Haskell/Existentially quantified types

Existential types, or 'existentials' for short, are a way of 'squashing' a group of types into one, single type.

Existentials are part of GHC's *type system extensions*. They aren't part of Haskell98, and as such you'll have to either compile any code that contains them with an extra command-line parameter of `-XExistentialQuantification`, or put `{-# LANGUAGE ExistentialQuantification #-}` at the top of your sources that use existentials.

The `forall` keyword

The `forall` keyword is used to explicitly bring type variables into scope. For example, consider something you've innocuously seen written a hundred times so far:

Example: A polymorphic function

```
map :: (a -> b) -> [a] -> [b]
```

But what are these `a` and `b`? Well, they're type variables, you answer. The compiler sees that they begin with a lowercase letter and as such allows any type to fill that role. Another way of putting this is that those variables are 'universally quantified'. If you've studied formal logic, you will have undoubtedly come across the quantifiers: 'for all' (or $\forall$) and 'exists' (or $\exists$). They 'quantify' whatever comes after them: for example, $\exists x$ means that whatever follows is true for at least one value of *x*. $\forall x$ means that what follows is true for every *x* you could imagine. For example, $\forall x,\, x^2 \geq 0$ and $\exists x,\, x^3 = 27$.

The `forall` keyword quantifies *types* in a similar way. We would rewrite `map`'s type as follows:

Example: Explicitly quantifying the type variables

```
map :: forall a b. (a -> b) -> [a] -> [b]
```

So we see that for any `a` and `b` we can imagine, `map` takes the type `(a -> b) -> [a] -> [b]`. For example, we might choose `a = Int` and `b = String`. Then it's valid to say that `map` has the type `(Int -> String) -> [Int] -> [String]`. We are *instantiating* the general type of `map` to a more specific type.

However, in Haskell, as we know, any use of a lowercase type implicitly begins with a `forall` keyword, so the two type declarations for `map` are equivalent, as are the declarations below:

Example: Two equivalent type statements

```
id :: a -> a
id :: forall a . a -> a
```

What makes life really interesting is that you can override this default behaviour by explicitly telling Haskell where the `forall` keyword goes. One use of this is for building **existentially quantified types**, also known as existential types, or simply existentials. But wait... isn't `forall` the *universal* quantifier? How do you get an existential type out of that? We look at this in a later section. However, first, let's dive right into the deep end by seeing an example of the power of existential types in action.

Example: heterogeneous lists

The premise behind Haskell's typeclass system is grouping types that all share a common property. So if you know a type instantiates some class `C`, you know certain things about that type. For example, `Int` instantiates `Eq`, so we know that elements of `Int` can be compared for equality.

Suppose we have a group of values, and we don't know if they are all the same type, but we do know they all instantiate some class, i.e. we know all the values have a certain property. It might be useful to throw all these values into a list. We can't do this normally because lists are homogeneous with respect to types: they can only contain a single type. However, existential types allow us to loosen this requirement by defining a 'type hider' or 'type box':

Example: Constructing a heterogeneous list

```
data ShowBox = forall s. Show s => SB s

heteroList :: [ShowBox]
heteroList = [SB (), SB 5, SB True]
```

We won't explain precisely what we mean by that datatype definition, but its meaning should be clear to your intuition. The important thing is that we're calling the constructor on three values of different types, and we place them all into a list, so we must end up with the same type for each one. Essentially this is because our use of the `forall` keyword gives our constructor the type `SB :: forall s. Show s => s -> ShowBox`. If we were now writing a function to which we intend to pass `heteroList`, we couldn't apply a function such as `not` to the values inside the `SB`, for their type might not be `Bool`. But we do know something about each of the elements: they can be converted to a string via `show`. In fact, that's pretty much the only thing we know about them.

Example: Using our heterogeneous list

```
instance Show ShowBox where
 show (SB s) = show s        -- (*) see the comment in the text below

f :: [ShowBox] -> IO ()
f xs = mapM_ print xs

main = f heteroList
```

Let's expand on this a bit more. In the definition of `show` for `ShowBox` – the line marked with `(*) see the comment in the text below` – we don't know the type of `s`. But as we mentioned, we *do* know that the type is an instance of Show due to the constraint on the `SB` constructor. Therefore, it's legal to use the function `show` on `s`, as seen in the right-hand side of the function definition.

As for `f`, recall the type of print:

Example: Types of the functions involved

```
print :: Show s => s -> IO () -- print x = putStrLn (show x)
mapM_ :: (a -> m b) -> [a] -> m ()
mapM_ print :: Show s => [s] -> IO ()
```

As we just declared `ShowBox` an instance of `Show`, we can print the values in the list.

Explaining the term *existential*

Let's get back to the question we asked ourselves a couple of sections back. Why are we calling these existential types if `forall` is the universal quantifier?

Since you can get existential types with `forall`, Haskell forgoes the use of an `exists` keyword, which would just be redundant.

Firstly, `forall` really does mean 'for all'. One way of thinking about types is as sets of values with that type, for example, Bool is the set {True, False, ⊥} (remember that bottom, ⊥, is a member of every type!), Integer is the set of integers (and bottom), String is the set of all possible strings (and bottom), and so on. `forall` serves as an *intersection* over those sets. For example, `forall a. a` is the intersection over all types, which must be {⊥}, that is, the type (i.e. set) whose only value (i.e. element) is bottom. Why? Think about it: how many of the elements of Bool appear in, for example, String? Bottom is the only value common to all types.

A few more examples:

1. `[forall a. a]` is the type of a list whose elements all have the type `forall a. a`, i.e. a list of bottoms.
2. `[forall a. Show a => a]` is the type of a list whose elements all have the type `forall a. Show a => a`. The Show class constraint limits the sets you intersect over (here we're only intersecting over instances of Show), but $\perp$ is still the only value common to all these types, so this too is a list of bottoms.
3. `[forall a. Num a => a]`. Again, the list where each element is a member of all types that instantiate Num. This could involve numeric literals, which have the type `forall a. Num a => a`, as well as bottom.
4. `forall a. [a]` is the type of the list whose elements have some (the same) type a, which can be assumed to be any type at all by a callee (and therefore this too is a list of bottoms).

We see that most intersections over types just lead to combinations of bottoms in some ways, because types don't have a lot of values in common.

Recall that in the last section, we developed a heterogeneous list using a 'type hider'. Ideally, we'd like the type of a heterogeneous list to be `[exists a. a]`, i.e. the list where all elements have type `exists a. a`. This '`exists`' keyword (which isn't present in Haskell) is, as you may guess, a *union* of types, so that `[exists a. a]` is the type of a list where all elements could take any type at all (and the types of different elements needn't be the same).

But we got almost the same behaviour above using datatypes. Let's declare one.

Example: An existential datatype

```
data T = forall a. MkT a
```

This means that:

Example: The type of our existential constructor

```
MkT :: forall a. a -> T
```

So we can pass any type we want to `MkT` and it'll convert it into a T. So what happens when we deconstruct a `MkT` value?

Example: Pattern matching on our existential constructor

```
foo (MkT x) = ... -- what is the type of x?
```

As we've just stated, `x` could be of any type. That means it's a member of some arbitrary type, so has the type `x :: exists a. a`. In other words, our declaration for T is isomorphic to the following one:

Example: An equivalent version of our existential datatype (pseudo-Haskell)

```
data T = MkT (exists a. a)
```

And suddenly we have existential types. Now we can make a heterogeneous list:

Example: Constructing the hetereogeneous list

```
heteroList = [MkT 5, MkT (), MkT True, MkT map]
```

Of course, when we pattern match on `heteroList` we can't do anything with its elements[1], as all we know is that they have some arbitrary type. However, if we are to introduce class constraints:

Example: A new existential datatype, with a class constraint

```
data T' = forall a. Show a => MkT' a
```

Which is isomorphic to:

Example: The new datatype, translated into 'true' existential types

```
data T' = MkT' (exists a. Show a => a)
```

Again the class constraint serves to limit the types we're unioning over, so that now we know the values inside a `MkT'` are elements of some arbitrary type *which instantiates Show*. The implication of this is that we can apply `show` to a value of type `exists a. Show a => a`. It doesn't matter exactly which type it turns out to be.

Example: Using our new heterogenous setup

```
heteroList' = [MkT' 5, MkT' (), MkT' True, MkT' "Sartre"]
main = mapM_ (\(MkT' x) -> print x) heteroList'

{- prints:
5
()
True
"Sartre"
-}
```

To summarise, the interaction of the universal quantifier with datatypes produces existential types. As most interesting applications of `forall`-involving types use this interaction, we label such types 'existential'. Whenever you want existential types, you must wrap them up in a datatype constructor, they can't exist "out in the open" like with `[exists a. a]`.

Example: `runST`

One monad that you may not have come across so far is the ST monad. This is essentially a more powerful version of the `State` monad: it has a much more complicated structure and involves some more advanced topics. It was originally written to provide Haskell with IO. As we mentioned in the Understanding monads chapter, IO is basically just a State monad with an environment of all the information about the real world. In fact, inside GHC at least, ST is used, and the environment is a type called `RealWorld`.

To get out of the State monad, you can use `runState`. The analogous function for ST is called `runST`, and it has a rather particular type:

Example: The `runST` function

```
runST :: forall a. (forall s. ST s a) -> a
```

This is actually an example of a more complicated language feature called rank-2 polymorphism, which we don't go into in detail here. It's important to notice that there is no parameter for the initial state. Indeed, ST uses a different notion of state to State; while State allows you to `get` and `put` the current state, ST provides an interface to *references*. You create references, which have type `STRef`, with `newSTRef :: a -> ST s (STRef s a)`, providing an initial value, then you can use `readSTRef :: STRef s a -> ST s a` and `writeSTRef :: STRef s a -> a -> ST s ()` to manipulate them. As such, the internal environment of a ST computation is not one specific value, but a mapping from references to values. Therefore, you don't need to provide an initial state to runST, as the initial state is just the empty mapping containing no references.

However, things aren't quite as simple as this. What stops you creating a reference in one ST computation, then using it in another? We don't want to allow this because (for reasons of thread-safety) no ST computation should be allowed to assume that the initial internal environment contains any specific references. More concretely, we want the following code to be invalid:

Example: Bad ST code

```
 let v = runST (newSTRef True)
 in runST (readSTRef v)
```

What would prevent this? The effect of the rank-2 polymorphism in `runST`'s type is to *constrain the scope of the type variable `s`* to be within the first parameter. In other words, if the type variable `s` appears in the first parameter it cannot also appear in the second. Let's take a look at how exactly this is done. Say we have some code like the following:

Example: Briefer bad ST code

```
... runST (newSTRef True) ...
```

The compiler tries to fit the types together:

Example: The compiler's typechecking stage

```
newSTRef True :: forall s. ST s (STRef s Bool)
runST :: forall a. (forall s. ST s a) -> a
together, forall a. (forall s. ST s (STRef s Bool)) -> STRef s Bool
```

The importance of the `forall` in the first bracket is that we can change the name of the `s`. That is, we could write:

Example: A type mismatch!

```
together, forall a. (forall s'. ST s' (STRef s' Bool)) -> STRef s Bool
```

This makes sense: in mathematics, saying $\forall x.x > 5$ is precisely the same as saying $\forall y.y > 5$; you're just giving the variable a different label. However, we have a problem with our above code. Notice that as the `forall` does *not* scope over the return type of `runST`, we don't rename the `s` there as well. But suddenly, we've got a type mismatch! The result type of the ST computation in the first parameter must match the result type of `runST`, but now it doesn't!

The key feature of the existential is that it allows the compiler to generalise the type of the state in the first parameter, and so the result type cannot depend on it. This neatly sidesteps our dependence problems, and 'compartmentalises' each call to `runST` into its own little heap, with references not being able to be shared between different calls.

Quantification as a primitive

Universal quantification is useful for defining data types that aren't already defined. Suppose there was no such thing as pairs built into haskell. Quantification could be used to define them.

```
{-# LANGUAGE ExistentialQuantification, RankNTypes #-}

newtype Pair a b = Pair (forall c. (a -> b -> c) -> c)

makePair :: a -> b -> Pair a b
makePair a b = Pair $ \f -> f a b
```

In GHCi:

```
λ> :bro
newtype Pair a b = Pair {runPair :: forall c. (a -> b -> c) -> c}
makePair :: a -> b -> Pair a b

λ> let pair = makePair "a" 'b'

λ> :t pair
pair :: Pair [Char] Char

λ> runPair pair (\x y -> x)
"a"

λ> runPair pair (\x y -> y)
'b'
```

Notes

1. Actually, we can apply them to functions whose type is `forall a. a -> R`, for some arbitrary `R`, as these accept values of any type as a parameter. Examples of such functions: `id`, `const k` for any `k`, `seq`. So technically, we can't do anything *useful* with its elements, except reduce them to WHNF.

Further reading

- GHC's user guide contains useful information (http://haskell.org/ghc/docs/latest/html/users_guide/data-type-extensions.html#existential-quantification) on existentials, including the various limitations placed on them (which you should know about).
- *Lazy Functional State Threads (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.3299), by Simon Peyton-Jones and John Launchbury, is a paper which explains more fully the ideas behind ST.*