Type-Safe Datatype-Generic Programming in F#

Nicholas Cowle

 nickcowle

 @nickcowle

September 2019

What is Datatype-Generic programming?

"We use it to mean polytypism, that is, parametrization by the shape of data structures rather than their contents."

Jeremy Gibbons

Datatype-Generic Programming

What do I mean by Type-Safe?

- No values of type obj
- No unsafe casts

Tonight's example: Writing a CSV parser

```
module CSVParser =

    val tryParse<'record>
        : FileInfo
        -> 'record seq option
```

```
type MyRecord =
    {
        Id          : int
        Name        : string
        DateOfBirth : DateTime
        NewUser     : bool
        Balance     : float
    }
```

```
type MyRecord =
    {
        Id          : int
        Name        : string
        DateOfBirth : DateTime
        NewUser     : bool
        Balance     : float
    }
```

TestData.csv:

```
001,Derry Williamson,1974-03-12,true ,12.34
002,Madelyn Milne   ,1988-11-23,false,56.78
```

```
type MyRecord =
    {
        Id           : int
        Name         : string
        DateOfBirth  : DateTime
        NewUser      : bool
        Balance      : float
    }
```

TestData.csv:

```
001,Derry Williamson,1974-03-12,true ,12.34
002,Madelyn Milne   ,1988-11-23,false,56.78
```

```
CSVParser.tryParse<MyRecord> "TestData.csv"
```

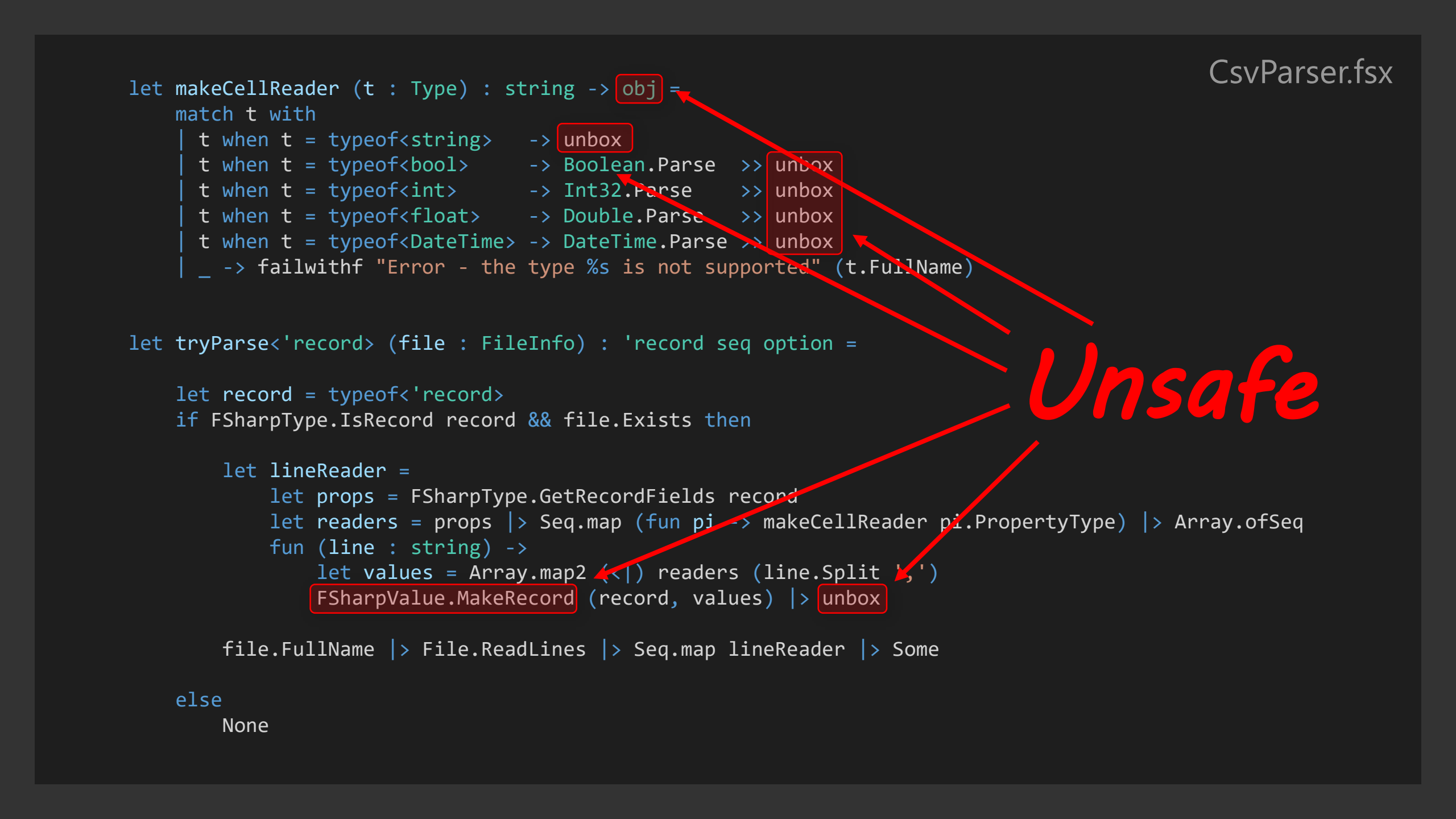
```
[
    {Id = 1; Name = "Derry Williamson"; DateOfBirth = 12/03/1974 00:00:00;
        NewUser = true; Balance = 12.34;};

    {Id = 2; Name = "Madelyn Milne"; DateOfBirth = 23/11/1988 00:00:00;
        NewUser = false; Balance = 56.78;}
]
```

CsvParser.fsx

```fsharp
let makeCellReader (t : Type) : string -> obj =
    match t with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (t.FullName)


let tryParse<'record> (file : FileInfo) : 'record seq option =

    let record = typeof<'record>
    if FSharpType.IsRecord record && file.Exists then

        let lineReader =
            let props = FSharpType.GetRecordFields record
            let readers = props |> Seq.map (fun pi -> makeCellReader pi.PropertyType) |> Array.ofSeq
            fun (line : string) ->
                let values = Array.map2 (<|) readers (line.Split ',')
                FSharpValue.MakeRecord (record, values) |> unbox

        file.FullName |> File.ReadLines |> Seq.map lineReader |> Some

    else
        None
```

CsvParser.fsx

```fsharp
let makeCellReader (t : Type) : string -> obj =
    match t with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (t.FullName)


let tryParse<'record> (file : FileInfo) : 'record seq option =

    let record = typeof<'record>
    if FSharpType.IsRecord record && file.Exists then

        let lineReader =
            let props = FSharpType.GetRecordFields record
            let readers = props |> Seq.map (fun pi -> makeCellReader pi.PropertyType) |> Array.ofSeq
            fun (line : string) ->
                let values = Array.map2 (<|) readers (line.Split ',')
                FSharpValue.MakeRecord (record, values) |> unbox

        file.FullName |> File.ReadLines |> Seq.map lineReader |> Some

    else
        None
```



CsvParser.fsx

```fsharp
let makeCellReader (t : Type) : string -> obj =
    match t with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (t.FullName)


let tryParse<'record> (file : FileInfo) : 'record seq option =

    let record = typeof<'record>
    if FSharpType.IsRecord record && file.Exists then

        let lineReader =
            let props = FSharpType.GetRecordFields record
            let readers = props |> Seq.map (fun pi -> makeCellReader pi.PropertyType) |> Array.ofSeq
            fun (line : string) ->
                let values = Array.map2 (<|) readers (line.Split ',')
                FSharpValue.MakeRecord (record, values) |> unbox

        file.FullName |> File.ReadLines |> Seq.map lineReader |> Some

    else
        None
```

CsvParser.fsx

```fsharp
let makeCellReader (t : Type) : string -> obj =
    match t with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  |> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (t.FullName)


let tryParse<'record> (file : FileInfo) : 'record seq option =

    let record = typeof<'record>
    if FSharpType.IsRecord record && file.Exists then

        let lineReader =
            let props = FSharpType.GetRecordFields record
            let readers = props |> Seq.map (fun pi -> makeCellReader pi.PropertyType) |> Array.ofSeq
            fun (line : string) ->
                let values = Array.map2 (<|) readers (line.Split ',')
                FSharpValue.MakeRecord (record, values) |> unbox

        file.FullName |> File.ReadLines |> Seq.map lineReader |> Some

    else
        None
```

```
System.InvalidCastException: Unable to cast object of type 'makeCellReader@19-
1' to type 'Microsoft.FSharp.Core.FSharpFunc`2[System.String,System.Object]'.
   at
Microsoft.FSharp.Core.LanguagePrimitives.IntrinsicFunctions.UnboxGeneric[T](Ob
ject source)
   at FSI_0002.CSVParser.makeCellReader(Type t) in
C:\Users\Nicholas\Dropbox\Datatype-Generic Talk\CsvParser.fsx:line 18
   at Microsoft.FSharp.Collections.Internal.IEnumerator.map@75.DoMoveNext(b&
curr)
   at
Microsoft.FSharp.Collections.Internal.IEnumerator.MapEnumerator`1.System-
Collections-IEnumerator-MoveNext()
   at System.Collections.Generic.List`1..ctor(IEnumerable`1 collection)
   at Microsoft.FSharp.Collections.SeqModule.ToArray[T](IEnumerable`1 source)
   at FSI_0002.CSVParser.tryParse[record](FileInfo file) in
C:\Users\Nicholas\Dropbox\Datatype-Generic Talk\CSVDirect.fsx:line 33
   at <StartupCode$FSI_0002>.$FSI_0002.main@()
Stopped due to error
```

```
System.InvalidCastException: Unable to cast object of type 'makeCellReader@19-1'
to type 'Microsoft.FSharp.Core.FSharpFunc`2[System.String,System.Object]'.
   at
Microsoft.FSharp.Core.LanguagePrimitives.IntrinsicFunctions.UnboxGeneric[T](Objec
t source)
   at FSI_0002.CSVParser.makeCellReader(Type t) in
C:\Users\Nicholas\Dropbox\Datatype-Generic Talk\CsvParser.fsx:line 18
   at Microsoft.FSharp.Collections.Internal.IEnumerator.map@75.DoMoveNext(b&
curr)
   at Microsoft.FSharp.Collections.Internal.IEnumerator.MapEnumerator`1.System-
Collections-IEnumerator-MoveNext()
   at System.Collections.Generic.List`1..ctor(IEnumerable`1 collection)
   at Microsoft.FSharp.Collections.SeqModule.ToArray[T](IEnumerable`1 source)
   at FSI_0002.CSVParser.tryParse[record](FileInfo file) in
C:\Users\Nicholas\Dropbox\Datatype-Generic Talk\CSVDirect.fsx:line 33
   at <StartupCode$FSI_0002>.$FSI_0002.main@()
Stopped due to error
```

CsvParser.fsx

```fsharp
let makeCellReader (t : Type) : string -> obj =
    match t with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (t.FullName)


let tryParse<'record> (file : FileInfo) : 'record seq option =

    let record = typeof<'record>
    if FSharpType.IsRecord record && file.Exists then

        let lineReader =
            let props = FSharpType.GetRecordFields record
            let readers = props |> Seq.map (fun pi -> makeCellReader pi.PropertyType) |> Array.ofSeq
            fun (line : string) ->
                let values = Array.map2 (<|) readers (line.Split ',')
                FSharpValue.MakeRecord (record, values) |> unbox

        file.FullName |> File.ReadLines |> Seq.map lineReader |> Some

    else
        None
```

CsvParser.fsx

```
let makeCellReader (t : Type) : string -> obj =
    match t with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (t.FullName)
```

CsvParser.fsx

```
let makeCellReader (t : Type) : string -> obj =
    match t with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (t.FullName)
```

```
let makeCellReader'<'a> : string -> 'a =
    match typeof<'a> with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (typeof<'a>.FullName)
```

CsvParser.fsx

```fsharp
let makeCellReader (t : Type) : string -> obj =
    match t with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (t.FullName)
```

```fsharp
let makeCellReader'<'a> : string -> 'a =
    match typeof<'a> with
    | t when t = typeof<string>   -> unbox
    | t when t = typeof<bool>     -> Boolean.Parse  >> unbox
    | t when t = typeof<int>      -> Int32.Parse    >> unbox
    | t when t = typeof<float>    -> Double.Parse   >> unbox
    | t when t = typeof<DateTime> -> DateTime.Parse >> unbox
    | _ -> failwithf "Error - the type %s is not supported" (typeof<'a>.FullName)
```

- Implementation is still unsafe
- Return type will be thrown away
- Harder to invoke

-> Even worse

To be continued...

Part I: TypeEquality

TypeEquality/Teq.fsi

```fsharp
namespace TypeEquality

/// A type for witnessing type equality between 'a and 'b
type Teq<'a, 'b>
```

TypeEquality/Teq.fsi

```fsharp
namespace TypeEquality

/// A type for witnessing type equality between 'a and 'b
type Teq<'a, 'b>

/// Module for creating and using type equalities, primarily useful for Generalised Algebraic Data Types (GADTs)
/// Invariant: If you use this module (without reflection shenanigans) and the
/// code builds, it will be correct.
[<RequireQualifiedAccess>]
module Teq =

    /// Converts an 'a to a 'b
    /// Alias for cast
    val castTo : Teq<'a, 'b> -> ('a -> 'b)

    /// Converts a 'b to an 'a
    /// Equivalent to symmetry >> cast, but more efficient
    val castFrom : Teq<'a, 'b> -> ('b -> 'a)
```

TypeEquality/Teq.fsi

```fsharp
namespace TypeEquality

/// A type for witnessing type equality between 'a and 'b
type Teq<'a, 'b>

/// Module for creating and using type equalities, primarily useful for Generalised Algebraic Data Types (GADTs)
/// Invariant: If you use this module (without reflection shenanigans) and the
/// code builds, it will be correct.
[<RequireQualifiedAccess>]
module Teq =

    /// Converts an 'a to a 'b
    /// Alias for cast
    val castTo : Teq<'a, 'b> -> ('a -> 'b)

    /// Converts a 'b to an 'a
    /// Equivalent to symmetry >> cast, but more efficient
    val castFrom : Teq<'a, 'b> -> ('b -> 'a)

    /// The single constructor for Teq - witnesses equality between 'a and 'a
    /// It would be nice to accept any isomorphism (i.e. 1-1 mapping between
    /// values, but Refl is the only provably correct constructor we can create
    /// in F#, so we choose soundness over completeness here).
    val refl<'a> : Teq<'a, 'a>
```

TypeEquality/Teq.fsi

```fsharp
namespace TypeEquality

/// A type for witnessing type equality between 'a and 'b
type Teq<'a, 'b>

/// Module for creating and using type equalities, primarily useful for Generalised Algebraic Data Types (GADTs)
/// Invariant: If you use this module (without reflection shenanigans) and the
/// code builds, it will be correct.
[<RequireQualifiedAccess>]
module Teq =

    /// Converts an 'a to a 'b
    /// Alias for cast
    val castTo : Teq<'a, 'b> -> ('a -> 'b)

    /// Converts a 'b to an 'a
    /// Equivalent to symmetry >> cast, but more efficient
    val castFrom : Teq<'a, 'b> -> ('b -> 'a)

    /// The single constructor for Teq - witnesses equality between 'a and 'a
    /// It would be nice to accept any isomorphism (i.e. 1-1 mapping between
    /// values, but Refl is the only provably correct constructor we can create
    /// in F#, so we choose soundness over completeness here).
    val refl<'a> : Teq<'a, 'a>
```

```fsharp
Teq of ('a -> 'b) * ('b -> 'a)

let castTo (Teq (f, _)) a = f a

let castFrom (Teq (_, g)) b = g b

let refl = Teq (id, id)
```

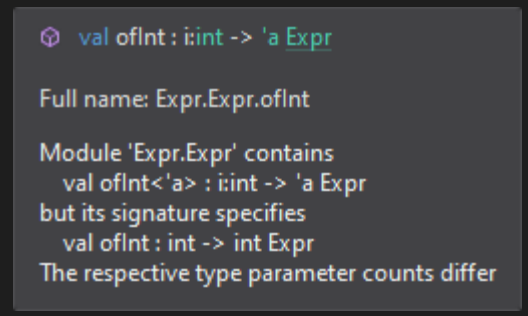
Expr.fsi

```
type 'a Expr

module Expr =

    val ofInt      : int       -> int  Expr
    val ofBool     : bool      -> bool Expr
    val not        : bool Expr -> bool Expr
    val add        : int  Expr -> int  Expr -> int Expr
    val ifThenElse : bool Expr -> 'a   Expr -> 'a  Expr -> 'a Expr

    val eval       : 'a Expr   -> 'a
```

Expr.fs

```
type 'a Expr =
| Int        of int
| Bool       of bool
| Not        of bool Expr
| Add        of int  Expr * int Expr
| IfThenElse of bool Expr * 'a  Expr * 'a Expr

module Expr =

    let ofInt i             = Expr.Int i
    let ofBool b            = Expr.Bool b
    let not e               = Expr.Not e
    let add e1 e2           = Expr.Add (e1, e2)
    let ifThenElse ec e1 e2 = Expr.IfThenElse (ec, e1, e2)
```

```fsharp
type 'a Expr =
| Int        of int
| Bool       of bool
| Not        of bool Expr
| Add        of int  Expr * int Expr
| IfThenElse of bool Expr * 'a  Expr * 'a Expr

module Expr =

    let ofInt i             = Expr.Int i
    let ofBool b            = Expr.Bool b
    let not e               = Expr.Not e
    let add e1 e2           = Expr.Add (e1, e2)
    let ifThenElse ec e1 e2 = Expr.IfThenElse (ec, e1, e2)
```



Expr.fs

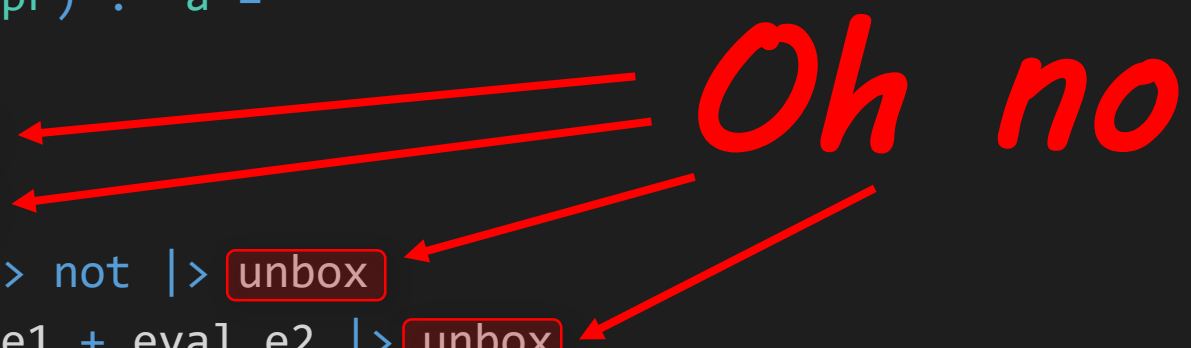
```fsharp
type 'a Expr =
| Int        of int
| Bool       of bool
| Not        of bool Expr
| Add        of int  Expr * int Expr
| IfThenElse of bool Expr * 'a  Expr * 'a Expr

module Expr =

    let ofInt i    : int  Expr = Expr.Int i
    let ofBool b   : bool Expr = Expr.Bool b
    let not e      : bool Expr = Expr.Not e
    let add e1 e2 : int  Expr = Expr.Add (e1, e2)
    let ifThenElse ec e1 e2    = Expr.IfThenElse (ec, e1, e2)
```

Expr.fs

```fsharp
module Expr =

    let rec eval<'a> (e : 'a Expr) : 'a =
        match e with
        | Int  i -> i |> unbox
        | Bool b -> b |> unbox
        | Not  e -> e |> eval |> not |> unbox
        | Add (e1, e2) -> eval e1 + eval e2 |> unbox
        | IfThenElse (ec, e1, e2) ->
            if eval ec then eval e1 else eval e2
```

Expr.fs

```fsharp
module Expr =

    let rec eval<'a> (e : 'a Expr) : 'a =
        match e with
        | Int  i -> i |> unbox
        | Bool b -> b |> unbox
        | Not  e -> e |> eval |> not |> unbox
        | Add (e1, e2) -> eval e1 + eval e2 |> unbox
        | IfThenElse (ec, e1, e2) ->
            if eval ec then eval e1 else eval e2
```



Expr2.fs

```
type 'a Expr =
| Int  of int                    * Teq<'a, int>
| Bool of bool                   * Teq<'a, bool>
| Not  of bool Expr              * Teq<'a, bool>
| Add  of int Expr * int Expr * Teq<'a, int>
| IfThenElse of bool Expr * 'a Expr * 'a Expr

module Expr =

    let ofInt i              = Expr.Int  (i,      Teq.refl)
    let ofBool b             = Expr.Bool (b,      Teq.refl)
    let not e                = Expr.Not  (e,      Teq.refl)
    let add e1 e2            = Expr.Add  (e1, e2, Teq.refl)
    let ifThenElse ec e1 e2 = Expr.IfThenElse (ec, e1, e2)
```

Expr2.fs

```fsharp
module Expr =

    let rec eval<'a> (e : 'a Expr) : 'a =
        match e with
        | Int  (i,       teq) -> i |> Teq.castFrom teq
        | Bool (b,       teq) -> b |> Teq.castFrom teq
        | Not  (e,       teq) -> e |> eval |> not |> Teq.castFrom teq
        | Add  (e1, e2, teq) -> eval e1 + eval e2 |> Teq.castFrom teq
        | IfThenElse (ec, e1, e2) ->
            if eval ec then eval e1 else eval e2
```

Expr.fs

```fsharp
let rec eval<'a> (e : 'a Expr) : 'a =
    match e with
    | Int  i -> i |> unbox
    | Bool b -> b |> unbox
    | Not  e -> e |> eval |> not |> unbox
    | Add (e1, e2) -> eval e1 + eval e2 |> unbox
    | IfThenElse (ec, e1, e2) ->
        if eval ec then eval e1 else eval e2
```

Expr2.fs

```fsharp
let rec eval<'a> (e : 'a Expr) : 'a =
    match e with
    | Int  (i,      teq) -> i |> Teq.castFrom teq
    | Bool (b,      teq) -> b |> Teq.castFrom teq
    | Not  (e,      teq) -> e |> eval |> not |> Teq.castFrom teq
    | Add  (e1, e2, teq) -> eval e1 + eval e2 |> Teq.castFrom teq
    | IfThenElse (ec, e1, e2) ->
        if eval ec then eval e1 else eval e2
```

Part II:
HCollections

a.k.a. “Heterogenous Collections”
a.k.a. Collections that can hold values with different types

Collections in F#

`'a List`

- List can be any length you like
- All elements must be of the same type

`'a * 'b * 'c`

- Each element can have a different type
- The “length” of the tuple is fixed

`???`

- Can be any length you like
- Each element can have a different type

3 Types To Look At:

TypeList

A variadic
type-level list
of types

HList

A variadic
heterogeneous list
of elements

HUnion

A variadic
heterogeneous
union type

HCollections/TypeList.fsi

TypeList

```
type 'ts TypeList

module TypeList =

    /// The unique empty TypeList
    val empty : unit TypeList

    /// Given an TypeList, prepends a new type
    /// to the list of types being represented.
    val cons<'t, 'ts> : 'ts TypeList -> ('t -> 'ts) TypeList
```

TypeList Examples

```
// (string -> bool -> int -> unit) TypeList
let example1 =
    TypeList.empty
    |> TypeList.cons<int, _>
    |> TypeList.cons<bool, _>
    |> TypeList.cons<string, _>
```

≈ [string ; bool ; int]

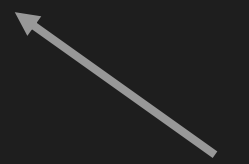
```
let example2 : (float -> bool -> float -> unit) TypeList =
    TypeList.empty |> TypeList.cons |> TypeList.cons |> TypeList.cons
```

≈ [float ; bool ; float]

HCollections/TypeList.fsi

Using TypeLists

```
module TypeList =

    /// Given a non-empty TypeList, returns a new TypeList containing all of the elements
    /// except the head.
    val tail<'t, 'ts> : ('t -> 'ts) TypeList -> 'ts TypeList

    /// Given a TypeList, returns either a proof that the list is empty, or a crate
    /// containing the tail of the TypeList.
    val split : 'ts TypeList -> Choice<Teq<'ts, unit>, 'ts TypeListConsCrate>
```



≈ Teq<'ts, 'u -> 'us>

HCollections/HList.fsi

HList

```
type 'ts HList

module HList =

    /// The unique empty HList
    val empty : unit HList

    /// Given an element and an HList, returns a new HList with the element prepended to it.
    val cons<'t, 'ts> : 't -> 'ts HList -> ('t -> 'ts) HList
```

HCollections/HList.fsi

HList

```
type 'ts HList

module HList =

    /// The unique empty HList
    val empty : unit HList

    /// Given an element and an HList, returns a new HList with the element prepended to it.
    val cons<'t, 'ts> : 't -> 'ts HList -> ('t -> 'ts) HList
```

HList Examples

```
// (string -> bool -> int -> unit) HList
let example1 =
    HList.empty
    |> HList.cons 1234
    |> HList.cons true
    |> HList.cons "hello"
```

≈ ("hello", true, 1234)

```
let example2 : (float -> bool -> float -> unit) HList =
    HList.empty |> HList.cons 5.5 |> HList.cons false |> HList.cons 12.5
```

≈ (12.5, false, 5.5)

HCollections/HList.fsi

Using HLists

```
module HList =

    /// Returns the length of the given HList
    val length<'ts> : 'ts HList -> int

    /// Given a non-empty HList, returns the first element.
    val head<'t, 'ts> : ('t -> 'ts) HList -> 't

    /// Given a non-empty HList, returns a new HList containing all of the elements
    /// except the head.
    val tail<'t, 'ts> : ('t -> 'ts) HList -> 'ts HList

    /// Given an HList, returns a TypeList whose types correspond to the values
    /// of the elements of the HList.
    val toTypeList<'ts> : 'ts HList -> 'ts TypeList
```

HCollections/HUnion.fsi

HUnion

```
type 'ts HUnion

module HUnion =

    /// Given a TypeList and a value, creates an HUnion whose cases are exactly the
    /// cases in the TypeList, plus one case for the value supplied.
    /// Notice that when an HUnion is created using make, the value that it holds is always
    /// the first of the choices. Use HUnion.Extend to prepend further choices.
    val make<'t, 'ts> : 'ts TypeList -> 't -> ('t -> 'ts) HUnion

    /// Given an HUnion, extends the choices by prepending a single additional choice to
    /// the front. Note that we do not have to supply a value as the HUnion must, by
    /// definition, already be holding a single value.
    val extend<'t, 'ts> : 'ts HUnion -> ('t -> 'ts) HUnion
```

HUnion Examples

```
// (string -> unit) HUnion
let example1 = HUnion.make TypeList.empty "hello"
```

```
≈ Choice1Of1 "hello"
  : Choice<string>
```

```
// (int -> bool -> string -> unit) HUnion
let example2 =
    let ts = TypeList.empty |> TypeList.cons<string, _>
    HUnion.make ts false
    |> HUnion.extend<int, _>
```

```
≈ Choice2Of3 false
  : Choice<int, bool, string>
```

HCollections/HUnion.fsi

Using HUnions

```
module HUnion =

    /// Given a ('t -> 'ts) HUnion, returns a choice of either a 't (in the case where
    /// the value of the union corresponded to the first case of the choice) or a
    /// 'ts HUnion in the case where the value of the union corresponds to one of the
    /// choices denoted by 'ts.
    val split<'t, 'ts> : ('t -> 'ts) HUnion -> Choice<'t, 'ts HUnion>

    /// Given an HUnion that contains only a single case, returns the value of that case.
    val getSingleton : ('t -> unit) HUnion -> 't

    /// Given an HUnion, returns a TypeList whose types correspond to the
    /// cases of the HUnion.
    val toTypeList : 'ts HUnion -> 'ts TypeList
```

Part III: TeqCrate

a.k.a. Teqs in Crates

TeqCrate/Patterns.fsi

Typed Type

```fsharp
/// Contains a set of active patterns to analyse typed runtime Types.
module Patterns =

    /// TType (short for 'Typed Type') is a value of a runtime type that is also generic on its value.
    /// We pattern match on typed types (rather than just runtime types) when trying to match against
    /// our TeqCrates so that the type that we're matching against corresponds to the type in the Teq.
    type 'a TType = TType of unit

    /// Single constructor for TType - creates a TType value of 'a when invoked with any generic
    /// type parameter 'a
    val tType<'a> : 'a TType
```

TeqCrate/Patterns.fsi

Simple Example

```
/// Contains a set of active patterns to analyse typed runtime Types.
module Patterns =

    /// Recognises tTypes that represent the string type.
    val (|String|_|) : 'a TType -> Teq<'a, string> option



let tryString (a : 'a) : string option =
    match tType<'a> with
    | String (teq : Teq<'a, string>) -> Teq.castTo teq a |> Some
    | _ -> None
```

TeqCrate/Patterns.fsi

List Example

```fsharp
/// Contains a set of active patterns to analyse typed runtime Types.
module Patterns =

    /// Recognises tTypes that represent a list type.
    val (|List|_|) : 'a TType -> 'a ListTeqCrate option
```

≈ Teq<'a, 'b list>

```fsharp
let tryListLength (a : 'a) : int option =
    match tType<'a> with
    | List crate ->
        crate.Apply
            { new ListTeqEvaluator<_,_> with
                member __.Eval (teq : Teq<'a, 'b list>) =
                    a |> Teq.castTo teq |> List.length |> Some
            }
    | _ -> None
```

TeqCrate/Patterns.fsi

List Example 2

```fsharp
let tryListSomeCount (a : 'a) : int option =
    match tType<'a> with
    | List crate ->
        crate.Apply
            { new ListTeqEvaluator<_,_> with
                member __.Eval (teq1 : Teq<'a, 'b list>) =
                    match tType<'b> with
                    | Option crate ->
                        crate.Apply
                            { new OptionTeqEvaluator<_,_> with
                                member __.Eval (teq2 : Teq<'b, 'c option>) =
                                    let teq : Teq<'a, 'c option list> =
                                        Teq.transitivity teq1 (Teq.Cong.list teq2)
                                    let xs : 'c option list = Teq.castTo teq a
                                    xs |> List.filter Option.isSome |> List.length |> Some
                            }
                    | _ -> None
            }
    | _ -> None
```

TeqCrate/Patterns.fsi

Tuple Example

```fsharp
/// Contains a set of active patterns to analyse typed runtime Types.
module Patterns =

    /// Recognises tTypes that represent a tuple type.
    val (|Tuple|_|) : 'a TType -> 'a TupleConvCrate option


let tryTupleLength (a : 'a) : int option =
    match tType<'a> with
    | Tuple crate ->
        crate.Apply
            { new TupleConvEvaluator<_,_> with
                member __.Eval (ts : 'ts TypeList) (conv : Conv<'a, 'ts HList>) =
                    a |> conv.To |> HList.length |> Some
            }
    | _ -> None
```

≈ Conv<'a, 'ts HList>

TeqCrate/Patterns.fsi

Tuple Example 2

```
let trySumTupleInts (a : 'a) : int option =
    match tType<'a> with
    | Tuple crate ->
        crate.Apply
            { new TupleConvEvaluator<_,_> with
                member __.Eval _ (conv : Conv<'a, 'ts HList>) =
                    let xs : 'ts HList = a |> conv.To
                    let folder =
                        { new HListFolder<int> with
                            member __.Folder sum (x : 'b) =
                                match tType<'b> with
                                | Int teq -> sum + (x |> Teq.castTo teq)
                                | _ -> sum
                        }
                    HList.fold folder 0 xs |> Some
            }
    | _ -> None
```

TeqCrate/Patterns.fsi

Record Example

```
let rec shoutify<'ts> (xs : 'ts HList) : 'ts HList =
    match xs |> HList.toTypeList |> TypeList.split with
    | Choice1Of2 _ -> xs
    | Choice2Of2 crate ->
        crate.Apply
            { new TypeListConsEvaluator<_,_> with
                member __.Eval _ (teq : Teq<'ts, 'u -> 'us>) =
                    let xs : ('u -> 'us) HList = xs |> Teq.castTo (HList.cong teq)
                    let head =
                        match tType<'u> with
                        | String teq -> (xs |> HList.head |> Teq.castTo teq).ToUpper () |> Teq.castFrom teq
                        | _ -> xs |> HList.head
                    let tail = xs |> HList.tail |> shoutify
                    HList.cons head tail |> Teq.castFrom (HList.cong teq)
            }

let tryShoutifyRecord (a : 'a) : 'a option =
    match tType<'a> with
    | Record crate ->
        crate.Apply
            { new RecordConvEvaluator<_,_> with
                member __.Eval _ _ (conv : Conv<'a, 'ts HList>) =
                    let xs : 'ts HList = a |> conv.To
                    shoutify xs |> conv.From |> Some
            }
    | _ -> None
```

Bringing it all together

```
module CSVParser =

    val tryParse<'record>
        : FileInfo
        -> 'record seq option
```

CsvParser2.fsx

```fsharp
let parseCell<'a> : string -> 'a =
    match tType<'a> with
    | String   (teq : Teq<'a, string  >) -> Teq.castFrom teq
    | Bool     (teq : Teq<'a, bool    >) -> Boolean .Parse >> Teq.castFrom teq
    | Int      (teq : Teq<'a, int     >) -> Int32   .Parse >> Teq.castFrom teq
    | Float    (teq : Teq<'a, float   >) -> Double  .Parse >> Teq.castFrom teq
    | DateTime (teq : Teq<'a, DateTime>) -> DateTime.Parse >> Teq.castFrom teq
    | _ -> failwithf "Error - the type %s is not supported" (typeof<'a>.FullName)

let rec parseRow<'ts> (ts : 'ts TypeList) (cells : string list) : 'ts HList =
    match TypeList.split ts with
    | Choice1Of2 (teq : Teq<'ts, unit>) -> HList.empty |> Teq.castFrom (HList.cong teq)
    | Choice2Of2 crate ->
        crate.Apply
            { new TypeListConsEvaluator<_,_> with
                member __.Eval (us : 'us TypeList) (teq : Teq<'ts, 'u -> 'us>) =
                    let head = cells |> List.head |> parseCell<'u>
                    let tail = cells |> List.tail |> parseRow us
                    HList.cons head tail |> Teq.castFrom (HList.cong teq)
            }

let tryParse<'record> (fileInfo : FileInfo) : 'record seq option =
    match tType<'record> with
    | Record crate ->
        crate.Apply
            { new RecordConvEvaluator<_,_> with
                member __.Eval _ (ts : 'ts TypeList) (conv : Conv<'record, 'ts HList>) =
                    File.ReadLines fileInfo.FullName
                    |> Seq.map (fun row -> row.Split ',' |> List.ofArray |> parseRow ts |> conv.From)
                    |> Some
            }
    | _ -> None
```

CsvParser2.fsx

```fsharp
let parseCell<'a> : string -> 'a =
    match tType<'a> with
    | String   (teq : Teq<'a, string  >) -> Teq.castFrom teq
    | Bool     (teq : Teq<'a, bool    >) -> Boolean .Parse >> Teq.castFrom teq
    | Int      (teq : Teq<'a, int     >) -> Int32   .Parse >> Teq.castFrom teq
    | Float    (teq : Teq<'a, float   >) -> Double  .Parse >> Teq.castFrom teq
    | DateTime (teq : Teq<'a, DateTime>) -> DateTime.Parse >> Teq.castFrom teq
    | _ -> failwithf "Error - the type %s is not supported" (typeof<'a>.FullName)

let rec parseRow<'ts> (ts : 'ts TypeList) (cells : string list) : 'ts HList =
    match TypeList.split ts with
    | Choice1Of2 (teq : Teq<'ts, unit>) -> HList.empty |> Teq.castFrom (HList.cong teq)
    | Choice2Of2 crate ->
        crate.Apply
            { new TypeListConsEvaluator<_,_> with
                member __.Eval (us : 'us TypeList) (teq : Teq<'ts, 'u -> 'us>) =
                    let head = cells |> List.head |> parseCell<'u>
                    let tail = cells |> List.tail |> parseRow us
                    HList.cons head tail |> Teq.castFrom (HList.cong teq)
            }

let tryParse<'record> (fileInfo : FileInfo) : 'record seq option =
    match tType<'record> with
    | Record crate ->
        crate.Apply
            { new RecordConvEvaluator<_,_> with
                member __.Eval _ (ts : 'ts TypeList) (conv : Conv<'record, 'ts HList>) =
                    File.ReadLines fileInfo.FullName
                    |> Seq.map (fun row -> row.Split ',' |> List.ofArray |> parseRow ts |> conv.From)
                    |> Some
            }
    | _ -> None
```

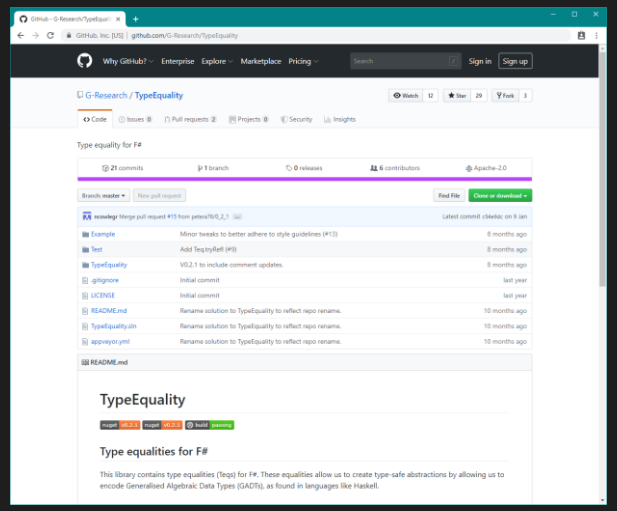
- No objects
- Sensible types for auxiliary methods
- No reflection code (that you can see!)

Success!

Try it out for yourself…

TypeEquality

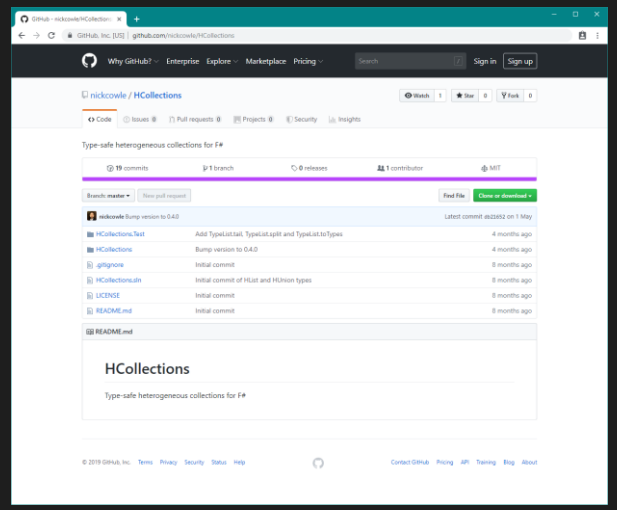
https://github.com/G-Research/TypeEquality



Teq

HCollections

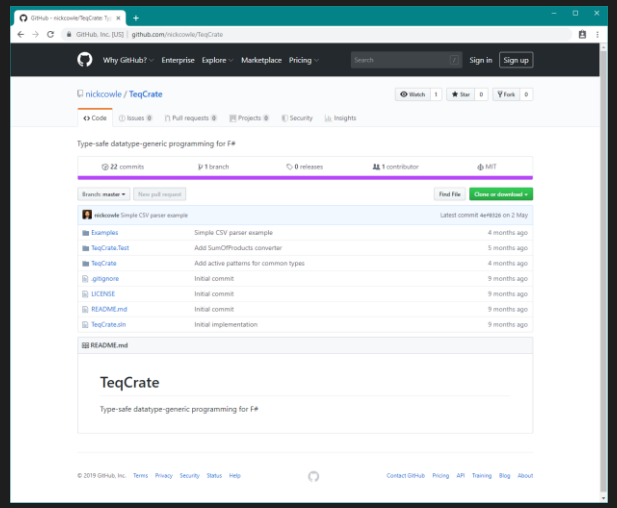
https://github.com/nickcowle/HCollections



TypeList, HList, HUnion

TeqCrate

https://github.com/nickcowle/TeqCrate



TType

Thanks for listening

Nicholas Cowle

 nickcowle

 @nickcowle

September 2019