Existentials:

Playing Hide and Seek With Your Types

Nicholas Cowle (@nickcowle)

G-Research

Example: Writing a parser

```
parseList
    "[ 1 ; 2 ; 3 ]"
        = [ 1 ; 2 ; 3 ]

parseList
    "[ true ; false ]"
        = [ true ; false ]
```

Example: Writing a parser

```
parseList "[ 1 ; 2 ; 3 ]" = [ 1 ; 2 ; 3 ]
parseList "[ true ; false ]" = [ true ; false ]

val parseList : string -> ?
val parseList : string -> obj 😭
val parseList : string -> obj list 😠
val parseList : string -> ∃ 'a . ('a list) 🤪
```

Universal Quantification

```
∀ 'a . ('a list -> int)
```

```
module List =

    val length : 'a list -> int
```

Universal Quantification vs. Generics

```
let sumLengths
    (xs : int list)
    (ys : string list)
    (getLength : ???)
    : int =

    getLength xs + getLength ys
```

Polymorphism in F#

Generics ✓
First-class universals ✗

Emulating Universal Quantification

```
type IGetListLength = abstract member Invoke<'a> : 'a list -> int
```

Emulating Universal Quantification

```
type IGetListLength = abstract member Invoke<'a> : 'a list -> int

let sumLengths
    (xs : int list)
    (ys : string list)
    (getLength : IGetListLength)
    : int =

    getLength.Invoke xs + getLength.Invoke ys
```

$$\exists \rightleftarrows \forall$$

We want to emulate:

```
∃ 'a . ('a list)
```

Another trick - Continuation Passing Style

```
 'a ≅ ∀ 'ret . (('a -> 'ret) -> 'ret)

int ≅ ∀ 'ret . ((int -> 'ret) -> 'ret)
```

```
type CPSInt = abstract member Eval<'ret> : (int -> 'ret) -> 'ret
```

Implementing our existential

```
           'a ≅ ∀ 'ret . (('a -> 'ret) -> 'ret)

∃ 'a . 'a list ≅ ∀ 'ret . ((∃ 'a . ('a list) -> 'ret) -> 'ret)
               ≅ ∀ 'ret . ( ∀ 'a . ('a list  -> 'ret) -> 'ret)
```

```
type ListCrate =
    abstract member Apply<'ret> : ListCrateEvaluator<'ret> -> 'ret

and ListCrateEvaluator<'ret> =
    abstract member Eval<'a> : 'a list -> 'ret
```

Implementing our existential

```
           'a ≅ ∀ 'ret . (('a -> 'ret) -> 'ret)

∃ 'a . 'a list ≅ ∀ 'ret . ((∃ 'a . ('a list) -> 'ret) -> 'ret)
               ≅ ∀ 'ret . ( ∀ 'a . ('a list  -> 'ret) -> 'ret)
```

```
type ListCrate =
    abstract member Apply<'ret> : ListCrateEvaluator<'ret> -> 'ret

and ListCrateEvaluator<'ret> =
    abstract member Eval<'a> : 'a list -> 'ret
```

Using Crates

```
let makeListCrate (list : 'a list) : ListCrate =
    { new ListCrate with
        member __.Apply e = e.Eval list
    }
```

```
let getLength (list : ListCrate) : int =
    list.Apply
        { new ListCrateEvaluator<int> with
            member __.Eval (list : 'a list) = List.length list
        }
```

https://www.gresearch.co.uk/2018/04/05/introducing-crates