Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge

Don Syme [1]

Microsoft Research, Cambridge, U.K.

Abstract

Mutual dependencies between objects arise frequently in programs, and programmers must typically solve this value recursion by manually filling "initialization holes" to help construct the corresponding object graphs, i.e. null values and/or explicitly mutable locations. This paper aims to augment ongoing theoretical work on value recursion with a description of a semi-safe mechanism for a generalized form of value recursion in an ML-like language, where initialization corresponds to a graph of lazy computations whose nodes are sequentially forced, requiring runtime checks for soundness during initialization in the style of Russo. Our primary contribution is to use the mechanism to develop compelling examples of how the absence of value recursion leads to real problems in the presence of abstraction boundaries, and give micro-examples that characterize how initialization graphs permit more programs to be expressed in the mutation-free fragment of ML. Finally we argue that in heterogeneous programming environments semi-safe variations on value-recursion may be appropriate for ML-like languages, because initialization effects from external libraries are difficult to characterize, document and control.

Key words:
ML, Value recursion, Initialization Graphs, GUI Programming

1 Introduction

One of the primary goals of a programming language is to permit the authoring of programs in a form that corresponds closely to an informal specification. For example, the following is an informal specification of a GUI form (i.e. window) where each menu item toggles the activation state of the other.

> A form f with title "Form" containing a Menu m with title "File" containing two menu items mi_1 and mi_2 with titles "Item1" and "Item2" where selecting mi_1 toggles the activation state of mi_2 and likewise selecting mi_2 toggles the activation state of mi_1.

[1] Email: `dsyme@microsoft.com`

We would like to program this in a safe language without either mutation or null pointers. In this paper we first describe a "dead simple" mechanism for value recursion in an ML-style language. This mechanism has its problems, but will let us write the above program as follows (we use an OCaml-like syntax):

```
let rec f = createForm("Form", [ m ])
    and m = createMenu("File", [ mi1; mi2 ])
    and mi1 = createMenuItem("Item1", λ(). toggle(mi2))          (A)
    and mi2 = createMenuItem("Item2", λ(). toggle(mi1))
```

Here we have assumed the API: [2]

```
type MenuItem    type Menu      type Form
val createForm: string * Menu list -> Form
val createMenu: string * MenuItem list -> Menu
val createMenuItem: string * (unit → unit) -> MenuItem
val toggle: MenuItem -> unit
```

The client program is not permitted in ML-family languages, because ML statically enforces a strong notion of initialization soundness, in particular that recursive bindings won't fail or have side-effects at all. To work around this the programmer codes an explicit initialization hole:

```
let the = function | Some v → v | None → failwith "bug"
let mi2 = ref None
let mi1 = createMenuItem("Item1", λ(). toggle(the(!mi2)))
let _ = mi2 := Some(createMenuItem("Item2", λ(). toggle(mi1)))
let m = createMenu("File", [mi1; the !mi2] )
let f = createForm("Form", [m])
```

The absence of value recursion has forced the programmer to rely on mutation and failure to write simple programs, even in a "safe" functional language. [3]

1.1 Contributions of this paper

The contributions of this paper are as follows:

- We describe how the emerging "platform-oriented" versions of ML that rely on external libraries (e.g. F#, MLj and SML.NET [22,2]) are exposed to the value recursion problem, because it is impossible to characterize the side-effects that may occur during the construction of rich objects through abstract external APIs.
- We describe a form of value recursion using runtime-checks and laziness through a formal calculus (§2) and prove some simple properties of this system. Although the principle of using laziness to mediate recursion is well-known, we have not seen this particular system proposed as an interpretation of recursive bindings in the core of a strict language.

[2] We use `λ().` for "`fun () ->`".

[3] Another alternative is the use of a reference cell holding a function value initialized to a polymorphic failing dummy value `let dummy _ = raise "bug"`.

- We give an example of a program that cannot be defined in the mutation-free fragment of ML, but can be defined using this form of value recursion.
- We give compelling and previously un-noted examples of how the lack of value recursion forces programmers to break abstraction boundaries (§3-§5), including an analysis of the *picklers* example from the literature due to Kennedy [13]. We also document for the first time the relationship between GUI programming and value recursion (§6).

We conclude with related work and future directions (§8-§9). Notice that this paper is *not* about imposing static controls on value recursion (a standard Core ML type system is assumed throughout). The static control of value recursion is a crucial goal, e.g. see [3,4,5,20,11] for discussions of attempts to control value recursion in the context of modules and mixins. Instead, this paper argues that a mix of safe and semi-safe mechanisms still appears to be required, especially for languages deployed in heterogeneous multi-language environments.

2 Initialization Graphs

2.1 Initialization Graphs via Explicit Uses of Laziness

Consider the following transformation of the program (A) from §1: [4]

```
let rec f' = lazy createForm("Form", [ force m' ])
    and m' = lazy createMenu("File", [ force mi1', force mi2' ])
    and mi1' = lazy createMenuItem("Item1", λ(). toggle(force mi2'))
    and mi2' = lazy createMenuItem("Item2", λ(). toggle(force mi1'))
let f = force f'
let m = force m'
let mi1 = force mi1'
let mi2 = force mi2'
```

The bindings have become lazy computations, but only for the purposes of initialization. We call a `let rec` interpreted in this way an *initialization graph*. We first note that this approach has obvious problems:

- The nodes of the graph are explored on-demand, so evaluation order may be counter-intuitive. However evaluation order is still precisely defined, and all nodes are eventually evaluated, provided no errors occur.
- Initialization graphs that result in cyclic initialization-time dependencies cause runtime errors. Runtime checks are needed for this condition.

Initialization graphs are a form of the *unrestricted recursion* proposed by Dreyer in the context of recursive modules for Standard ML [5]. Initialization graphs are a blunt and simple approach to value recursion, and the general observation that laziness can be used to encode value recursion is well known

[4] Here `lazy` and `force` generate and consume delayed computations of the type α `lazy`, defined using an appropriate discriminated union and reference cell.

(e.g. see discussion in [11]). However, no one seems to have documented a formal calculus that uses laziness (as opposed to null-pointers) as the basis for value recursion. Furthermore, compelling examples of the importance of value recursion have been missing from the literature: for example, the interaction between mixin modules and value recursion has been studied [11,3,4], but the examples given are just simple recursive functions. Furthermore, semanticists have assumed that unrestricted recursion is an evil to be avoided at all costs, but it occurs in practice as part of the dynamic linking semantics of initialization for C# and Java (see §11), and thus exploring options for unrestricted recursion appears sensible as an adjunct to strictly static techniques.

2.2 Terminology: Immediate and Delayed Dependencies

We make the following distinctions, similar to those made by Dreyer [5]:

- When execution of the bindings of a `let rec` evaluates a reference to a recursively bound variable we say an *immediate dependency* has arisen, i.e. if we evaluate a reference to v that syntactically occurs in a binding for u then an immediate dependency has occurred from u to v.
- After the bindings have been completed, closures may still reference these variables. We say the subsequent evaluation of these generates *delayed dependencies*, i.e. the evaluation of a reference to v that syntactically occurs in a binding for u generates a delayed dependency from u to v.

Immediate and delayed dependencies are dynamic notions: in general it is not possible to statically determine if a given syntactic occurrence of a recursively bound variable results in immediate or delayed dependencies, or even both (in the general case it is undecidable which parts of the initialization bindings will execute at all). Delayed dependencies are irrelevant to initialization soundness: they are purely part of the emergent behaviour of the object values being defined.

2.3 λ_I: A calculus for initialization graphs

This section presents a typed lambda calculus extended with initialization graphs. The language is defined by the grammar in Figure 1 and is standard apart from allowing arbitrary expressions on the right of recursive bindings. We incorporate the effectful action `print()`. The typing rules for this language are also standard and are not presented here. Non-standard are the evaluation environments and rules in Figure 2. [5] Evaluation environments are maps to locations rather than maps to values. We have omitted rules for `print()` and for propagating errors (see §2.7). The given calculus can be extended to

[5] We adopt an implicit rule that the overall output over the program is recorded in the state, though in general output plays no role in the semantics other than support a minimalist effectful operation.

v	$= id$	Variable
e	$= v$	Value/Node Reference
	$= e\ e$	Function Application
	$=$ `letrec` b_1 `and` ... b_n `in` e	Recursive Binding
	$=$ `fun` v `->` e	Lambda Abstraction
	$=$ `print()`	A simple effectful construct
b	$= v = e$	Binding

Fig. 1. Syntax for λ_I

Γ	$= id \to l$		Environment
σ	$= l \to V$		Initialization Graph State
V	$= v$		Evaluated Initialization Thunk
	$\mid (\Gamma, \lambda_0 e)$		Delayed Initialization Thunk
	$\mid$ `error`		Initialization Error
v	$= (\Gamma, \lambda x.e)$		Closure Value
	$f \oplus (x \mapsto y)$		Function extension

$$\frac{\Gamma(x) = l \quad \sigma(l) = v}{\Gamma, \sigma \vdash x \rightsquigarrow v, \sigma} \qquad \frac{\begin{array}{l}\Gamma(x) = l \\ \sigma(l) = (\Gamma', \lambda_0 e) \\ \Gamma', \sigma \oplus (l \mapsto \texttt{error}) \vdash e \rightsquigarrow v, \sigma' \\ \sigma'' = \sigma' \oplus (l \mapsto v)\end{array}}{\Gamma, \sigma \vdash x \rightsquigarrow v, \sigma''}$$

$$\frac{}{\Gamma, \sigma \vdash (\texttt{fun}\ x\ \texttt{->}\ e) \rightsquigarrow (\Gamma, \lambda x.e), \sigma} \qquad \frac{\begin{array}{l}\Gamma, \sigma_0 \vdash e_1 \rightsquigarrow (\Gamma', \lambda x.e), \sigma_1 \\ \Gamma, \sigma_1 \vdash e_2 \rightsquigarrow v_1, \sigma_2 \\ l \text{ fresh} \\ \Gamma' \oplus (x \mapsto l), \sigma_2 \oplus (l \mapsto v_1) \vdash e \rightsquigarrow v_2, \sigma_3\end{array}}{\Gamma, \sigma_0 \vdash (e_1\ e_2) \rightsquigarrow v_2, \sigma_3}$$

$$\frac{\begin{array}{ll}l_i \text{ fresh} & \\ \Gamma' = \Gamma \oplus (x_i \mapsto l_i) & (1 \le i \le n) \\ \sigma_0 = \sigma \oplus (l_i \mapsto (\Gamma', \lambda_0 e_i)) & (1 \le i \le n) \\ \Gamma', \sigma_{i-1} \vdash x_i \rightsquigarrow v_i, \sigma_i & (1 \le i \le n) \\ \Gamma', \sigma_n \vdash e \rightsquigarrow v, \sigma' & \end{array}}{\Gamma, \sigma \vdash (\texttt{let rec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e) \rightsquigarrow v, \sigma'}$$

Fig. 2. Selected Rules from Operational Semantics for λ_I

include conditionals, non-recursive structured data, pattern matching, mutable state and I/O in completely standard ways – most examples in this paper will assume these extensions have been made. If recursive data is included then one must consider the interaction between immediately recursively-tied data and value recursion [12], which is beyond the scope of this paper. [6] We consider the issue of exceptions later.

The evaluation of a recursive binding initially assigns a new delayed computation for each variable, and then evaluates each thunk. Hence the execution of a recursive binding leaves no unresolved delayed computations, and thus the delayed computations do not escape their lexical scope, as captured by the following theorem: [7]

[6] Our prototype implementation supports both but demands that each `let rec` utilize either data recursion or be an initialization graph, but not both.

[7] Note the theorem is in terms of the quantity and state of initialization thunks in the thunk heap: something that cannot normally be observed.

Theorem 2.1 (Successful initialization eliminates initialization thunks) *Let* $T(\sigma) = \{l \mid \exists \Gamma, e.\ \sigma(l) = (\Gamma, \lambda_0 e)\}$. *Then* $\Gamma, \sigma \vdash e \rightsquigarrow v, \sigma'$ *implies* $T(\sigma') \subseteq T(\sigma)$.

The proof is by induction over the derivation, with an appropriate analysis at `let rec` bindings to prove that each fresh location is eventually assigned a completed value. A corollary is that the evaluation of a term from a state with no initialization thunks produces a state with no initialization thunks. We informally propose that a corresponding result holds when λ_I is extended to contain the full constructs of a typical ML-family language, excluding a construct to catch exceptions. Note that:

- Expressions never evaluate to delayed initialization thunks.
- Locations in the initialization graph are never aliased, since they are not directly referred by expressions.
- A recursive binding `let rec` x `=` e_1 `in` e_2 where x is not used in e_1 is equivalent to the traditional call-by-value interpretation of `let x =` e_1 `in` e_2.)
- If all expressions on the right of a `let rec` are immediate functions then we have the traditional semantics for `let rec`. (The initialization thunks reduce to closure values).

2.4 Expressivity

Initialization graphs are an extension to core ML: all existing core ML programs can be accepted without warnings and run with unchanged behavior. No initialization failures occur if the mechanism is not used. However, initialization graphs do extend the expressive power of variations of ML that include data abstraction but which do not include mutation. Consider the following module, defining streams where stream generators have identity. [8]

```
module type MSig = sig
  type t
  val mk: (unit -> t) -> t
  val next: t -> t
  val id: t -> string
end
```

```
module M : MSig = struct
  type t = T of string * (unit -> t)
  let mk f = T (gen(), f)
  let next (T (n,f)) = f ()
  let id (T (n,f)) = n
end
```

Given the above, it is not possible to generate a value of type t where:

```
id x1 = id(next(x1)) = id(next(next(x1))) = ...
```

without using mutation, i.e. an explicit initialization hole, e.g.

```
let x1hole = ref None
let rec x1 = mk (fun () -> the !x1hole)
let _ = x1hole := Some(x1)
```

However, initialization graphs allow us to define such a value:

[8] We assume a function `gen()` that generates fresh names (`print()` could also be used, where the number of outputs indicates the number of names generated).

```
let rec x1 = mk (fun () -> x1)
```

The best we can do without using value recursion or mutation is:

```
let rec xf () = mk xf
let x2 = xf ()
```

But each call to `next` now causes a call to `mk`, and thus multiple object identities, so `id(x1)` $\neq$ `id(next(x1))` $\neq$ `id(next(next(x1)))`. Note we have only achieved this increased expressiveness by using runtime-checking for initialization soundness. See §9 for further discussion of this example.

2.5 Static Warnings and Errors

λ_I permits nonsensical definitions such as `let rec x = x + 1` where the evaluation of `x` on the right-hand-side of the binding will cause an immediate exception. A technical report gives details of a simple static analysis that reports errors for many such cases [23]. See also Boudol [3,4] for an in-depth treatment of inference issues related to one particular static type system for value recursion.

Furthermore, initialization graphs should clearly be used with care, and so in our prototype implementation warnings are given whenever the mechanism is used. These warnings are similar to the "incomplete pattern match" warnings given by typical ML compilers: a possibility of unsoundness exists, and the programmer is informed of this.

2.6 Implementation Techniques

The initialization graph interpretation of `let rec` can easily be avoided completely for normal recursive functions, i.e. bindings x `=` e where e is a λ or some other delayed computation, so the performance of normal ML code is not impaired. These tend to cover nearly all `let rec` bindings in real programs, and we have yet to encounter a non-contrived program where an initialization graph dominates computation, because such graphs are usually used to specify GUIs and other reactive machines. However, they are easy to implement in practice: a simple transformation can convert recursive bindings into a target language that supports `lazy` computations, e.g. as provided in OCaml. Every expression of the form

`let rec` x_1 `=` e_1 `...` x_n `=` e_n `in` e

is transformed to

```
let x1,...,xn = let rec x'1 = lazy e'1 ... x'n = lazy e'n
                in (force x'1,...,force x'n)
in e
```

where each e'_i is formed by taking e_i and replacing all references to each x_j with `force` x'_j. That is, uses of value recursion are replaced by initialization thunks implemented as `lazy` computations, and references to recursively bound vari-

ables are replaced by `force` operations. All expressions in `let rec`s are now delayed computations, a form of recursive data supported by OCaml, F# and optionally in SML/NJ.

2.7 The awkward squad: value-carrying exceptions, concurrency and continuations

The accompanying technical report [23] addresses three additional topics: the semantics given for initialization actions that can throw exceptions (in particular ones that can carry sophisticated data values), start threads or capture continuations. Since our aim is to have a semi-safe mechanism that can complement static techniques, we follow the typical ML response to these problems: ML does not attempt to control effects, and the specification of the language simply defines the results of computations on a single thread.

In practice the use of effects is discouraged during initialization except in precisely those cases where the programmer judges that the effects are an essential part of constructing and wiring together the objects that form the recursive binding. For example, in the context of .NET or Java programming thread objects can be created and their properties specified, but the threads themselves should not be started. If a monadic system to control effects were added to ML then it would be desirable to restrict the monad associated with initializing computations (see §8 and [10]).

We mention in passing that ML's well known *value restriction* on type parameterization can also lead to problems when defining and using APIs. This simply restricts the class of let-bindings that can be given generalized polymorphic types, but means abstract APIs must sometimes declare constructs as computations rather than values. This is orthogonal to the issues discussed in this paper, and other statically typed programming languages such as Java have even more onerous restrictions.

3 Examples of Value Recursion: Caches

One of the main contributions of this paper is to use initialization graphs to present a series of compelling examples of value recursion where

- abstract APIs are required to create graphs of related objects; and
- the object graphs incorporate immediate and/or delayed dependencies, but where there are no cycles in the immediate dependencies.

Our first examples show how the lack of value recursion leads programmers to break abstraction boundaries. A frequent example of this in practical ML programming relates to caches. For example, consider a function cache with the following abstract API, where the first argument is a comparison function:

```
val cache: ('a -> 'a -> int) -> ('a -> 'b) -> ('a -> 'b)
```

Implementations can vary, e.g. the following naive attempt, assuming functions `mem_assoc` and `assoc` for using querying association lists with the given comparison function.

```
let cache cf f = let cref = ref [] in fun x ->
    if mem_assoc cf x (!cref) then assoc cf x !cref
    else let y = f x in cref := (x,y) :: (!cref); y
```

The following simple program attempts to cache calls to `even`:

```
let rec odd n = even(n-1)
and even = cache compare (fun n -> n = 0 or odd(n-1))
```

but is rejected due to `let rec` restrictions. A common result is that programmers reveal and/or duplicate their cache implementations, or simply avoid adding caching to recursive bindings even when it is, from a performance perspective, appropriate.

4 Examples of Value Recursion: Automata

For our next example we use initialization graphs to describe the mutually referential states of automata. We are particularly interested in cases where the implementation of automata states is hidden.

Assume we wish to programmatically specify an automaton that transitions between control states `paused`, `running` and `finish` in response to signals, e.g. Unix file handles or instances .NET's `WaitHandle` class. We could code such a specification using explicit calls to platform primitives, e.g. Unix's `select` or .NET's `WaitHandle.WaitAny`, the latter of type `WaitHandle array` → `int`. However there are advantages to using combinators and abstract values to represent the control states: the implementation of states can be uniformly augmented with additional tracing, caching, profiling and/or model-checking functionality without requiring changes to the specification. So instead assume we have the following API:

```
type State
type Signal = WaitHandle
type Transition = Signal * NextState
type NextState = unit → State
val waitAll: Signal list * NextState → State
val waitAny: Transition list → State
val peekOne: Transition list * NextState → State
val doThen: (unit → unit) * NextState → State
val finish: State
val run: State → unit
```

Here automaton are `State` objects with associated behaviour, that is, nodes in a programmatically-specified graph of states. A `waitAll` automaton blocks until all given signals have been set; a `waitAny` automaton selects between signals and commits to one selected transition; `peekOne` is `waitAny` with a zero-timeout and a default transition; `finish` terminates. Executing an automata

using `run` causes a thread to transition from state to state under the control of the signals. An automaton in a `doThen` state performs the given computation on this value and then proceeds to the next state (it does not respond to signals while performing the computation). The API uses computations for `NextState` values. A simple implementation of a portion of the API is:

```
type State = { runAction: unit -> unit; count: int ref }
let run st  = incr st.count; st.runAction()
let finish = { runAction=(λ(). ()); count=ref 0 }
let waitAny transitions =
  let waithandles = Array.of_list (map fst transitions) in
  let actions = Array.of_list (map snd transitions) in
  let runAction () =
    let i = WaitHandle.WaitAny(waithandles) in
    run  (actions.(i) ()) in
  { runAction = runAction; count = ref 0 }
```

Here the states are augmented with private tracing to count the number of times each state is entered. In addition the `waitAny` function pre-computes two arrays that are re-used during the actual execution of the graph.

The transitions of worker automaton can now be specified using initialization graphs. Here is one such graph (we assume supporting functions `reset` and `step` of type `unit → unit` that peform an underlying computation):

```
let rec initial = resetThenPause
and paused = waitAny [ stepSignal,  (λ(). stepThenPause);
                       resetSignal, (λ(). resetThenPause);
                       exitSignal,  (λ(). finish) ]
and stepThenPause = doThen(step, (λ(). paused))
and resetThenPause = doThen(reset, (λ(). paused))
let _ = run initial
```

This is a compact declaration of a set of mutually dependent abstract objects combined with their behaviour.

4.1 Automata in ML without initialization graphs

State machines are traditionally programmed using recursive functions, e.g.

```
let waithandles = [| stepSignal; resetSignal; exitSignal |]
let rec initial () = resetThenRun ()
and paused () =
  match (WaitHandle.WaitAny(waithandles)) with
  | 0 -> step(); paused()
  | 1 -> reset(); paused()
  | 2 -> ()
  | _ -> failwith "unexpected"
```

This avoids the `let rec` restriction by defining states as functions. However, this familiar idiom has the significant drawback that states are *not* abstract: they are known to be functions of type `unit → unit`. The above program uses explicit pre-computed arrays for the `waitAny` call, because an abstract API would have hit `let rec` restrictions that prevented these from being pre-

computed from a more abstract specification. Furthermore, state-counting instrumentation can only be added by altering client programs, rather than augmenting a library, a violation of abstraction. Ideally the existence of instrumentation and performance-related caches should not even be revealed.

5 Examples of Value Recursion: Picklers

The next example is drawn from Kennedy [13], who introduces a combinator library for specifying *picklers*, a compositional way of specifying objects that manage both the marshalling and unmarshalling of data structures. The programmer controls what is marshalled, the marshalling order, sharing in the marshalled graph and the shape of the underlying data format. Corresponding unmarshallers are built automatically, ensuring consistency. Marshallers can be thought of as objects with a pair of marshal/unmarshal methods, though an implementation may augment them with additional functionality. The aim is to build marshallers via combinators such as those in the following channel-oriented version of the API:

```
type Channel (* e.g. a file stream *)
type α Mrshl
val marshal: α Mrshl → α * Channel → ()
val unmarshal: α Mrshl → Channel → α
val pairMrshl: α Mrshl * β Mrshl → (α * β) Mrshl
val listMrshl: α Mrshl → (α list) Mrshl
val innerMrshl: (α → β) * (β → α) → α Mrshl → β Mrshl
val intMrshl: int Mrshl
val stringMrshl: string Mrshl
```

Marshallers are instances of `α Mrshl`. Combinators shown here are those for pairs (`pairMrshl`), lists (`listMrshl`) and internal data (`innerMrshl`). The type of marshalling objects is abstract, but could be implemented by an object or record type such as the following:

```
type α Mrshl = { marshal: α * Channel → ();
                 unmarshal: Channel → α }
```

For example if files are represented by some structured data then marshallers can be constructed quite easily:

```
type file = int * string
let fileMrshl = pairMrshl(intMrshl,stringMrshl)
let filesMrshl = listMrshl(fileMrshl)
```

Kennedy observes how specifying marshallers for recursive data structures runs into trouble with `let rec` restrictions in Standard ML. For example, consider the following recursive data type (we add some helper functions to make the subsequent code more concise):

```
type folder = { files: file list; subfldrs: folders }
and folders = folder list
let mkFldr (x,y) = { files=x; subfldrs=y}
let destFldr f = (f.files,f.subfldrs)
```

```
let fldrInnerMrshl(f,g) = innerMrshl (mkFldr,destFldr) (pairMrshl(f,g))
```

We now wish to create marshallers for both a single folder and a list of folders. One attempt is as follows:

```
let rec fldrMrshl = fldrInnerMrshl(filesMrshl,fldrsMrshl)
and fldrsMrshl = listMrshl(fldrMrshl)
```

However, this declaration is rejected because of ML's restrictions on value recursion. It would also be an invalid initialization graph since it has an immediate cycle.[9] Even if you reveal the implementation of marshallers (as we did for the abstract type of states in §4), you still can't use Standard ML's value recursion, which can only define functions, and not records containing functions. To quote Kennedy

> This problem is overcome in ML implementations of parser combinators [18] by exposing the concrete function type of parsers... We can't apply this trick because marshallers are *pairs* of functions.

In the context of initialization graphs a simple solution is possible. Firstly, we add the following function to the API:

```
val delayMrshl: (() → α Mrshl) → α Mrshl
let delayMrshl p =
   { marshal = (fun x → (p ()).marshal x);
     unmarshal = (fun y → (p ()).unmarshal y) }
```

This function takes a delayed computation that is only evaluated when an marshal/unmarshal operation is invoked – it can only be defined because marshallers only exhibit *delayed* (i.e. reactive) behaviour. This is exactly what lets us build a recursive graph of marshaller objects using an initialization graph. This can be used to break cycles amongst immediate dependencies:

```
let rec fldrMrshl = fldrInnerMrshl(pairMrshl(filesMrshl,fldrsMrshl))
and fldrsMrshl = listMrshl(delayMrshl(λ(). fldrMrshl))
```

Note how we have been able to define a mutually-recursive graph of interacting marshalling objects in a concise style.

6 Examples of Value Recursion: Abstract APIs for GUIs

GUIs provide an excellent source of examples of mutually referential graphs of objects where both immediate and delayed dependencies feature prominently. Typically the widget containment hierarchy of a collection of related GUI objects must be specified at the point of creation through the use of immediate dependencies. Furthermore imperative functionality to configure GUI components is often exercised during initialization. We have already shown simplistic examples of self references amongst simple GUI components. This story repeats itself on a larger scale in typical hand-programmed or tool-generated

[9] This is obvious since there is a dependency cycle and yet there are no delayed computations on the right-hand-side, so all dependencies are immediate.

GUI code, to the extent that the initialization code for GUI components can run to many pages, full of delayed and immediate self-references between a network of components.

For example the GUI components of a program called `ConcurrentLife` (see samples at [22]) involve a form, a menu, 7 menu items, a background worker thread and a bitmap to record the state of the display. All GUI objects are abstract values created and manipulated using the `Windows Forms` library of .NET. Dependencies arise as follows:

- Immediate dependencies arise from the widget containment hierarchy;
- Many delayed-dependency loops exist between the GUI components.
- All major GUI APIs are *single threaded*: worker threads may not directly manipulate GUI components, but must serialize their GUI update actions through the event loop of the GUI thread. [10] This leads to non-local delayed dependency loops: a form refers to a menu item whose action causes a thread to serialize computed results back via the form. The dependencies are not immediate because the thread object is created during initialization, but not started.

The author developed three versions of variations of this program, using the following techniques to mediate the recursion:

- explicit initialization holes (`option ref`);
- explicit use of laziness;
- initialization graphs.

Recursive references occurred at essentially every line of the recursive binding. The version coded with initialization graphs appears simpler and easier to maintain and modify. In practice all versions of the program were sound w.r.t. initialization, but in no case can this be checked by the static type system. However, the versions using explicit techniques were found to contain several race conditions w.r.t. multi-threading. The repetitive clutter associated with explicitly mediating value recursion achieved very little and obscured the real logic of the program.

6.1 "Create and Configure" APIs

Most GUI programming APIs require a combination of direct-specification and an additional style of initialization which we call *create-and-configure*. For example, the API from the introduction could in practice be structured as follows:

```
val createForm: string -> Form
val createMenu: string -> Menu
```

[10] In the context of the .NET Windows Forms library this is done by using the `Form.BeginInvoke` method provided on each `Form` object that acts as a container of a related group of GUI components.

```
val createMenuItem: string -> MenuItem
val toggle: MenuItem -> unit
val setMenus: Form * Menu list -> unit
val setMenuItems: Menu * MenuItem list -> unit
val setAction: MenuItem * (unit -> unit) -> unit
```

Here the API uses explicit mutation to affect the post-hoc configuration of a component. Uses of create-and-configure APIs suffer from a lack of locality: the configuration information that specifies an object is spread across the creation and configuration sections of code. The possible call-graphs also become harder to understand. [11]

Create-and-configure APIs can be used in conjunction with initialization graphs by adding the configuration actions to the graph. For example, example (A) from §1 could be written:

```
let rec f = createForm("Form")                    (a)
    and _ = setMenus(f, [ m ])                    (b)
    and m = createMenu("File")                    (c)
    and _ = setMenuItems(m, [ mi1; mi2 ])         (d)
    and mi1 = createMenuItem("Item1")             (e)
    and _ = setAction(mi1, λ(). toggle(mi2))      (f)
    and mi2 = createMenuItem("Item2")             (g)
    and _ = setAction(mi2, λ(). toggle(mi1))      (h)
```

In this case the bindings will be completed in order a,c,b,e,g,d,f,h. Clearly it is crucial that the programmer only use such bindings for initialization actions that are essentially commutative, that is, the programmer should ensure that the same result would be achieved if configuration actions are executed after all bindings have been established.

7 Initialization Graphs and `self` in Object Oriented Languages

For completeness, we record the connection between value recursion and recurring problems in the design and use of OO languages. OO languages use the identifier `self` (or `this`) for self referential access, and typically feature a potential for unsoundness arising from the use of the object during initialization. For example, calling a virtual method in the middle of a constructor can lead to many problems, and different languages even implement different semantics for the associated dispatch. Languages that restrict the use of `self` until all fields are known to have been initialized appear too inflexible, though more liberal systems have even caused a number of security bugs in the virtual machine verifiers [9].

We note that initialization graphs let you encode self references in methods

[11] OO APIs also allow configuration of components via method overriding. For the purposes of this paper this is simply a convenient way to directly specify functional parameter values during initialization.

without the need for a `self` keyword in the language at all. For example, consider the following encoding of an object as an ML record. [12]

```
type object = { getName: unit -> string;
                lengthOfName: unit -> int }
let mkObject name =
  let rec obj = { getName = λ(). name;
                  lengthOfName =  λ(). length(obj.getName()); }
  in obj
```

The inner recursive binding is an initialization graph and the self reference `obj.getName` will never result in a runtime error because the dependency is delayed, i.e. will only be exhibited once `lengthOfName` is called at a later point. The technical report [23] gives additional examples where `self` runs into limitations when defining mutually referential objects.

This means that initialization graphs admit a disciplined approach to introducing limited static checking for initialization soundness in the context of OO programming. For example, consider the following erroneous program:

```
let rec mkObject name =
   let rec obj =
     let len = length(obj.getName()) in
     { getName = λ(). name;
       lengthOfName =  λ(). len; } in
   obj
```

In OO parlance the method `getName` is being invoked during the construction logic for `obj`. The above error will be caught by the static checking described in §2.5, so initialization graphs give at least some protection against this kind of error (a typical OO language would not complain about the invocation of a virtual method during initialization). Furthermore initialization graphs are the exception rather than the norm – only a handful of such graphs would occur in a typical program, meaning the vast majority of a program would be free of the possibility of initialization failures. This is in stark contrast to most OO languages, where the pervasive use of recursive initialization references through `self` complicates many aspects of design, reasoning and analysis.

Top-level initialization in the presence of dynamic loading

Initialization graphs do actually occur in the semantics of the top-level static initialization of dynamically loaded components of OO languages. C# and Java support top-level initialization through class-initializers. A C# execution engine (e.g. the CLR [14]) generally executes class-initializers upon first access to a static field of a class. All static fields are initially set to null values. If mutual references exist between the statics of two classes then one class-initializer will complete first and null values can be observed, even if all static fields appear to be initialized by each static initializer. In a concurrent

[12] Encodings of object systems into ML hit limitations – for example the encoding used here does not support subtyping [1]. However that is orthogonal to the issues discussed in this paper.

setting mutual-exclusion is only applied at the granularity of individual class-initializers, and so threads executing mutually referential class-initializers can deadlock. The CLR breaks these deadlocks arbitrarily, and null-values can be observed that are not observable in a single-threaded situation. In practice mutual references between class initializers are avoided by programmers.

8 Related Work

Recursion is a topic that pervades theoretical and practical computer science, and the concept of initialization graphs has strong affinity with ideas presented in other settings. We trust that the mechanisms and examples studied in this paper will be of use to those pursuing more theoretical aspects of disciplined approaches to dynamic linking, recursion, fix points and effects. We also trust that it will provide added motivation for the development of type systems in this area.

Scheme's `let rec`

The accompanying report [23] describes the approach to value recursion in a number of other languages, including Java and Scheme. In Scheme the example from the introduction becomes:

```
(letrec ((mi1 (createMenuItem("Item1", λ(). toggle(mi2))))
         (mi2 (createMenuItem("Item2", λ(). toggle(mi1))))
         (m (createMenu("File", (mi1, mi2))))
         (f (createForm("Form", (m)))) ...)
```

Scheme executes with values initially set to `undef`, a form of initialization hole. The problem here is that the programmer must manually sort the declarations according to the graph of immediate dependencies, and no protection is given if this order is incorrectly specified. We have shown how immediate dependencies arise quite naturally, especially when there is a containment relation between the objects being defined, or in the case of combinator-generated objects such as the marshallers of §5, where delayed dependencies are the exception rather than the rule.

API, Data or Language?

One approach to value recursion is to assume that the problem lies with the API, rather than the programming language. Progress has been made recently on type systems where APIs can be annotated with dependency information, e.g. Dreyer's work [5,11]. This is problematic when APIs must be used that are not marked with full type information regarding recursive effects, which is the case for any language which permits the automatic import of COM, Corba, Java and/or .NET APIs [22,2]. Dreyer admits the likely need for a mechanism for unrestricted recursion (see §8).

Another approach is to specify objects using recursive data rather than side-effecting function applications. Recent versions of OCaml support directly recursive data without the use of null values or mutation: both con-

structed data and delayed values are permitted on the right of the recursive bindings [12]. This approach lacks simple abstraction properties — even simple functions that generate concrete data cannot be used as part of such a recursive binding. It also means wrapping all external APIs to make them entirely data-driven, which is not a serious option when making use of the enormous external APIs available in .NET and Java, and in any case the interpretation layer from data to constructed objects will itself face value-recursion problems.

Another approach to GUI APIs is to use explicit "create and configure" wiring for all event connections (see §6.1). However, this approach does not apply to other problematic APIs (§4-5). A final approach is to equip APIs with a set of fixed point operators for describing recursion: see [13] for an example. This approach has not yet been shown to scale, or at least not within the confines of the ML type system: mutual recursion can require an arbitrary number of different operators, even more so if multiple types of objects are involved.

Recursive Modules and Type Systems

Recursive initialization considerations arise in the context of proposals for recursive modules in ML-style languages. Dreyer's work gives an excellent overview [5], also [11,3,4]. Dreyer also defines a static system based on tracking dependencies through annotations on function types, including polymorphism over the sets of names that represent these dependencies. Dreyer's work provides a strong conceptual foundation for further attempts to statically exclude the possibility of runtime failures. However, type inference for the system appears very difficult and, like many systems of effect inference, gives rise to extremely complicated types.

In addition to Dreyer's static system, forms of unrestricted recursion are used by both Russo and Dreyer [20,5] and they both give semantics for their respective constructs (Dreyer's is based on laziness). Our aim in this paper has been to explore the ramifications of unrestricted recursion within ML's core language. Both Russo and Dreyer's unrestricted recursion constructs result in runtime errors if immediate dependencies are present. This is akin to an initialization graph with only one node, i.e. all self references must be delayed. The evaluation semantics for both these systems are similar to those presented in §2 in that the evaluation of values can result in errors.

Mixins

Leroy, Hirschowitz and Boudol's work on source languages and foundational calculii for *mixins* is highly relevant (e.g. [11,3]). Fundamentally we agree with their analysis that a static type system is essential to tame value recursion in most cases. However, the question we have addressed in this paper is whether such a mechanism should be augmented by unrestricted recursion, to ensure no barriers to expressiveness and abstraction are added to

the language, particularly when using external APIs.

For example, it is apparent that the program from §2.4 can be encoded without runtime checks into an appropriately extended version of the λ_B calculus [11]. This reinforces the result from that section: value recursion increases expressiveness when Pure Core ML is extended with abstract types. Furthermore, if their mixin modules are used carefully for all the constructs in that example then it appears Hirschowitz's source language can express that recursion structure, and Boudol's provides additional flexibility [3]. However, if normal ML modules are used at any point (i.e. dependency information is lost) then there is no recovery: the possibility of cyclic immediate dependencies would be reported, and programs such as the given one would again become undefinable. This would appear to place a heavy burden on the API designer. It seems likely that a combined static/dynamic system may be appropriate to ensure a flexibility between expressiveness and safety.

Initialization graphs as described in this paper are not immediately composable: if two graphs with dangling references are to be separately constructed (e.g. via a function that returns a tuple) and then combined then they must communicate their mutual-references to each other via delayed computations, rather than by simple value names, which would force execution during composition. Some variations on mixins naturally give rise to partial initialization graphs, and the compilation of mixins using lazy techniques may be appropriate in the context of platform-oriented languages.

A mixin-like theory for the dynamic linking of mutually-dependent compilation units has been developed by Flatt and Felleisen [8], where uninitialized references at `letrec` are used to help permit on-demand dynamic linking.

Laziness in strict languages

Wadler et al. describe techniques to add on-demand computations to strict languages [24], and explain how doing it in the wrong way can result in problems. Here is their recommended way of defining an infinite stream:

```
type 'a streamres = Nil | Cons of 'a * 'a stream
and 'a stream = 'a streamres lazy
(* map : ('a -> 'b) -> 'a stream -> 'b streamres *)
let rec map f l = force (match force l with
                         | Nil -> lazy Nil
                         | Cons(h,t) -> lazy (Cons(f h,map f t)))
```

However their approach only helps with value recursion when datatypes have been explicitly design with delays. For example, using their syntactic sugar a single value that represents an infinite stream of "3"s can be defined using "`let rec $threes = Cons(3, threes)`". However, adding and exposing delays in data type definitions breaks abstraction boundaries (the type of "stream" is not abstract), and furthermore makes this mechanism insufficient when interfacing to external library components where the representations of types can't be modified. However if we follow the style of §5 and assume an abstract stream type comes with functions

```
val cons : 'a -> 'a stream -> 'a stream
val delay : (unit -> 'a stream) -> 'a stream
// e.g. let delay s = lazy (force (s()))
```

then we can use an initialization graph:

```
   let rec threes =  cons 3 (delay (fun () -> threes))
or let rec threes =  delay (fun () -> cons 3 threes)
```

Evidently Okasaki-style syntactic sugar might help greatly if `$` were shorthand for insertion of `delay` nodes, e.g. admitting

```
let rec $threes =  cons 3 threes
```

even if the stream were abstract. This hints at the possibility of classifying types that may appear in value recursion cycles via a type class mechanism:

```
typeclass 't Delayable = { delay : (unit -> 't) -> 't }
```

Such a construct could be used to express the delayed nature of the dependencies of purely behavioural objects such as lazy computations, streams and automata, though its applicability to GUI programming is limited since GUI elements tend to be multi-modal, exhibiting a mixture of immediate and delayed functionality.

An additional dimension to adding laziness to strict langauges is found in constructs for concurrent programming such as *futures* and *promises*, e.g. as in the Alice programming language [17,19]. These can be used to encode an explicit version of initialziation graphs. Like `lazy` values and initialization graphs the use of these constructs can result in runtime failures when cyclic dependencies occur.

Monadic Approaches to Recursion

This paper is about strict (call-by-value) languages: value recursion is much less of a problem for languages such as Haskell. However this only applies if initialization does not have side-effects, controlled through monads. The question is then whether values produced by executing monadic operations can be mutually-dependent: in a Haskell interpretation of the kind of value recursion considered in this paper each initializing computation may have effects within a particular monad. Launchbury and Erkök have described an axiomatization of value recursion in certain monads, and this is implemented as an extension to Haskell [7]. Friedman and Sabry [10] have proposed an alternative operational view of value recursion which is applicable to a wider range of monads (e.g. the continuation monad) – this requires that initialization is an operation in a state monad. Moggi and Sabry have given a semantics for a monadic meta-language incorporating this construct [15]. Both approaches incorporate a notion of runtime initialization failure, and also limit the role of forward immediate references.

Declarative GUI Programming

As an aside we note that the notion of "declarative" GUIs can be taken in a rather different direction where abstract behaviours in terms of event streams are used to give declarative combinatorial descriptions of reactive systems, e.g. see Fran and FranTk [21,6].

9 Discussion and Future Work

It has been observed elsewhere that value recursion yields a tension between expressiveness, efficiency, simplicity and soundness [12]. In particular, a language that admits many self referential programs may also admit unsound programs whose execution may result in a runtime exception. Likewise, stricter languages may reject too many sound programs and/or require artificial coding techniques.

A primary aim of ML programming is to eliminate or minimize the use of explicit initialization holes, `null` values and/or mutation-based APIs. In this paper we have shown how ML's restrictions on recursion leads to practical problems in the presence of abstract APIs. We have argued that "platform oriented" languages such as those that connect to .NET and/or Java are particularly exposed to this problem, because of the practical difficulty of characterizing the effects that may be caused by calling object-construction and object-wiring routines.

As an alternative we have presented a semi-safe approach to value recursion called *initialization graphs*. Given the prevailing importance of the above platforms a mixed static/dynamic approach to value recursion appears necessary for any platform-oriented language that seeks to eliminate the use of other problematic constructs.

We acknowledge the limitations of initialization graphs and note that care must be taken to ensure that programmers are warned of the possible dangers and limitations of any unsafe mechanism to deal with value recursion. In future work we will consider whether it is possible to characterize a reasonable set of default assumptions about typical APIs from these platforms in terms of a statically-checked system (e.g. Boudol's [3]) or a linear calculus of locations and capabilities [16]. However, given the prevailing importance of these platforms a mixed static/dynamic approach to value recursion appears necessary for any platform-oriented language that seeks to eliminate or minimize the use of explicit initialization holes, `null` values, mutation-based APIs and/or `self` references. We have argued that solving initalization puzzles by these other techniques does nothing to restrict the scope of possible initialization failures in theory or practice. However care must be taken to ensure that programmers are warned of the possible dangers and limitations of any unsafe mechanism to deal with value recursion.

Section §7 has shown the connection between value recursion and `self` and indicates that the mechanism may also allow us to introduce a notion

of partial initialization soundness in OO languages – a notion that currently barely features in their design.

Acknowledgement

We thank Andrew Kennedy, Nick Benton, Simon Peyton-Jones, Byron Cook, Georges Gonthier, Gavin Bierman, James Margetson, Paul Govereau and Claudio Russo for helpful discussions related to this work. We also thank anonymous reviewers for their helpful remarks and lively discussion.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996.

[2] P. N. Benton, A. J. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[3] Gérard Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 14(3):263–315, April 2004.

[4] Gérard Boudol. Safe recursive boxes. Technical Report 5115, INRIA, February 2004.

[5] Derek Dreyer. A type system for well-founded recursion. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–305. ACM Press, 2004.

[6] Conal Elliott. Declarative event-oriented programming. In *Principles and Practice of Declarative Programming*, pages 56–67, 2000.

[7] Levent Erkök and John Launchbury. A recursive do for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 29–37. ACM Press, 2002.

[8] Matthew Flatt and Matthias Felleisen. Units: cool modules for hot languages. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 236–248. ACM Press, 1998.

[9] Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[10] Daniel P. Friedman and Amr Sabry. Recursion is a computational effect. Technical Report 459, Indiana University, December 2000.

[11] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, pages 6–20, 2002.

[12] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Principles and Practice of Declarative Programming*, pages 160–171. ACM Press, 2003.

[13] Andrew J. Kennedy. Functional Pearl: Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

[14] Microsoft Corporation. The .NET Common Language Runtime. `http://msdn.microsoft.com/net/`.

[15] Eugenio Moggi and Amr Sabry. An abstract monadic semantics for value recursion. In *2003 Workshop on Fixed Points in Computer Science*, April 2003.

[16] Greg Morrisett, Amal Ahmed, and Matthew Fluet. L^3: A linear language with locations. In *TLCA'04: Proceedings of the Seventh International Conference on Typed Lambda Calculi and Applications*, pages 293–307. Springer-Verlag, April 2005.

[17] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. In Bernhard Gramlich, editor, *5th International Workshop on Frontiers in Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 248–263. Springer, August 2005.

[18] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, July 1996.

[19] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. *Trends in Functional Programming, Volume 5*, 5, 2004.

[20] Claudio V. Russo. Recursive structures for Standard ML. In *International Conference on Functional Programming*, pages 50–61, 2001.

[21] Meurig Sage. FranTk - a declarative GUI language for Haskell. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 106–117. ACM Press, 2000.

[22] Don Syme. The F# programming language. `http://research.microsoft.com/projects/fsharp`.

[23] Don Syme. An alternative approach to initializing mutually referential objects. Technical Report 2005-31, Microsoft Research, March 2005.

[24] Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language without even being odd, September 1998.