Let vs. Where

From HaskellWiki

Haskell programmers often wonder whether to use
`let`
or
`where`
.

This seems to be only a matter of taste in the sense of "Declaration vs. expression style", however there is more to it.

It is important to know that
`let ... in ...`
is an expression, that is, it can be written wherever expressions are allowed. In contrast,
`where`
is bound to a surrounding syntactic construct, like the pattern matching line of a function definition.

Contents

- 1 Advantages of let
- 2 Advantages of where
- 3 Lambda Lifting
- 4 Problems with where

1 Advantages of let

Suppose you have the function

```
f :: s -> (a,s)
f x = y
   where y = ... x ...
```

and later you decide to put this into the
`Control.Monad.State`
monad.

However, transforming to

```
f :: State s a
f = State $ \x -> y
   where y = ... x ...
```

will not work, because
`where`
refers to the pattern matching
`f =`
, where no
`x`
is in scope. In contrast, if you had started with
`let`
, then you wouldn't have trouble.

```
f :: s -> (a,s)
f x =
   let y = ... x ...
   in  y
```

This is easily transformed to:

```
f :: State s a
f = State $ \x ->
   let y = ... x ...
   in  y
```

2 Advantages of where

Because "where" blocks are bound to a syntactic construct, they can be used to share bindings between parts of a function that are not syntactically expressions. For example:

```
f x
  | cond1 x   = a
  | cond2 x   = g a
  | otherwise = f (h x a)
  where
    a = w x
```

In expression style, you might use an explicit
`case`
:

```
f x
  = let a = w x
    in case () of
         _ | cond1 x   -> a
           | cond2 x   -> g a
           | otherwise -> f (h x a)
```

or a functional equivalent:

```
f x =
   let a = w x
   in  select (f (h x a))
          [(cond1 x, a),
           (cond2 x, g a)]
```

or a series of if-then-else expressions:

```
f x
```

```
  = let a = w x
    in if cond1 x
       then a
       else if cond2 x
       then g a
       else f (h x a)
```

These alternatives are arguably less readable and hide the structure of the function more than simply using

```
where
```

.

3 Lambda Lifting

One other approach to consider is that let or where can often be implemented using lambda lifting and let floating, incurring at least the cost of introducing a new name. The above example:

```
f x
  | cond1 x   = a
  | cond2 x   = g a
  | otherwise = f (h x a)
  where
    a = w x
```

could be implemented as:

```
f x = f' (w x) x

f' a x
  | cond1 x   = a
  | cond2 x   = g a
  | otherwise = f (h x a)
```

The auxiliary definition can either be a top-level binding, or included in f using

```
let
```

or

```
where
```

.

4 Problems with where

If you run both

```
fib = (map fib' [0 ..] !!)
    where
      fib' 0 = 0
      fib' 1 = 1
      fib' n = fib (n - 1) + fib (n - 2)
```

and

```
fib x = map fib' [0 ..] !! x
    where
      fib' 0 = 0
```

```
      fib' 1 = 1
      fib' n = fib (n - 1) + fib (n - 2)
```
you will notice that the second one runs considerably slower than the first. You may wonder why simply adding an explicit argument to
`fib`
(known as eta expansion) degrades performance so dramatically. You might see the reason better if you rewrote this code using
`let`
.

Compare

```
fib =
    let fib' 0 = 0
        fib' 1 = 1
        fib' n = fib (n - 1) + fib (n - 2)
    in  (map fib' [0 ..] !!)
```

and

```
fib x =
    let fib' 0 = 0
        fib' 1 = 1
        fib' n = fib (n - 1) + fib (n - 2)
    in  map fib' [0 ..] !! x
```
In the second case,
`fib'`
is redefined for every argument
`x`
. The compiler cannot know whether you intended this -- while it increases time complexity it may reduce space complexity. Thus it will not float the definition out from under the binding of x.

In contrast, in the first function,
`fib'`
can be moved to the top level by the compiler. The
`where`
clause hid this structure and made the application to
`x`
look like a plain eta expansion, which it is not.

- Haskell-Cafe on Eta-expansion destroys memoization? (http://www.haskell.org/pipermail/haskell-cafe/2010-October/084538.html)