff's suggestion to build on Linux is a good one; in 2013 I had been a Linux user for many years. I had used Qt a decade earlier, and a few years later would use it again. In 2025, I still work primarily on Mac and Linux, although I use Windows laptops when required by employers. I mainly use a Mac laptop as my daily driver and a number of Linux boxes on my home network, including the one I'm connected to right now, using VSCode, to do my writing in Markdown and building my web site with a Makefile.

Blogger reader Matt Walton wrote:

> It's not really a Haskell problem, the problem is GTK on a Mac.
>
> May I suggest wxHaskell instead? It's based on wxWidgets which has an OS X backend, so should be a lot easier to install. There's also some interesting stuff going on around it, like the reactive-banana bindings for it that let you use FRP to wire up your Wx GUI's functionality.

I commented:

> Thanks for reading, guys. I realize it turned out to be a pretty useless set of blog posts. I would have preferred if I'd been able to get something working and present a recipe for other people to follow rather than just a story of failure!
>
> I do have a quad-core Xeon Ubuntu box sitting here. I work on

> various platforms including Windows (when I have to). The main thing that keeps me from firing it up is that we are having a heat wave and my office is in a half-attic. The idea of firing up another computer in here fills me with dread. I fried one mobo *[motherboard]* last summer even with an air conditioner in here and I'm not eager to fry another. But I'm glad to hear it is maintained well enough to work pretty effortlessly on Linux. I'm not sure just what the issue is with the Mac libraries but it seems like it is affecting more than just this. It certainly could be because I have tinkered with this system quite a bit and built a lot of things from scratch. But judging from various comments on message boards out there, I suspect it is not.
>
> I will give wxWindows a shot and see how far I can get with that. I don't remember it fondly from attempting to write a cross-platform GUI a decade ago — the code was pretty ugly then and I had to bail out and use Qt instead — but most likely it has improved.

Then later, I added:

> Just FYI, wxHaskell is working for me, at least for the most basic Hello World app. So maybe I'll have some more significant progress to report soon!

Day 7: Towards a GUI, Continued

So, trying wxHaskell. First, I want to try removing everything that might be left over from yesterday's experiments:

```
Pauls-Mac-Pro:~ paul$ brew list
gettext  libffi  pkg-config xz
Pauls-Mac-Pro:~ paul$ brew uninstall gettext libffi pkg-config xz
Uninstalling /usr/local/Cellar/gettext/0.18.3...
Uninstalling /usr/local/Cellar/libffi/3.0.13...
Uninstalling /usr/local/Cellar/pkg-config/0.28...
Uninstalling /usr/local/Cellar/xz/5.0.5...
Pauls-Mac-Pro:~ paul$ port installed
The following ports are currently installed:
  libiconv @1.14_0 (active)
  pkgconfig @0.28_0 (active)
Pauls-Mac-Pro:~ paul$ sudo port uninstall pkgconfig libiconv
--->  Deactivating pkgconfig @0.28_0
--->  Cleaning pkgconfig
--->  Uninstalling pkgconfig @0.28_0
--->  Cleaning pkgconfig
--->  Deactivating libiconv @1.14_0
--->  Cleaning libiconv
```

```
--->  Uninstalling libiconv @1.14_0
--->  Cleaning libiconv
```

Then install **wxmac**: I'm attempting the directions here.

brew install wxmac –use-llvm –devel

```
brew install wxmac --devel
Warning: It appears you have MacPorts or Fink installed.
Software installed with other package managers causes known problems
for Homebrew. If a formula fails to build, uninstall MacPorts/Fink
and try again.
```

There shouldn't be any libraries or binaries in the various paths to interfere, so I'll ignore this.

And, it seemed to succeed. So, next step from the instructions above: check your path to make sure you are using your wxWidgets and not the default Mac one. The command **which wx-config** should not return the file path **/usr/bin/wx-config** (On my system it returns **/usr/local/bin/wx-config**). Next, **cabal install wx cabal-macosx**. That chugs away for a while and I see an unnervingly large number of warnings, but it builds. And then, I saved this file as hello-ex.hs and **ghc –make HelloWorld.hs** and **macosx-app hello-wx** and **./hello-wx.app/Contents/MacOS/hello-wx** and the result runs and I get a window, although it pops up off the bottom of my primary display, and the application's main menu does not seem to render its menu items quite right (they say "Hide H" and "Quit H" instead of the application name). But still — promising!

So — some code. To facilitate working with a GUI module in a separate **.hs** file I am now calling the core logic **ArcticSlideCore.hs**. and that file begins with **module ArcticSlideCore where**. I don't have very much working yet, but here's what is in my **ArcticSlideGui.hs** file so far. First I define my module and do my imports:

```
module Main where
import Graphics.UI.WX
import ArcticSlideCore
```

Then I define some bitmaps. For purposes of experimentation I made **.png** files out of the original Polar game's CICN resources. I want to redraw them — first, to avoid blatant copyright infringement and second, to make them bigger. But temporarily, I'll just use the originals:

```
bomb = bitmap "bomb.png"
heart = bitmap "heart.png"
house = bitmap "house.png"
ice = bitmap "ice_block.png"
tree = bitmap "tree.png"
```

While they are not game tiles as such, there are icons for the penguin facing in the four cardinal directions and icons for a breaking ice block and exploding bomb that were used in original animations, so I have borrowed those:

```
    penguin_e = bitmap "penguin_east.png"
    penguin_s = bitmap "penguin_south.png"
    penguin_w = bitmap "penguin_west.png"
    penguin_n = bitmap "penguin_north.png"

    ice_break = bitmap "ice_block_breaking.png"
    bomb_explode = bitmap "bomb_exploding.png"
```

I noticed that wxHaskell's **Point** type operates in reverse. I'm accustomed to C arrays where the higher-rank indices come first (so y, x or row index, column index for tile positions), but points are backwards. My icons are 24x24 pixels, so I rearrange and scale them like so:

```
    posToPoint :: Pos -> Point
    posToPoint pos = ( Point ( posX pos * 24 ) ( posY pos * 24 ) )
```

Now, some convenience function for drawing bitmaps based on **Tile** type or based on a wxHaskell bitmap. These are two more cases where I was not sure of the type signature, so I wrote the functions without them:

```
    drawBmp dc bmp pos = drawBitmap dc bmp point True []
        where point = posToPoint pos

    drawTile dc tile pos = drawBmp dc bmp pos
        where bmp = case tile of Bomb  -> bomb
                                 Heart -> heart
                                 House -> house
                                 Ice   -> ice
                                 Tree  -> tree
```

GHCI says:

```
Prelude Main> :t drawBmp
drawBmp
  :: Graphics.UI.WXCore.WxcClassTypes.DC a
     -> Graphics.UI.WXCore.WxcClassTypes.Bitmap ()
     -> ArcticSlideCore.Pos
     -> IO ()
```

That boils down to **drawBmp :: DC a -> Bitmap () -> Pos -> IO ()**, and the signature for DrawTile similarly boils down to **drawTile :: DC a -> Tile -> Pos -> IO ()**. Thanks, GHCI!

Next, I need a view method. This is just a placeholder test to verify that I can draw all my icons in the bounds where I expect them:

```
draw dc view
    = do
        drawTile dc Bomb        ( Pos 0 0  )
        drawTile dc Heart       ( Pos 0 1  )
        drawTile dc House       ( Pos 0 2  )
        drawTile dc Ice         ( Pos 0 3  )
        drawTile dc Tree        ( Pos 0 4  )
        drawBmp dc penguin_e    ( Pos 1 0  )
        drawBmp dc penguin_s    ( Pos 1 1  )
        drawBmp dc penguin_w    ( Pos 1 2  )
        drawBmp dc penguin_n    ( Pos 1 3  )
        drawBmp dc ice_break    ( Pos 0 23 )
        drawBmp dc bomb_explode ( Pos 3 23 )
```

Now, my **gui** function is where things get interesting and wxHaskell shows off a little. I read this paper that talks about some of the layout options and other tricks of the wxHaskell implementation, and discovered that this maps really nicely to defining my window in terms of a grid of icons. **space 24 24** returns a layout item of the appropriate size, and **grid** returns a layout item when given spacing values (I want the icons touching, so I use 0 0) and a list of lists for rows and columns. To generate the proper structure of 4 rows of 24 columns I just take what I need from infinite lists: **take 4 $ repeat $ take 24 $ repeat $ space 24 24** Oh, that's nifty!

```
gui :: IO ()
gui
    = do f <- frame [text := "Arctic Slide"]
         t <- timer f [ interval := 250
                      ]
         set f [ layout   := grid 0 0 $ take 4 $ repeat $
                             take 24 $ repeat $ space 24 24
                ,bgcolor  := white
                ,on paint := draw
               ]
         return ()
```

And finally, main:

```
main :: IO ()
main
  = start gui
```

To build this for use as a MacOS X GUI app I just do **ghc −make ./arctic-SlideGui.hs**, and if it compiles properly then **macosx-app arcticSlideGui; ./arcticSlideGui.app/Contents/MacOS/arcticSlideGui** and I have a little GUI window:

Sweet! Now I've got some more thinking to do. There's some plumbing that needs to get hooked up between the core game logic and the GUI layer. The

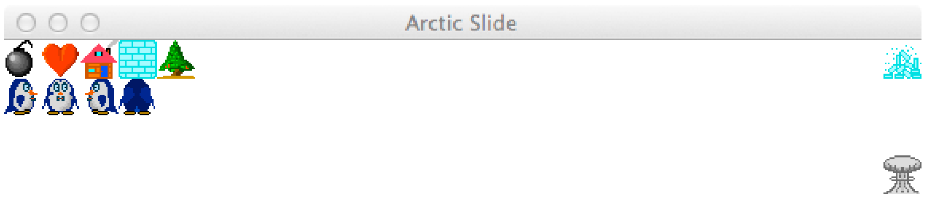

Figure 3: "Polar GUI Window"

core game logic is mostly factored the way I want it to be — it gets a world and a penguin move and returns an updated world — but I need to do a little more than just map the tiles to a series of **drawTile** calls. I might want to support timed sequences of changes to the GUI representation of the board — for example, smooth sliding of game pieces and smooth walking of the penguin. The **draw** method should draw the board pieces and the penguin all at once, with no redundancy if possible. Sound effects would be nice. Animation for crushing an ice block and blowing up a mountain would be nice. I've got some ideas along these lines based on event queues and a timer, and some other pieces of sample code I've been looking at.

Meanwhile, if any of you would like to take a crack at redrawing the graphics, please be my guest. It would be nice if the replacement icons would fit on an iPhone or iPod Touch. 48x84 is just a little bit too big — 48 pixels by 24 icons is 1152 pixels, and the iPhone 4 and 5 screens are 640x960 and 640x1136. 40 pixels wide would fit perfectly on an iPhone 4. Note that the icons don't actually have to be square — there is room to spare vertically. It might be nice, though, to leave room for a few extra rows, to support game boards that break out of the original 4-row height.

Day 7: Towards a GUI, Continued Comments

Blogger reader Jeff Licquia wrote:

> Excellent!
>
> Meanwhile, I need to push my refactor of scoring and penguin movement tracking that uses the **State** monad, so you can see if you prefer it to the current **Writer** monad logic. The nice thing is that we can add more state variables easily (like the heart counter you were talking about earlier). Plus, I think I did something... not *wrong*, really, but not quite *right*, either, when doing the **Writer** implementation, and I think the **State** version is actually more clear.

Blogger reader Matt Walton wrote:

> Brilliant! Glad to see some success in getting the GUI up and running.

> I'm interested in the animations, because in my limited experience that kind of thing's a pain no matter what language you're working in. At least in Haskell you should be able to build a decent abstraction for it if you can't find one.

And, sadly, that was the end.

I'm not quite sure what happened next in 2013. It's been a minute, and if I have more notes, I'm not sure where they are. But it doesn't look like I did much more with Haskell at that time. I was looking for a job, so probably got busy with job search stuff again.

I did, fortunately, wind up getting work again soon, a contract job testing a medical device, that required some travel to Fort Wayne, Indiana but allowed me to work mostly from home, which at the time was in Saginaw, Michigan.

As I write this in 2025, from Pittsfield Township, Michigan, I am unemployed again; most recently, I was working remotely with a virtual platform team at Boeing, as a contractor, and was laid off with all the rest of the team's contractors last October as the company cut costs in response to the Boeing machinist's strike.

As I look for work in 2025 I am attempting to improve my online portfolio, and editing this content and migrating it from Blogger to my personal web site represents a small piece of that.

I'm also looking into languages I might like to work with in the future, including Zig and OCaml. Although I find Haskell fascinating, and feel that using it has taught me a lot, I can't shake the feeling that Haskell, a language designed to be all things to all people — as long as those people are Ph.D. computer scientists with a working knowledge of category theory who want to write Haskell code for their dissertation projects — is never really going to be my jam; maybe I'd rather work with languages that take the best parts of Haskell and make them more simple and practical to use for systems and application programmers, the same way that Dylan took some of the best parts of Common Lisp and made them more accessible. What is that language in 2025, that supports the advanced programming idioms that I love, and that I can also get paid to write? I still don't know.
