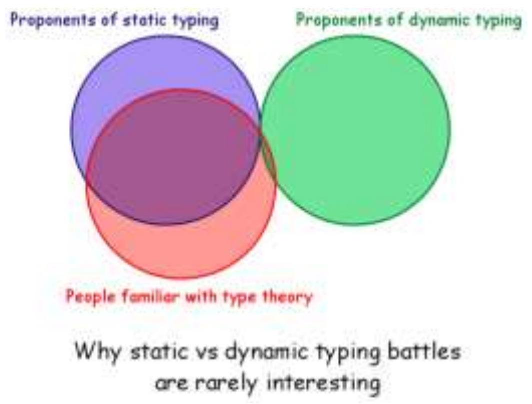
High Level Languages:

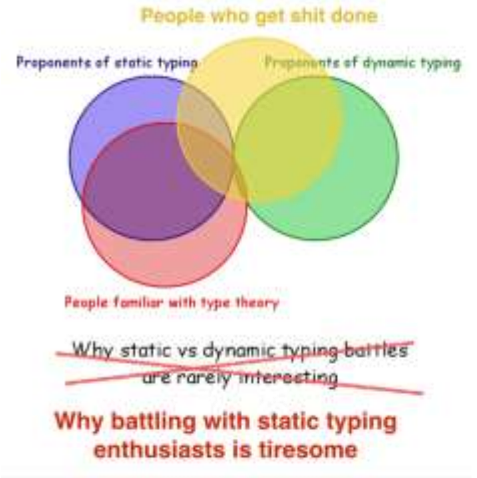
Type Systems

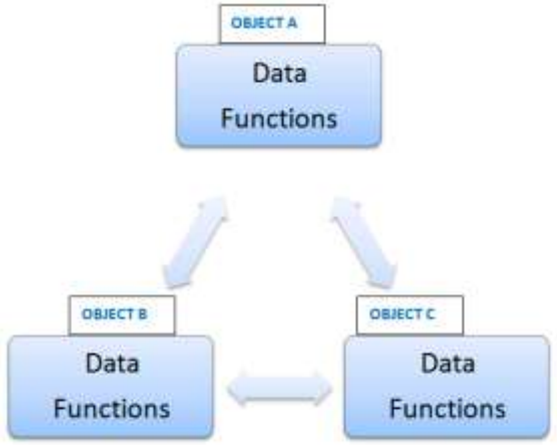
What makes a High Level Language?

1. Compiler does more work!
   - ❖ Static type systems
     - Use types as partial specifications
     - Use type inference to reduce work
     - Use type checking to prevent bugs
   - ❖ vs dynamic type systems
2. Objects and Object Hierarchies
3. Run-time System does more work!
   - ❖ Automatic memory allocation/deallocation
   - ❖ No more pointer bugs
   - ❖ Garbage collection

Two popular views…


Proponents of static typing
Proponents of dynamic typing
People familiar with type theory
Why static vs dynamic typing battles are rarely interesting


People who get shit done
Proponents of static typing
Proponents of dynamic typing
People familiar with type theory
~~Why static vs dynamic typing battles are rarely interesting~~
Why battling with static typing enthusiasts is tiresome



What makes a High level Language (2)?

- 4. Libraries!
  - e.g. Matlab toolboxes...
  - e.g. Python "Batteries included"
  - e.g. Javascipt and .node ecosystem
  - All JVM languages can access all Java libraries
    - Java, Scala, (not Microsoft)
  - All CLR languages can access all .NET libraries
    - C#, C++, Visual Basic, F#
    - Microsoft (used in enterprize software)
    - Hated (and loved)
- Languages need good interoperation (**interop**) with a large existing code base
- Good Interop => **compatible** type system
  - New languages must use old code bases
  - Interoperability makes compromise inevitable

What makes a High Level Language?

- Great Tooling
  - Integrated Development Environments
  - Intelligent Editors
  - Debuggers

- Static typing wins the tooling battle
  - Type-aware tooling
  - Type-based autocomplete
  - Hover type annotation
  - Refactoring support
    - Rename function throughout project

What is a good type system?

- We want types to catch detailed information about data and functions
  - Better error checking
  - Better runtime efficiency!
- We want types to be as expressive as possible – so programmers can do anything!
  - Reflection
  - Debugging frameworks
  - Code transformations
- Solution: more expressive (static) type systems?
  - Compilers don't understand them – no type inference
  - People don't understand them...
  - Hindley Milner is a "sweet spot" but still too complex for some people

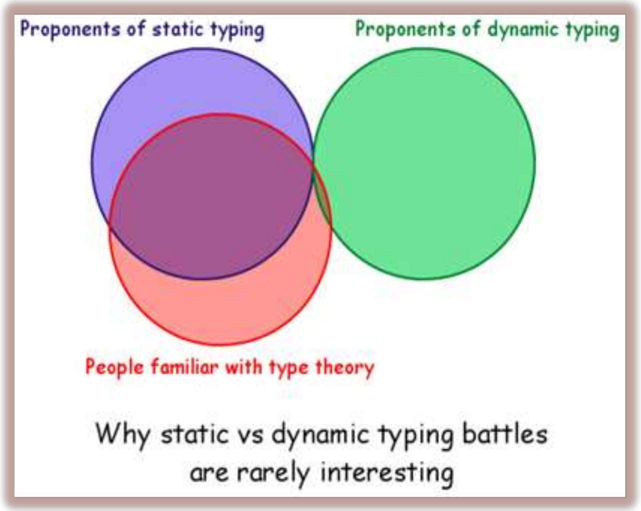
Static: every identifier has a fixed type label known by the compiler

Dynamic: types are owned by data: identifiers can change type arbitrarily

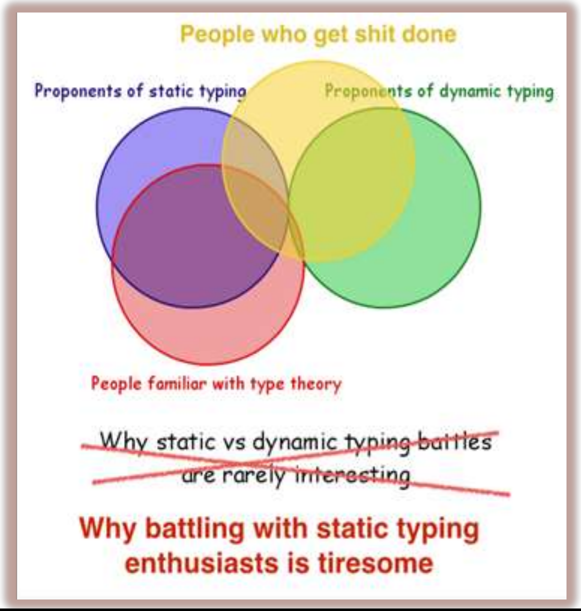
- No compiler cost for expressiveness
- Run-time cost working out type

ref: Dr. Dobbs journal

"Now! *That* should clear up a few things around here!"


Proponents of static typing
Proponents of dynamic typing
People familiar with type theory
Why static vs dynamic typing battles are rarely interesting

ref: Dr. Dobbs journal


People who get shit done
Proponents of static typing
Proponents of dynamic typing
People familiar with type theory
~~Why static vs dynamic typing battles are rarely interesting~~
Why battling with static typing enthusiasts is tiresome

Refs & Further Reading

good 2005 internet paper comment

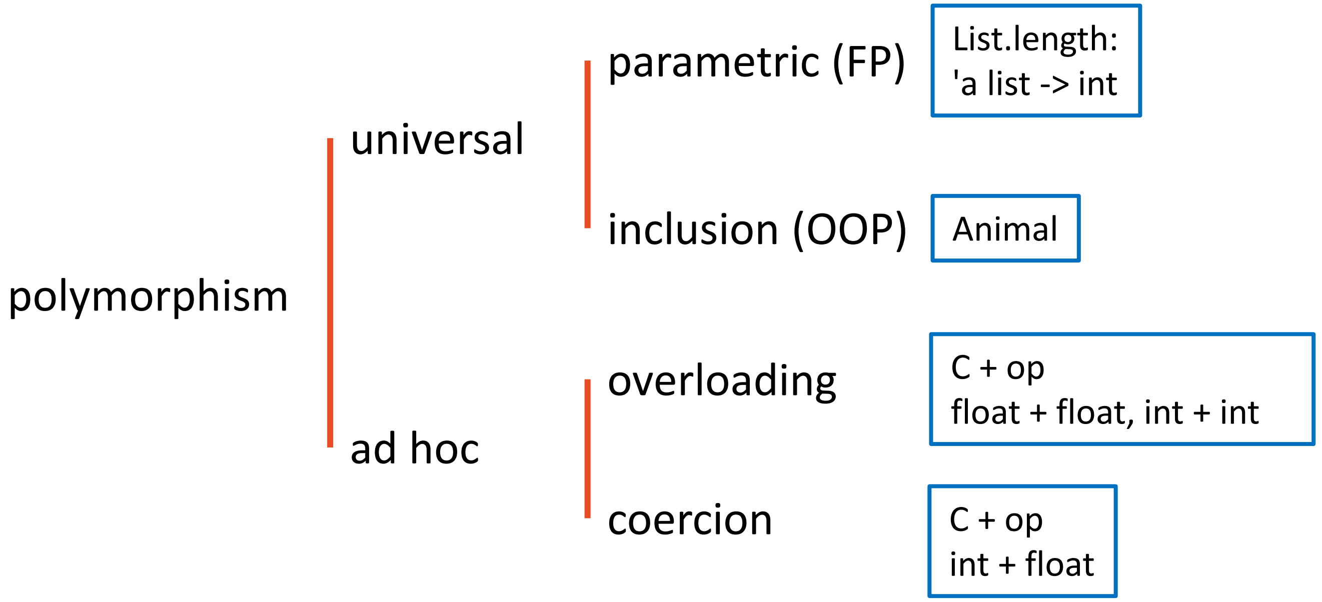
Gradual typing (Siek & Taha 2006)

Flow (Javascript partial type checking)

good Cornell U review of trade-offs slides 26-56

Type theory

- Classic (and good) reference: Cardelli 1985
- Think of type as a set of allowed values
- Type systems allow (a limited number) of useful types to be constructed.
- Polymorphism: values can have multiple types
- Polymorphic type: the set of all types a value satisfies



Examples of types:

int:	set of 32 bit signed integers
card:	set of non-negative signed 32 bit integers
0..7:	set of integers in range 0-7
int -> bool:	set of functions from int to bool

Polymorphic (generic) types in F#: parametric polymorphism

- Hindley-Milner Type system (ML, F#, OCAML, Haskell)
- **Parametric polymorphism** allows types to be built recursively from:
  - Fixed base types:
    - `int`, `float`,`string`
  - Wildcards (AKA polymorphic type variables or generics):
    - `'a`, `'b`
  - Data type constructors:
    - **tuple**, `list`, `map`, `option`, **any discriminated union**
  - Function type constructor `->`
- Is this same as C#/Java generics?
  - Yes – abstract concept is the same
  - No – details of how types can be refined are very different: FP parametric polymorphism versus OOP inclusion polymorphism

```
(+): int -> int -> int
(+): float -> float -> float

('a->'a->'a) -> 'a list ->'a
```

Specificity relation on polymorphic types (FP)

- ➢ Wildcards are universally quantified at top level
  - ❖ a given wildcard can represent any type, but must be the same type throughout the type expression
- ➢ We can order (partial order) polymorphic types according to specificity
- ➢ A type variable can stand for any type, it is less specific than anything else: base type, function type, data constructor
- ➢ If T_1 is more specific than T_2, T_1 values can always be used where T_2 values are expected
- ➢ Specificity corresponds to subset relationship on values of types:

$$T_1 \subseteq T_2$$

arrows point from a **more specific** subtype to a **more general** supertype

$$\begin{tikzcd}
 & & \texttt{'a} & \\
 & \texttt{'b -> 'c} \arrow[ur] \arrow[dr] & \texttt{int} \arrow[u] & \texttt{'a list} \arrow[ul] \\
\texttt{'b -> 'int} \arrow[ur] \arrow[dr] & & \texttt{int -> 'c} & \\
 & \texttt{int -> int} \arrow[ur] & &
\end{tikzcd}$$

Type constructors: Contravariance and Covariance

- How does specificity combine with → ?
- Note the surprising contravariance that applies to function type **input** parameters
  - data that is consumed
- **Output** parameters are covariant
  - data that is produced

$$\begin{tikzcd} 'b & & 'a \to 'b \\ & \text{$S \to T$ is \textbf{covariant} in T} & \\ \mathrm{int} \arrow[uu] & & 'a \to \mathrm{int} \arrow[uu] \end{tikzcd}$$

$$\begin{tikzcd} 'a & & 'a \to 'b \arrow[dd] \\ & \text{$S \to T$ is \textbf{contravariant} in S} & \\ \mathrm{int} \arrow[uu] & & \mathrm{int} \to 'b \end{tikzcd}$$

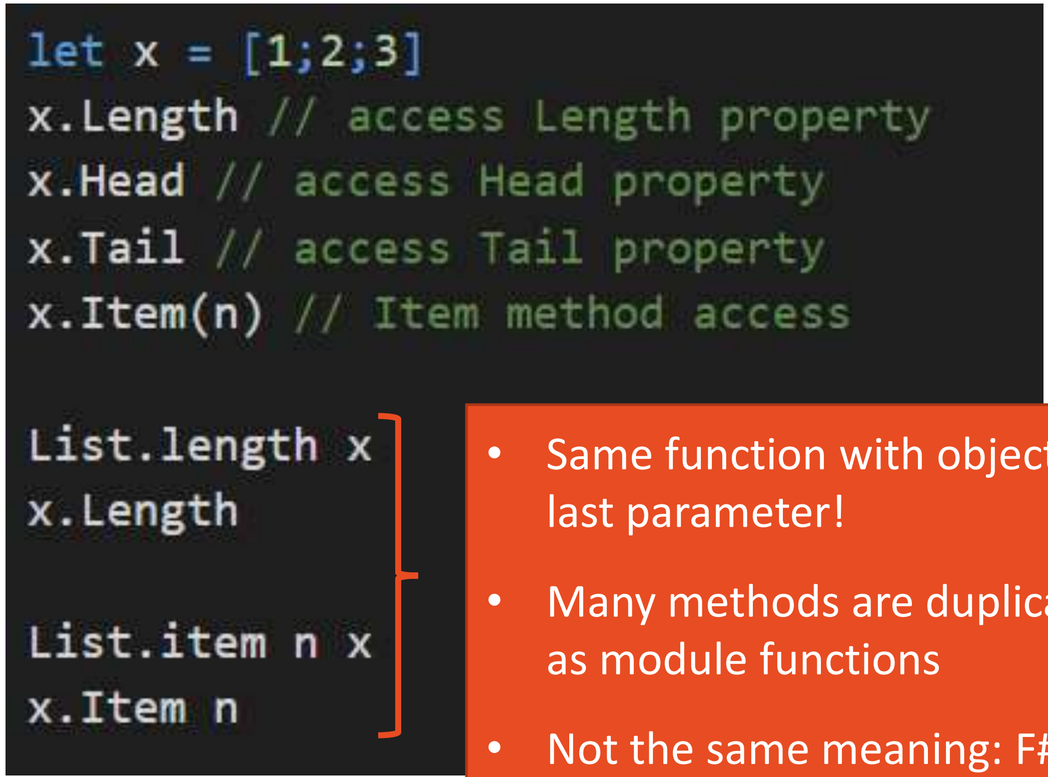
Further Reading

OOP Basics in F#

```
let x = [| 0..200 |]
let s = x.GetSlice( Some 10, Some 99)
// s = [|10..99|]
```

Method parameters are never curried

- OOP Basics
  - Objects are data with extra complexity:
    - Methods (functions operating on the data)
    - Properties (record fields defining the data)
  - Methods and properties accessed via **.** selector as for fields in a record
  - Every object is an **instance** of its **Class**
  - Internally:
    - Objects are records with fields for all the **properties**.
    - Classes are defined by the names and type signatures of property and method definitions: different objects share these but can have different data
- In F#
  - All types are also Classes.
  - Standard methods (ToString etc) apply to all types.
  - Use autocomplete on . to find methods + documentation



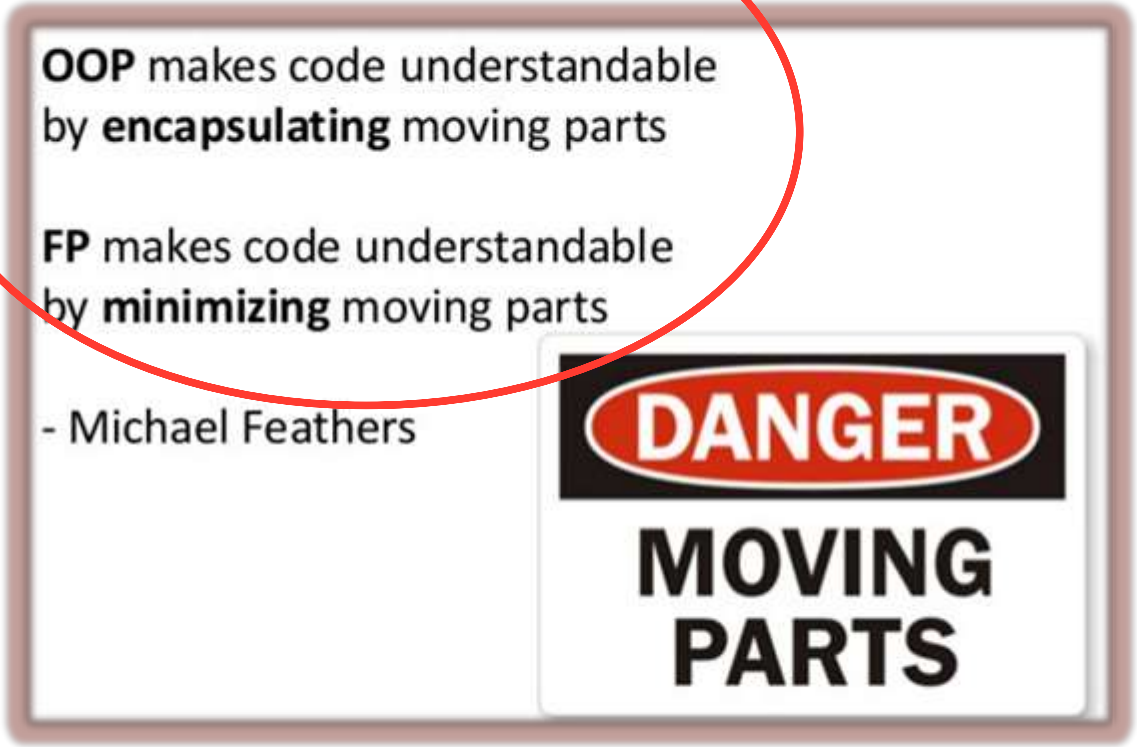
```
let x = [1;2;3]
x.Length // access Length property
x.Head // access Head property
x.Tail // access Tail property
x.Item(n) // Item method access

List.length x
x.Length

List.item n x
x.Item n
```

- Same function with object as last parameter!
- Many methods are duplicated as module functions
- Not the same meaning: F# type inference does not work so well on methods!

Are Objects Evil?

- The problem with OOP is that usually **objects** contain **mutable** state
  - NB some objects don't – e.g. F# lists
- In fact objects encapsulate state
  - A good thing?
  - Information hiding in objects is easy and works
  - cf abstract data types. GOOD not EVIL.
- Evil 1: Composing **mutable** objects to make a program is difficult to reason about
- Evil 2: Mutability + concurrency => non-determinism
- Further reading
  - Great slides from Mario Fusco

The expression problem
Neat problem showing how FP and OOP approaches have different strenths and weaknesses when incrementally adding code (Full references on next slide)



OOP makes code understandable by **encapsulating** moving parts

FP makes code understandable by **minimizing** moving parts

- Michael Feathers

DANGER
MOVING PARTS

Moore's Law means FP becomes more important!

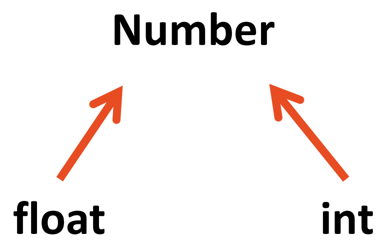
The Expression Problem: References for further reading

- Great problem in language theory illustrating the difference between FP and OOP approaches.
- Original source, ECOOP 1998: https://www.cs.rice.edu/~cork/teachjava/2003/readings/visitor1.pdf
- The "Expression problem" name came from Phil Wadler in email discussions after this, and has stuck.
- Some modern web pages describing the problem and/or solutions:
  - Eli Bendersky – detailed discussion using examples from compilers interpreters
  - Easy to read but insightful blog review from Joel Burget.
  - A practical solution using the concept of object algebras from ECOOP 2012
    - example code (including F#) http://i.cs.hku.hk/~bruno/oa/

Object Oriented Inheritance Hierarchy

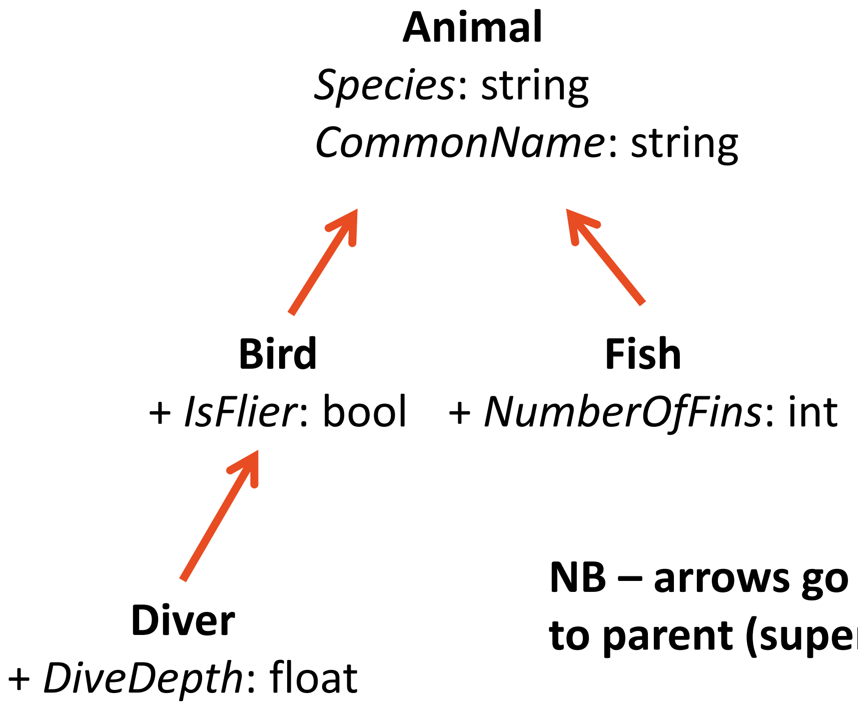
- Basic idea: types can be related via **inheritance** subtyping
  - Every value in the subtype is also a value of the type
  - e.g. built-in primitive types Number, float, int.
- Object oriented type hierarchy
  - Basic idea: refine an object type A adding additional properties/methods to make a subtype B
  - T1 value can be used whenever T2 value is required
    - T1 :> T2 (T1 is a subtype of T2)
  - Subtype relationship is subset relationship on values:

$$T_1 \subseteq T_2$$



Inheritance is clever but user-defined

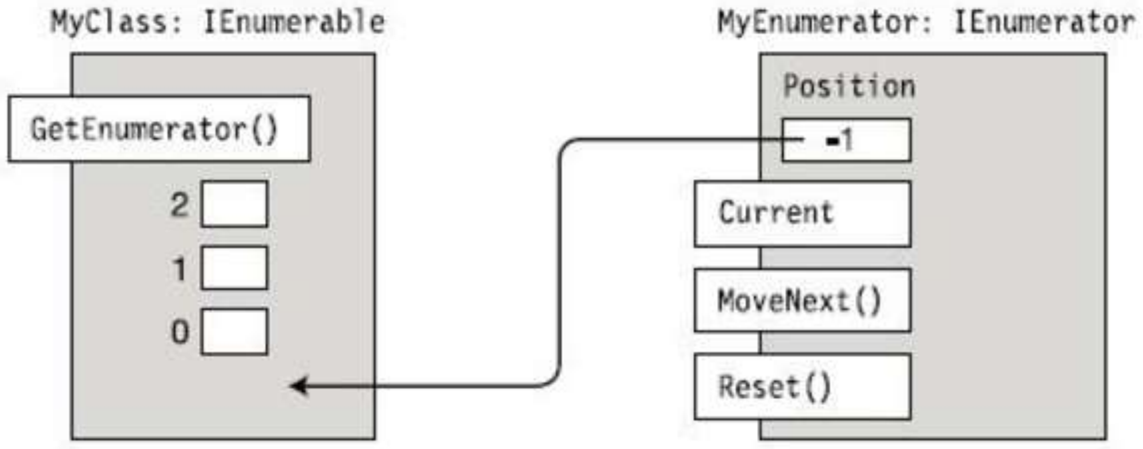
Inheritance is often a bad idea!
insightful further reading



NB – arrows go from child (subtype) to parent (supertype)

Interfaces in .NET (and other OO type systems)

- An interface specifies an abstract set of *members* that other OO classes can implement
- members are
  - properties: fields of object
  - methods: functions operating on an object
- If a class implements all the members of an interface we say it implements the interface.
- Interfaces are used to represent particular properties that many different classes can have
- Interfaces can have generic type parameters and so fit many different types
- generic = polymorphic



```
Interface:  IEnumerable<'T>

// Meaning:
// collections where you can step through 'T items
// one by one

satisfied by: 'T list, 'T array, 'T seq

// underlying non-functional .NET interface
type IEnumerator<'T> = interface
  abstract Current: 'T with get
  abstract MoveNext : unit -> bool
  abstract Reset: unit -> unit
  abstract Dispose : unit -> unit


type IEnumerable<'T> = interface
    abstract GetEnumerator : unit -> IEnumerator<'T>
```

$$\begin{tikzcd} & \text{IEnumerable<'T>} & \\ \text{'T list} \arrow[ur] & \text{'T array} \arrow[u] & \text{'T seq} \arrow[ul] \end{tikzcd}$$

IEnumerator interface and seq in F#

- .NET `IEnumerator<'T>` Interface defines F# **`'T seq`** data type
  - ❖ **`'T seq`** in F# packages the underlying IEnumerator properties and methods
  - ❖ **`'T seq`** allows F# programs to interface with the many .NET libraries using IEnumerator
  - ❖ Implementing `IEnumerator<'T>` in an F# class allows it to pretend to be a **`'T seq`** .
  - ❖ **`'T seq`** objects are often generated in F# from seq {} expressions.
- seq, array, list can all be converted easily e.g. **`List.ofSeq`**, **`List.toSeq`**

$$\begin{tikzcd}
& \texttt{IEnumerator<'T>} \\ & \texttt{'T seq} & \\
\texttt{List<'T>} \arrow[ur] \arrow[rr, phantom, "\text{implements}"] & & \texttt{Array<'T>} \arrow[ul]
\end{tikzcd}$$

- ***Implements*** **is like** ***inherits*** **in object hierarchy but with possibly multiple implemented interfaces**
- ***Interface*** **is like** ***class*** **but without implementation**
- **S implements T means S is a subtype of T**

Covariance and contravariance of interfaces?

- Obviously a sequence of Cats will match OK when a sequence of Animals is expected but not vice versa
  - So `IEnumerator<A>` will implement `IEnumerator<B>` if A is a subtype of B.
  - 'T is a **covariant parameter**
  - `IEnumerator` is defined by a function that **outputs** T.

$$\begin{tikzcd}\text{Animal} \arrow[d] & \text{IEnumerator<Animal>} \arrow[d] \\ \text{Cat} & \text{IEnumerator<Cat>}\end{tikzcd}$$

- What about a **contravariant parameter?** ?
  - `IComparable<'T>` interface (Int, float, string, etc)
  - this is implemented by a function `CompareTo: 'T X 'T -> int` that has 'T as **input**
  - An `IComparable<Animal>` function will certainly be able to compare Cats, so `IComparable<Animal>` can implement `IComparable<Cat>`
  - `IComparable<A>` will implement `IComparable<B>` if A is a supertype of B
  - **In** `IComparable<'T>` `'T` is a **contravariant** parameter

$$\begin{tikzcd}\text{Animal} \arrow[d] & \text{IComparable<Animal>} \\ \text{Cat} & \text{IComparable<Cat>} \arrow[u]\end{tikzcd}$$

F# and OOP vs FP polymorphism

- Ideally we should combine **inclusion** and **parametric** polymorphism
    - Bounded polymorphism. Allow a subtype constraint on any polymorphic wildcard
    - `'a when 'a :> S` *// F# flexible type*. **`'a`** is constrained to be subtype of **S**

upcasting

- Data structures that can be mutated make this matter more complex because the type constraint on **assignment** to a function output is contravariant.
    - F# does not allow contravariant constraints: **`S :> 'a`** *// not allowed*
- Different OOP languages adopt different strategies
- Balance:
    - **Expressivity**
    - **Lack of noise**: don't want to have to specify types all the time, so want accurate type inference
    - **Static safety**: would like all possible errors to be detected by compiler

Reference:

Fsharp flexible types

F# type system – fine points

- The value restriction
  - In some circumstances (see next slide) automatic type generalisation **causes problems**
  - This is inevitable in the Hindley-Milner type system if side effects are allowed
  - What seems like a compiler bug in F# is a way to step round this
- Lack of more expressive polymorphism
  - How can you write a "generic" map function that will work over all data structures?
  - Haskell **type classes** are the feature not available in F#
  - F# provides pragmatic (less clean, but usable) equivalents for many but not all use cases
  - See this very good introduction from Liam McLennan.
  - See F# typeshape for a pragmatic solution using F# features

Don Syme interviewed on F# Design

Do you see yourself as fundamentally a purist or a pragmatist?

I've never been asked that before. :-) To a purist, then I'm pragmatist, to a pragmatist I probably come across as a purist. I'm probably just in the middle. :-)

My job has definitely been to bridge the gap between the academically-oriented programming worlds and industry. If you go back to 2004, in our summary talks on .NET generics, we've got these slides where there was academia on one side and industry on the other and a huge divide in-between.

The F# value restriction

```
let v = ref [] // does not compile (try it)

let v<'T> : 'T list Ref = ref [] //ok

let v := [1] // ok

let x: int list = !v  // ok

val x: int list = [] //where has the [1] gone?
```

- The problem with **v** is that the Ref cell it is assigned does not have a known type: **'T** is generic.
  - Therefore the Ref cell cannot be created until after it is assigned.
  - v is actually a *type function* which returns the Ref cell only when the type is known.
  - It will therefore return a new `[]: 'T Ref` cell each time it is used, because before use the type is generic and each use could have a different type and therefore must be a different cell!

- Further information here

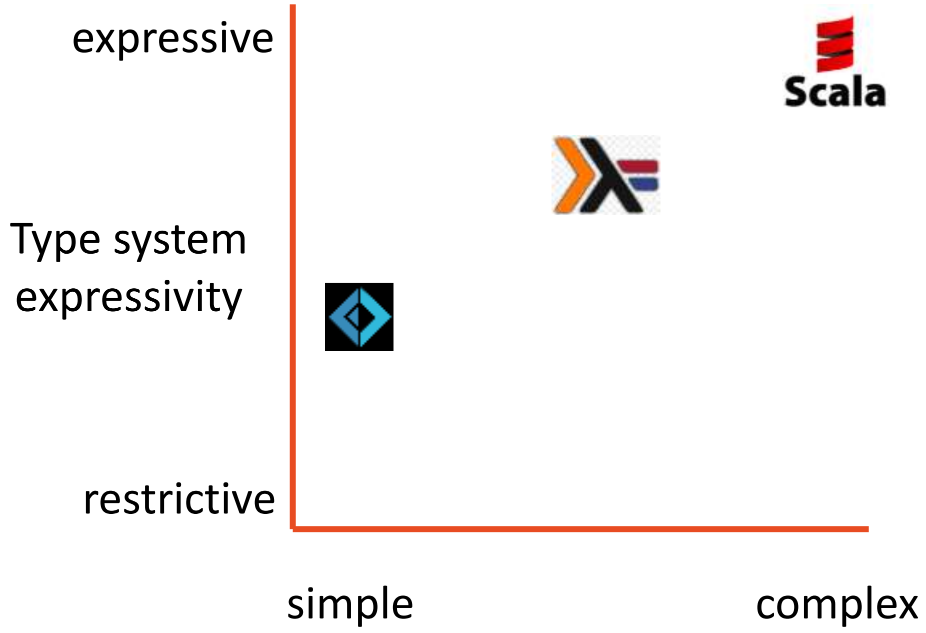
- This problem only happens because ref is mutable and therefore a copy is semantically different from a reference
- Any pure functional type and there is no trouble
- The F# compiler refuses to compile (without type annotation) cases like this when it cannot be sure there is no mutable value from the immediate function.
- This (mostly) is the right thing to do

Hybrid Static Type Systems

- Want to use FP + OOP types

  => there is a type system trade-off as for OOP

- F# goes for relative simplicity
  - "Functional first"
  - Less expressive
  - Can use escape hatches at some cost in safety
- Scala goes for complexity
  - True hybrid, or arguably OOP first
  - Type system much more expressive than F#
  - More conceptually difficult
  - More annotation needed
- Haskell
  - Is not a serious hybrid language: "Functional so pure it is loved or hated"
  - Adds optional complexity (type kinds) and type annotations for more expressivity in FP only programming


expressive
Type system expressivity
restrictive
simple
complex
Scala