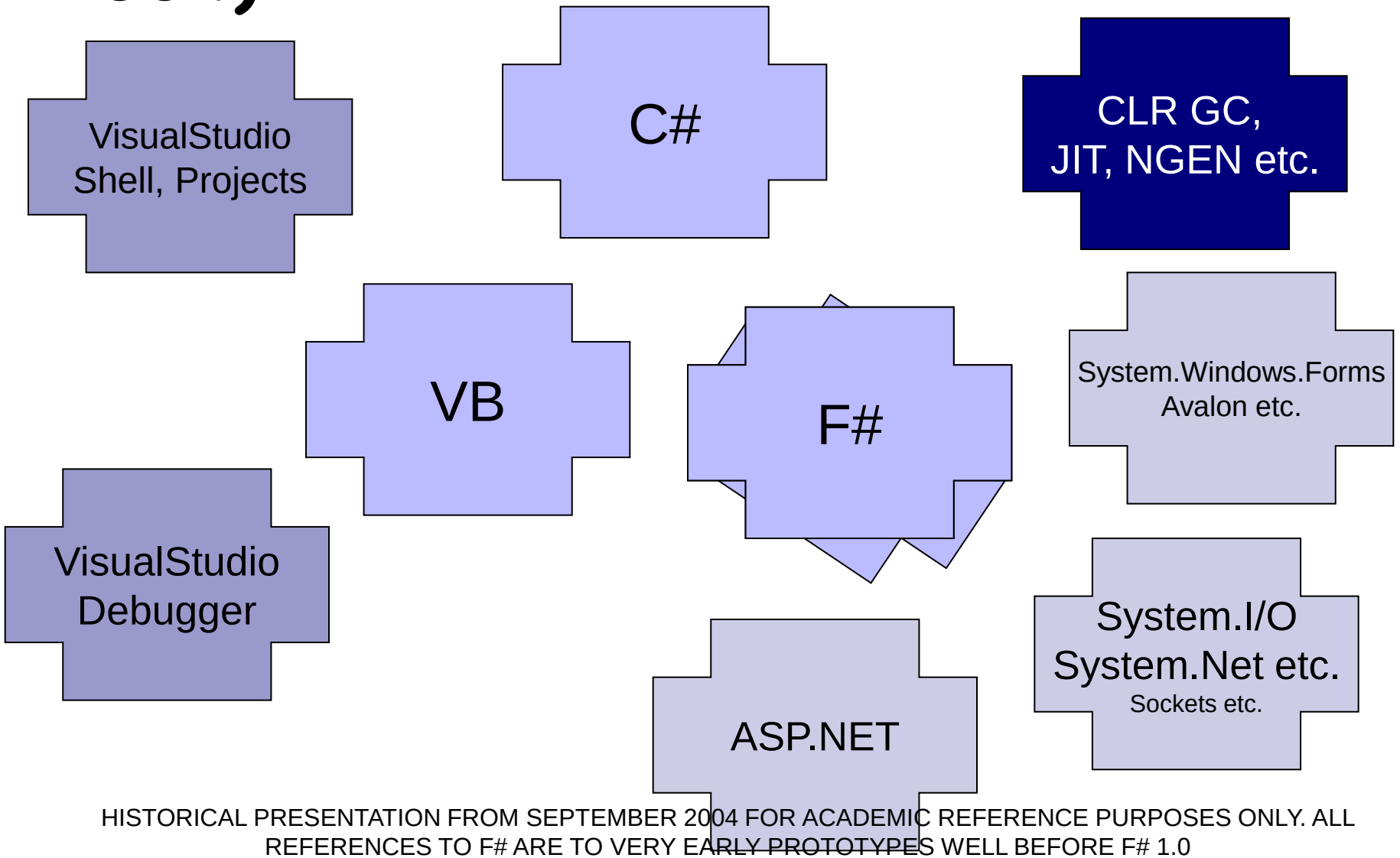
F#, GUI Programming and the Problem of Mutually Referential Objects in ML-style Programming

Don Syme
MSR Cambridge

Topics for today (Sept 2004)

- A brief introduction to F#
  - goals, project status etc.
- An F# sample
- An experimental extension related to value recursion
  - Motivated by and used in the sample
  - "Value recursion in a strict language when initialization effects cannot be statically controlled"

F# = ML as an equal player (Sept 2004)



What does ML offer over C#? (Sept 2004)

- Type inference
- Tuples, lists
- Discriminated unions
- Inner definitions
- Functions as first-class values
- Simple but powerful optimization story
- Explicit mutual dependencies and self-reference (e.g. no 'this' pointer by default)
- Immutability the norm
- However the same "basic model", e.g. w.r.t I/O, effects, exceptions

What does ML offer over C#? (Sept 2004)

- I remain convinced that ML-style programming is superior as the "core" of an approach to programming
- E.g. less is more: design is easier in ML
  - Only a handful of techniques to learn
  - Many types definitions just disappear if you make good use of tuples, lists, generics, discriminated unions
  - Reduced design/coding/testing-time for common idioms
- However I am also convinced there are real problems "around the edges"

What does F# offer over ML? (Sept 2004)

- e.g. NJ SML or OCaml
- Libraries galore
- Tools
- Bi-directional interop with C#, VB etc.
  - All public ML types and code can be immediately be used from C# etc.
  - All CLS constructs can be used from F#
  - F# is (almost) a "CLS Extender" language
- Bi-directional interop with C/COM
  - Wrap the C using C#, and call it from F#
  - Easier and less error-prone than the OCaml FFI
- Can build DLLs
  - Hard to do with OCaml and friends
  - F#'s compiled binary form is now essentially stable, hence versionable
- Multi-threading
  - Even OCaml is largely single-threaded, e.g. differentiates "ML threads" from others
- No significant runtime components
  - GC etc. is not part of the package, so much simpler

What does F# offer over ML? (Sept 2004)

- Missing:
  - No OCaml-style "objects"
  - No higher-kinded polymorphism (think "template template parameters")
  - No modules-as-values.
  - A slightly weaker notion of type abstraction
  - Some loss of raw computational performance (as good as C#, but 2-4x from OCaml native)
    - But functional programming will live or die on overall productivity and a broader notion of performance

F# Status (Sept 2004)

- Version "1.0" in preparation
- Will be used by SDV team to help them dig themselves out of their "OCaml hole"
- Very stable core language and compiler
  - Some interesting language work will go on around the edges, e.g. ML-modules-without-ML-modules
- VS integration prototype available internally
- ML compatibility library
  - A set of modules with design similar to a subset of the OCaml 3.06 standard modules
  - Permits cross-development of OCaml/ML code
- Samples etc.
- Some tools in progress
  - Lexer/LALR Parser Generators, Dependency Analysis

Some useful things to know (Sept 2004)

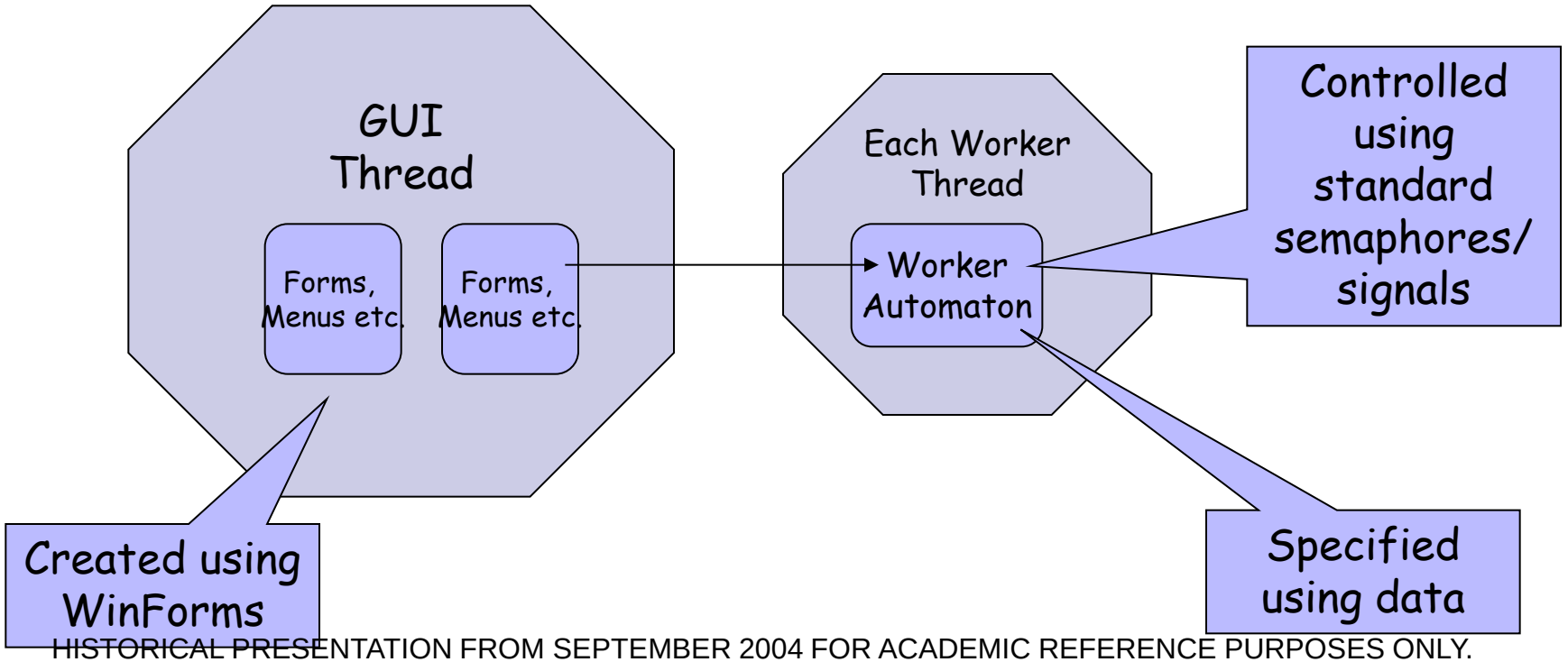
- ML's notion of an "object" is really a value which owns/manages/hides some state (i.e. a handle)
  - Much more like C than C++
  - Forget all about virtual methods, inheritance etc. You don't need them.
- ML idioms used today:
  - "let x = … in" -- let-binding
  - "match x with …" -- pattern matching
  - "let rec … and … in …" -- a set of recusrive bindings
  - "ignore(a)" -- discard value
  - "let f x y z = …" -- currying and partial application
  - "let f () = …" -- unit == "()" == void
  - "(fun x y -> …)" -- lambda expressions
  - "Data(data1,…,dataN)" -- allocate a value
  - "let x = ref 0" -- allocate a reference cell
  - "Some(x)" or "None" -- optional values
  - "let x = ref (Lazy(fun () -> expr))" -- laziness by data + explicit delays + holes
  - "lazy expr" and "force expr" -- syntactic sugar
- F# extensions used today:
  - "let x = new MyCSharpClass(…)" -- making an object
  - "let x = new MyCSharpDelegate(fun x y ->…)" -- making a delegate (a named function type)
  - "MyCSharpClass.StaticMethod(…)" -- calling a method
  - "x.Method(…)" -- calling a method
  - "x.Property" -- getting a property
  - "x.Property <- value" -- setting a propery
  - "(x :> MyCSharpBaseType)" -- upcast coercion

F# Observations (Sept 2004)

- "An ML I can use without hurting other people in my team"
- Surprisingly F# appears to be a better **client** of the .NET libraries than a language for **authoring** .NET libraries, i.e.
  - excellent for using .NET libraries
  - excellent for writing ML libraries
  - Can use ML libraries as .NET libraries, but could be better (compiled library interface is not entirely what a C# programmer would expect)
- So the niche seems to be for writing sophisticated applications
  - probably making use of the .NET components
  - probably with a symbolic processing component
  - probably with some components written in C# etc.
  - probably with some high-value components written in F# and "spun-off" along the way

An F# sample (Sept 2004)

- Glossing over some points which will be addressed later in the talk



Part II: Value recursion in strict programming

Summary

- Forget subtyping. Forget inheritance. The restrictions on self-referential and mutually-referential objects is what makes ML a poor GUI programming language.
  - At least when driving reasonable libraries such as System.Windows.Forms
  - The problem gets worse the more "declarative" a library gets
  - The others may be problems when it comes to authoring a GUI API
  - The problems also crop up elsewhere
- In an ideal world we would solve this by redesigning all APIs
  - e.g. introduce additional delays
  - e.g. annotate them with complex types describing the fact that they do not "tie knots"
- Practically speaking
  - C# "solves" this through a mishmash of implicit nulls and/or "create-and-configure" APIs.
  - ML "solves" it in a similar way.
  - Haskell has little choice but to heavily annotate and re-design the APIs.
- F# permits the above techniques, but also offers another "solution"
  - Limited but very useful self-referentiality for values through an augmented "let rec" construct
  - Compiler gives warnings when this is used (think "incomplete pattern match")
- F# is a great setting for exploring the practical implications of alternative approaches to this problem

Self-referential functions (recursion)

"rec" required

```
let x = x + 1
```

Not a function

```
let rec x = x + 1
```

"rec" required

```
let f x = if x < 2 then 1 else f(x-1) + f(x-2)
```

✓

```
let rec f x = if x < 2 then 1 else f (x-1) + f (x-2)
```

✓

```
let rec f(x) = if x < 2 then 1 else g(x-1) + g(x-2)
and g(x) = if x < 2 then 3 else f(x-1)
```

For this talk "self-referential" and "mutually-referential" are used interchangeably, and refer to self/mutually-referential VALUES, rather than not recursive functions.

Self-referential values/objects

- Self-referential non-stateful objects are not so interesting
- Self-referential stateful objects are very common
  - GUIs and other reactive machines
  - Indeed this is the heart of OO programming (forget inheritance!)
  - Closures are not a substitute
  - Haskell-style laziness probably is, if all libraries are designed with this in mind
- Dealing with self-referentiality
  - Three games:
    - inheritance and "self" – but this does not deal with mutual references
    - "create then configure + null pointers" – two phase initialization via mutation
    - "insert delays" – self-references permitted via lazy computations and recursive calls
  - Scripting languages: null pointers, "create and configure"
  - SML: build graphs of data via mutation
  - OCaml: a few more tricks available

Self-referential values via mutation

A graph of values (almost)

```
type node = { data: int;
              next: node ref }

let dummy = {data=0; next= ref(???) }
let x1 = {data=3; next=dummy}
let x2 = {data=4; next=dummy}
let x3 = {data=5; next=dummy}
x1.next := x2;
x2.next := x3;
x3.next := x1;
```

Self-referential values via mutation

A graph of values
built via holes+mutation

✓

```
type node = { data: int;
              next: node option ref }

let dummy = {data=0; next=ref(None)}
let x1 = {data=3; next=ref(None)}
let x2 = {data=4; next=ref(None)}
let x3 = {data=5; next=ref(None)}
x1.next := Some x2;
x2.next := Some x3;
x3.next := Some x1;

let rec iterate (n: node) =
  printf "%d;" n.data;
  iterate(getSome !(n.next)))
```

Types must be designed with holes+recursion in mind

This is a typical "create-and-configure" API

Self-referential values via delays

"force" required ✗

```
let rec x = Lazy(fun () -> x + 1)
→ error
```

✓ (ocaml)

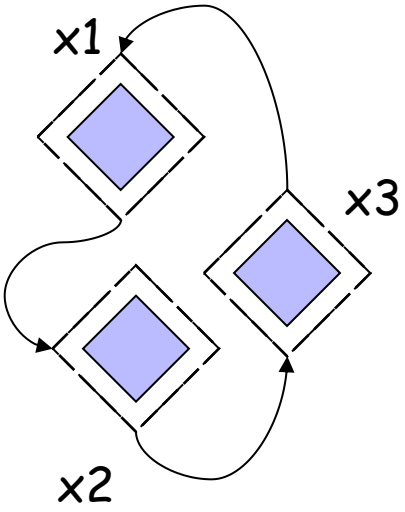
```
let rec x = Lazy(fun () -> (force x) + 1)
→ x: int Lazy
```

As with mutually recursive closures OCaml can "wire up" the self-reference at compile-time

== (ocaml)

```
let rec x = lazy (force x + 1)
```

A graph of lazy values



✓

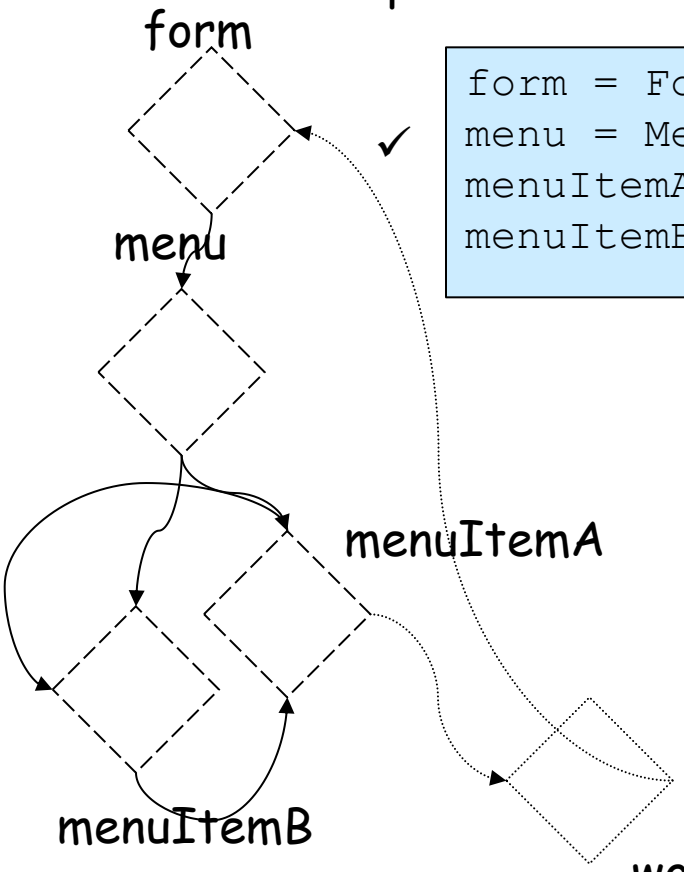
```
type node = Node of int * node lazy
let rec x1 = lazy(Node(3,x2))
    and x2 = lazy(Node(4,x3))
    and x3 = lazy(Node(5,x1))

let rec iterate (n: node lazy) =
  let Node(n,next) = force n in
  printf "%d;" n;
  iterate(next)
```

Really just an alternative way of specifying a "hole"

GUI elements are highly self-referential

A specification:

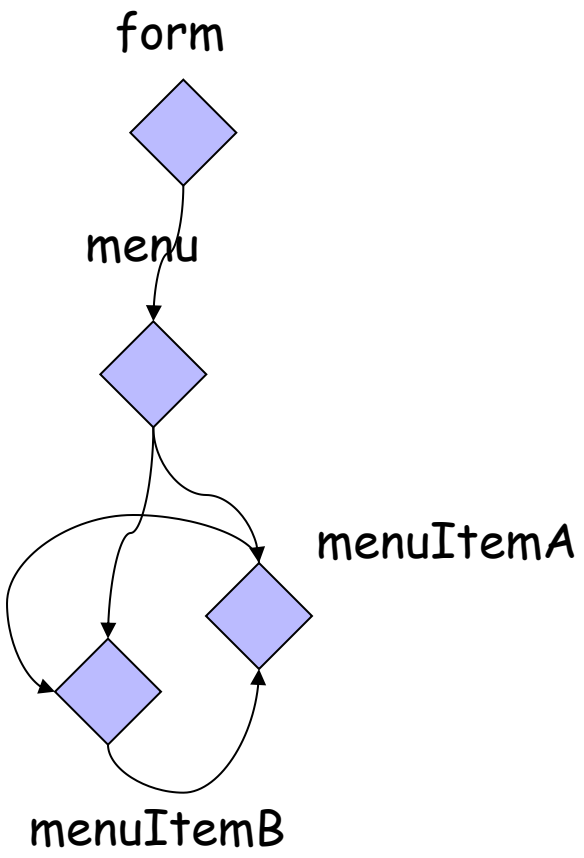



```
form = Form(menu)
menu = Menu(menuItemA,menuItemB)
menuItemA = MenuItem("A", {menuItemB.Deactivate} )
menuItemB = MenuItem("B", {menuItemA.Activate} )
```

This smells like a small "knot". However another huge source of self-referentiality is that for all GUIs forms have a message queue and a thread affinity, and so messages from worker threads and the thread pool must be pumped via reference to the form.

"Create and configure" in C#

Rough C# code, if well written:




```
class C
{ Form form;
  Menu menu;
  MenuItem menuItemA;
  MenuItem menuItemB;
  C() {
        // Create
        form = new Form();
        menu = new Menu();
        menuItemA = new MenuItem("A
        menuItemB = new MenuItem("B
        // Configure
        form.AddMenu(menu);
        menu.AddMenuItem(menuItemA);
        menu.AddMenuItem(menuItemB);
        menuItemA.OnClick +=
            delegate(Sender object,EventArgs x)
                { … };
        menuItemB.OnClick += …
        // etc.
  }
}
```

✘ Anonymous delegate syntax is gross

✘ Null pointer exceptions possible (Some help from compiler)

✘ Lack of locality

✘ In reality a mishmash – some configuration mixed with creation.

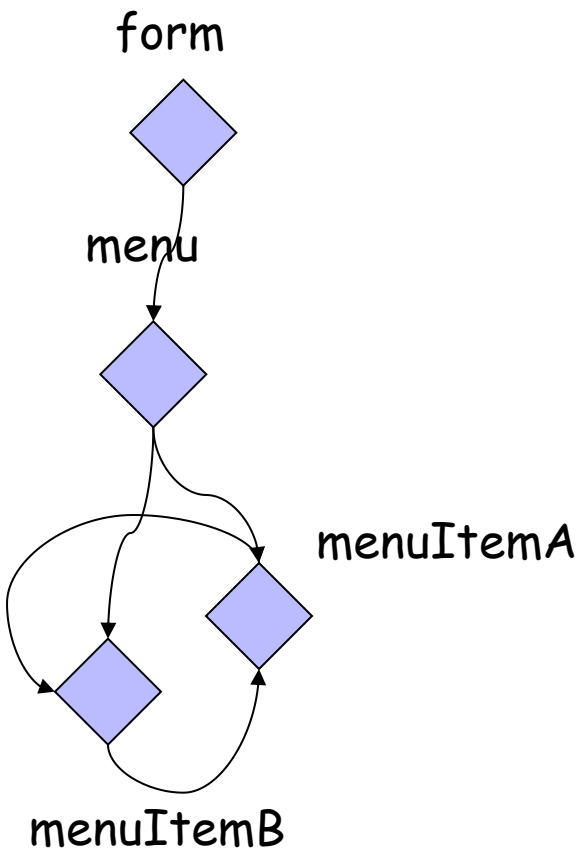
✘ Need to use classes

✘ Easy to get lost in OO fairyland (e.g. virtuals, inheritance, mixins)

✓Programmers understand null pointers

✓ Programmers always have a path to work around problems.

"Create and configure" in F#



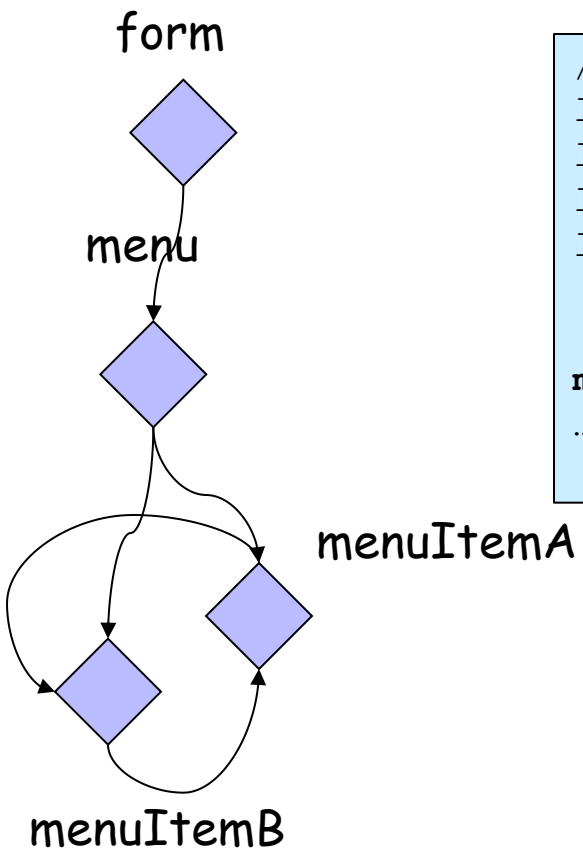

```
// Create
let form = new Form() in
let menu = new Menu() in
let menuItemA = new MenuItem("A") in
let menuItemB = new MenuItem("B") in
...
// Configure
form.AddMenu(menu);
menu.AddMenuItem(menuItemA);
menu.AddMenuItem(menuItemB);
menuItemA.add_OnClick(new EventHandler(fun x y -> …))
menuItemB.add_OnClick(new EventHandler(fun x y -> …))
```

- Lack of locality for large specifications

- In reality a mishmash – some configuration mixed with creation

"Create and configure" in F#

Often library design permits/forces configuration to be mixed with creation. If so it ends up a mishmash.




```
// Create
let form = new Form() in
let menu = new Menu() in
let menuItemB = ref None in
let menuItemA =
    new MenuItem("A",
      (fun () -> (the !menuItemB).Deactivate()) in
menuItemB := Some(new MenuItem("B", ...));
…
```

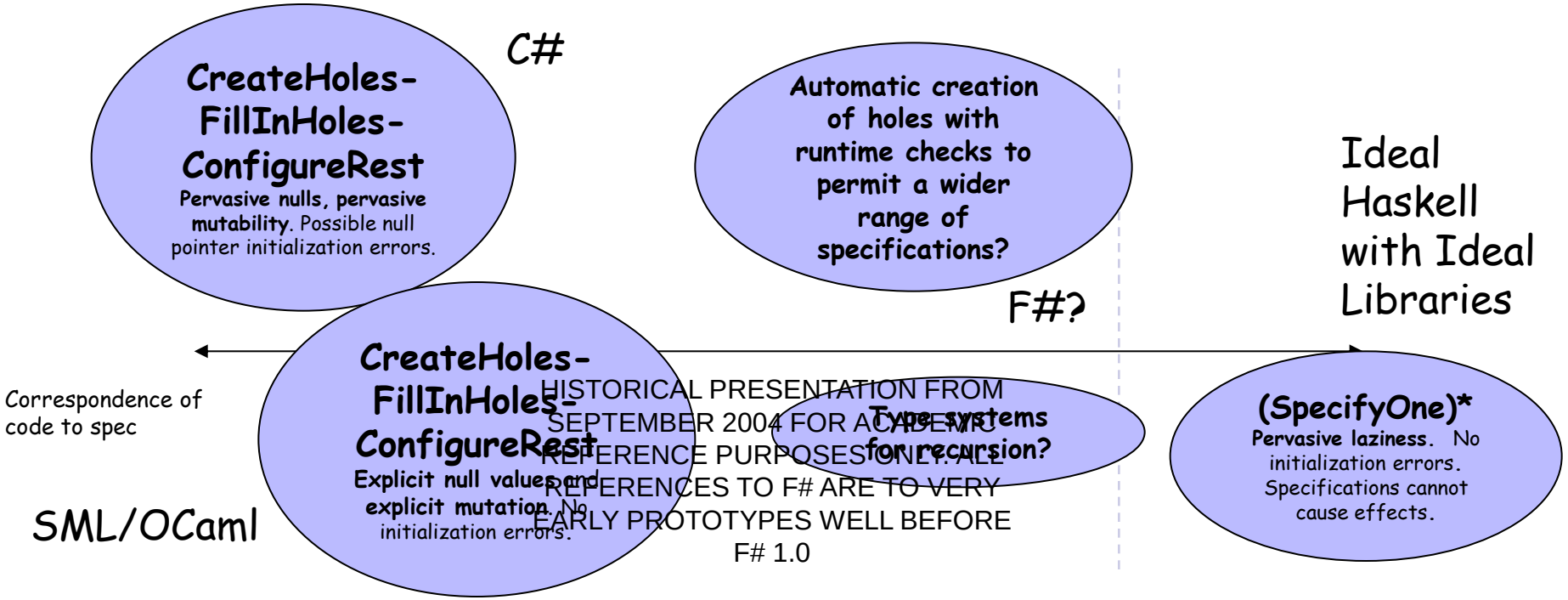
We manually build "holes" to fill in later to cope with the value-recursion

✓Programmers understand ref, Some, None.

✖ Programmers normally hand-minimize the number of "ref options", so the structure of their code depends on solving a recursive puzzle.

Recap

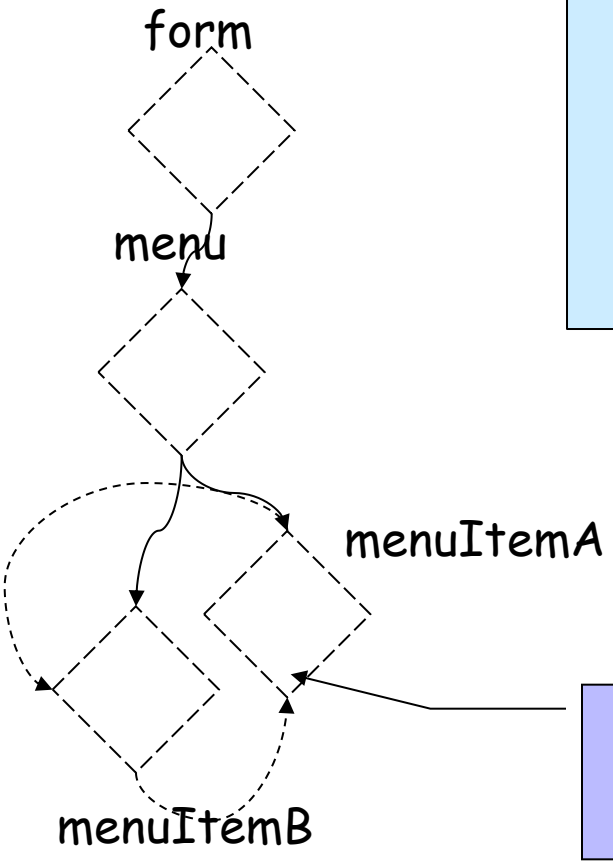
- Self/Mutual-referential objects are a real problem
- They are one of the real reasons for null pointers and create-and-mutate APIs and OO



F#'s experimental new mechanism

```
let rec form = new Form(menu)
    and menu = new Menu(menuItemA, menuItemB)
    and menuItemB =
      new MenuItem("B",
        (fun () -> menuItemA.Deact
    and menuItemA =
      new MenuItem("A",
        (fun () -> menuItemB.Deact
```

The goal: a general semi-safe mechanism to cope with the very common case where self/mutual-references are not evaluated during initialization.



- Expression values force EAGER evaluations of nodes in a graph of LAZY computations
- Runtime initialization errors will occur if the creating the objects causes a self-reference during initialization

i.e. these are not initialization-time references, hence OK

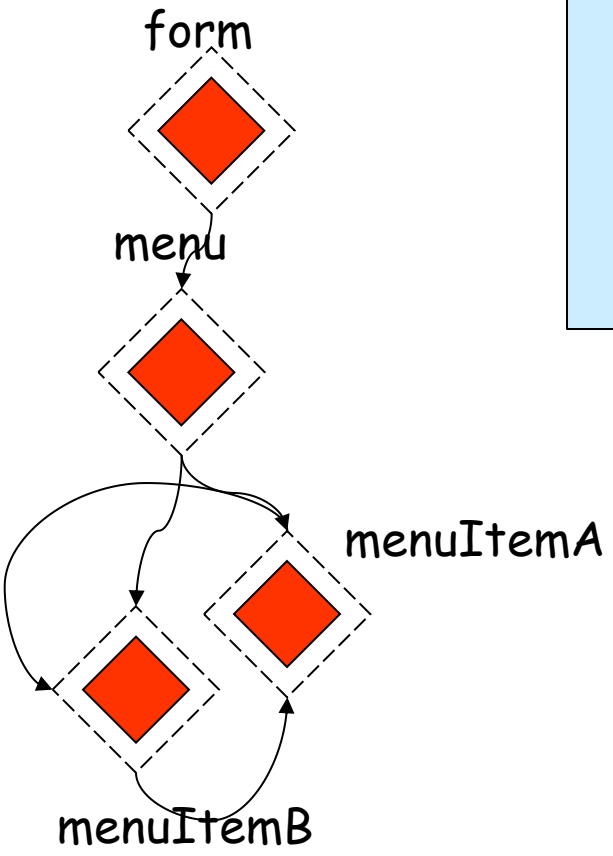
But we cannot statically check this without knowing a lot about the MenuItem constructor code

F#'s experimental new mechanism

```
let rec form = new Form(menu)
    and menu = new Menu(menuItemA, menuItemB)
    and menuItemB =
      new MenuItem("B",
        (fun () -> menuItemA.Deact
    and menuItemA =
      new MenuItem("A",
        (fun () -> menuItemB.Deact
```

Also happen to permit some forward references.

The goal: a general semi-safe mechanism to cope with the very common case where self/mutual-references are not evaluated during initialization.



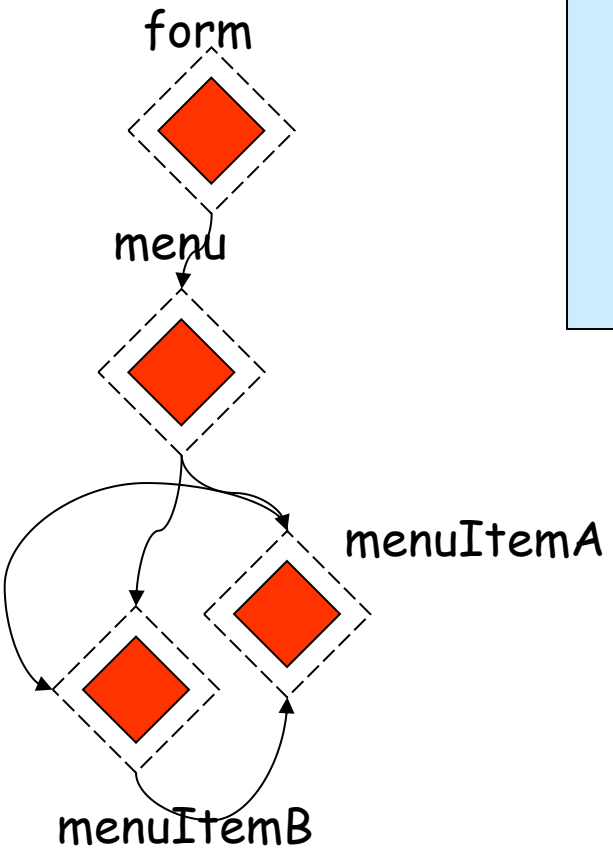
- Expression values force EAGER evaluations of nodes in a graph of LAZY computations
- Thus the "let rec" specifes a tuple of lazy values and eagerly evaluates them one by one
- Runtime initialization errors will occur if the creating the objects causes a self-reference

F#'s experimental new mechanism

```
let rec form = new Form(menu)
    and menu = new Menu(menuItemA, menuItemB)
    and menuItemB =
      new MenuItem("B",
        (fun () -> menuItemA.Deact
    and menuItemA =
      new MenuItem("A",
        (fun () -> menuItemB.Deact
```

Also happen to permit some forward references.

The goal: a general semi-safe mechanism to cope with the very common case where self/mutual-references are not evaluated during initialization.



- Expression values force EAGER evaluations of nodes in a graph of LAZY computations
- Thus the "let rec" specifes a tuple of lazy values and eagerly evaluates them one by one
- Runtime initialization errors will occur if the creating the objects causes a self-reference

F#'s experimental new mechanism

- Errors if evaluation is statically determined to always cause a failure (actually more conservative: assumes any branch of conditionals may be taken)

```
let rec x = y
    and y = x
```

mistake.fs(3,8): error: Value 'x' will be evaluated as part of its own definition. Value 'x' will evaluate 'y' will evaluate 'x'

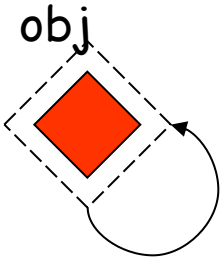
- Warnings if the mechanism is used at all

```
let rec x = new MenuItem("X", new EventHandler(fun sender e -> x.Enabled <- false))
```

ok.fs(13,63): warning: This recursive use will be checked for initialization-soundness at runtime.

"let rec" (on values): objects

```
let rec obj = {new Object() with
                  GetHashCode() = obj.ToString().Length }
```



"Self" drops out for free within methods, with a compiler warning.

Expect this to be rarely to be used (indicating how useless "self" is for F#.)

Observations about the F# mechanism

- It is incredibly useful
  - Immediately helped me "scale up" samples without correspondence to the specification
  - Was able to take non-programmers through the code and the "got it", i.e. saw the mapping to what was on the screen
  - A relatively inexperienced programmer was able to take the sample and modify it
- If as on the previous slide then it's almost certainly too subtle
  - Programmers would probably hang themselves, though the warnings may be enough.
  - Bindings can be executed out-of-order (though this could be considered an error)
- It can be optimized
  - Neither references nor laziness escape in any "real" sense, hence scope for optimization
- It may be too limited
  - What if you need an array of menu items? You have to be careful: do you need "an strict array of lazy items", or "a lazy array of strict items"?
  - However if there are no initialization-time requirements between the items in the array then things will still work out
  - What about three-phase or multi-phase initialization? Any examples?

Tweaks to the F# mechanism

- Concurrency:
  - If initialization starts a thread then you have potential problems.
  - Really need to prevent leaks to new treads/thread-pool items as a side-effect of initialization
  - Raises broader issues for a language (e.g. "the creation of new concurrency contexts must be made explicit and tracked by the type system")
- What to do to make things a bit more explicit?
  - My thought: annotate each binding with "**lazy**"
  - Byron's suggestion: annotate each binding with "**eager**"
  - Interesting!

```
let rec eager form = new Form(menu)
    and eager menu = new Menu(menuItemA, menuItemB)
    and eager menuItemB =
      new MenuItem("B",
        (fun () -> menuItemA.Deactivate())
    and eager menuItemA =
      new MenuItem("A",
        (fun () -> menuItemB.Deactivate())
```

Observations about the F# mechanism

- It works well as an augmentation of ML's existing "let rec"
- Each binding can be an arbitrary computation. This allows configuration to be co-located with creation.

```
let rec eager form = new Form()
    and eager do form.Add(menu)

    and eager menu = new Menu()
    and eager do menu.Add(menuItemA)
    and eager do menu.Add(menuItemB)

    and eager menuItemB = ...
    and eager menuItemA = ...
```

An area in flux

- SML 97: restricts to recursion between functions
  - Function values thus have a very special status in the language: if you want mutually-self-referential objects then you must explicitly represent them as code, not values
- OCaml 3.0X: adds recursion for some concrete data
  - Data constructors now also have a special status in the language
  - Leads to even more constructs with "poor abstraction properties"
  - However it is safe and conservative: no runtime errors
- Moscow ML 2.0: adds mutual-recursion for modules
  - Sort of like F#'s "let rec eager" but only one value can be bound (not a tuple)
  - Main focus was on mutually-referential modules, not values
  - Runtime initialization errors possible
- Haskell: Always permitted arbitrary self-referentiality, somewhat checked at runtime (blackholes)
  - Problems with threading
  - Laziness everywhere
- Various papers: "a theory of well-founded recursion"
  - Effectively you have to prove the recursion well-founded e.g. orderings
  - Had a friend spend an entire PhD doing a handful of this sort of proof
  - Can be worse than termination proofs.
  - Paper also suggests using lazy thunks

Questions?