Squashed c x let a library writer provide *x* in "*c*-irrelevant" way to a library user.

```
newtype Squashed c x = Squash
  {getSquashed :: ∀r.c r ⇒ (x → r) → r}
```

Squashed is almost like *Cont*[1] or *Codensity* [2], so *Squashed* is a *Monad*:

```
instance Monad (Squashed c) where
  return x = Squash ($x)
  m >>= k = Squash $ λbx →
    getSquashed m $ λa →
    getSquashed (k a) bx
instance Applicative (Squashed c) where
  pure  = return
  liftA2 = liftM2
instance Functor (Squashed c) where
  fmap  = liftM
```

Monad-instance allows to work on the wrapped value, for example

```
squashedTree' :: Squashed Monoid (Tree String)
squashedTree' = pure $ Node "x" [pure "yz", pure "foo"]
squashedTree :: Squashed Monoid (Tree Int)
squashedTree = do
  x ← squashedTree'
  return (fmap length x)
```

However, we cannot *extract* the original value, only as much as the constraint let us:

```
  -- 6
example_1 :: Int
example_1 = getSum (getSquashed squashedTree (foldMap Sum))
  -- [1,2,3]
example_2 :: [Int]
example_2 = getSquashed squashedTree (foldMap pure)
```

This restriction maybe be useful to enforce correctness, without relying on the module system!

Squash c x is a generalised notion of "free *c* over *x*", e.g. *Monoid* as described in Free Monoids in Haskell[3]. It should be possible to write *c* (*Squashed c x*) instances for all (reasonable) *c*. Or actually (∀*x*.*c*′ *x* ⇒ *c x*) ⇒ *c* (*Squashed c*′ *a*) after Quantified Constraints -proposal[4] is implemented. (TODO: amend when we have the extension in released GHC).

```
instance Semigroup (Squashed Semigroup x) where
  a ◇ b = Squash $ λk → getSquashed a k ◇ getSquashed b k
instance Monoid (Squashed Monoid x) where
  mempty      = Squash $ λ_ → mempty
  mappend a b = Squash $ λk → getSquashed a k `mappend` getSquashed b k
```

As with Singleton containers[5], tell me if you have seen this construction in the wild!

[1] https://hackage.haskell.org/package/transformers-0.5.5.0/docs/Control-Monad-Trans-Cont.html#t:ContT

[2] http://hackage.haskell.org/package/kan-extensions-5.1/docs/Control-Monad-Codensity.html#t:Codensity

[3] http://comonad.com/reader/2015/free-monoids-in-haskell/

[4] https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0018-quantified-constraints.rst

[5] http://oleg.fi/gists/posts/2018-05-12-singleton-container.html

Addendum: As `Iceland_jack` pointed on Twitter[6][7]there is a `free-functors`[8] package on Hackage, and more is written about *Squash*:

- `http://comonad.com/reader/2015/domains-sets-traversals-and-applicatives/`
- `https://www.cs.ox.ac.uk/ralf.hinze/Kan.pdf`

Note, that *Squash* doesn't let us turn a thing into something it isn't...

```
newtype Squashed1 c f x = Squash1
  {getSquashed1 :: ∀g.c g ⇒ (∀y.f y → g y) → g x}
squash1 :: f x → Squashed1 c f x
squash1 fx = Squash1 ($fx)
instance Monad (Squashed1 Monad f) where
  return x = Squash1 $ λ_ → return x
  m >>= k = Squash1 $ λf →
    getSquashed1 m f >>= λy →
    getSquashed1 (k y) f
instance Applicative (Squashed1 Monad f) where
  pure  = return
  liftA2 = liftM2
instance Functor (Squashed1 Monad f) where
  fmap  = liftM
```

... though we can foolishly think so:

```
intSet' :: Squashed1 Monad Set Int
intSet' = squash1 $ Set.fromList [1,2,3]
intSet :: Squashed1 Monad Set Int
intSet = intSet' >>= λ_ → return 5
  -- [5,5,5]
intList :: [Int]
intList = getSquashed1 intSet Set.toList
```

So *Squash* let's only forget, not to "remember" anything new.

By the way, this post is genuine Literate Haskell file, using LaTeX, not Markdown. If interested on how, check the gists repository[9]. I'm weird, as after some point of markup complexity, I actually prefer LaTeX.

[6] `https://twitter.com/Iceland_jack/status/1001081879045525504`
[7] `https://twitter.com/Iceland_jack/status/1001083326965407745`
[8] `https://hackage.haskell.org/package/free-functors`
[9] `https://github.com/phadej/gists`