An Alternative Approach to Initializing Mutually Referential Objects

Don Syme
dsyme@microsoft.com

March 10, 2005

Technical Report
MSR-TR-2005-31

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract

Mutual dependencies between objects arise frequently in programs, and programmers must typically resort to manually filling “initialization holes” to help construct the corresponding object graphs, i.e. null values and/or explicitly mutable locations. This report describes a “base-line” proposal for a generalized form of value recursion in an ML-like language called *initialization graphs*, where value recursion is given the simplistic semantics of a graph of lazy computations whose nodes are sequentially forced, with uses of recursive values checked for initialization-soundness at runtime. We then develop examples using this mechanism to show how problematic the issue of value recursion is for ML-like languages, and in particular how sophisticated reactive objects cannot be defined in the language without using initialization holes, and how this forces ML programmers to break abstraction boundaries. At the same time we show how OO languages rely extensively on null pointers during initialization. We propose that a general, semi-safe mechanism allows value recursion to be used in conjunction with existing sophisicated abstract APIs such GUI libraries, and allows freshly defined APIs to be both abstract and yet not require clients to use explicit initialization holes. We propose that the initialization mechanism permits more programs to be expressed in the mutation-free fragment of ML, though we do not formally prove this result.

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.

Ideally we would like to express such a program in a “safe” language, e.g. one that does not support either mutation or null pointers: there is nothing in this specification that would appear to require such constructs in a language.

In this report we first describe a “dead simple” mechanism for value recursion in an ML-style language (we use an OCaml-like syntax). This mechanism has problems, but it will let us write the above program as follows (we assume an API with functions `createForm`, `createMenu` and `createMenuItem` - see §1.1):

```
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))
```

The above definition is not permitted in OCaml, nor is its equivalent in other ML-family languages (we consider Scheme, Java and C# in §1.1). This is because ML statically enforces a very strong notion of "initialization soundness", in particular that recursive bindings won't fail or have side-effects at all, and only lambda expressions may appear on the right of a set of recursive bindings. To get around this the programmer must write something like the following:

```
let the = function | Some v → v | None → failwith "initialization 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 programmer has had to code an explicit "initialization hole" using the built-in type α `option = Some` α `| None` and then use mutation to fill that hole. A version of ML without mutation would not have permitted the above program to have been written at all. 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.

We call a recursive value binding like (A) above an *initialization graph.* Initialization graphs are a form of the *unrestricted recursion* proposed by Dreyer in the context of recursive modules for Standard ML [7], and also mentioned in passing by other authors. Informally, the semantics are to construct a delayed computation for each binding and then to force these computations both eagerly and sequentially: if a reference to a delayed computation is encountered then the computation is executed immediately. Evaluation order is precisely defined, but not necessarily left-to-right. Initialization failures occur if the result of a delayed computation is required before the computation is completed. Warnings can always be emitted when the success of the bindings cannot be statically guaranteed or when evaluation order may deviate from strict left-to-right evaluation.

Initialization graphs are a blunt and simple approach to value recursion, and while the exact mechanism is not used in existing languages, they are similar in spirit to `letrec` in Scheme, and the general observation that laziness can be used to encode value recursion is well known. However, semanticists seem to have assumed that unrestricted recursion is an unmitigated evil to be avoided at all costs, except perhaps in languages such as Scheme where "all bets are off anyway". In the long-run this is an important goal, and we encourage the reader to consult [4, 7, 23, 15] for some of the excellent papers that attempt to tame value recursion by static techniques.

Despite this, we argue that unrestricted value recursion through laziness is not as bad as has been previously thought, even in the drastic presence of arbitrary effects such threads: it is still better to be able to write programs in a natural style, to avoid using mutation and nulls, and to avoid breaking abstraction boundaries (e.g. see §5). We argue this holds especially in the context of programming against complex external abstract libraries such as GUI APIs (e.g. the Java Swing libraries, or the .NET Windows Forms libraries), where there is little or no hope of augmenting these enormous libraries with sufficient static information to allow static approaches to value recursion to be used. We point out in passing that more and more of modern programming consists of programming against such APIs, and that no type system yet exists that is known to be able to capture the initialization properties required for expressing client programs of these APIs.

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. Initialization graphs are not *semantically* more powerful than ML, since the same effect can be achieved through explicit coding, e.g. using explicit initialization holes as above. However we propose that they do add expressivity to the non-mutating subset of ML (though we do not formalize this result in this report). This is done only by placing some initialization-soundness properties into the category of runtime-checked rather than statically enforced conditions.

This report proposes that, with appropriate checks and warnings, unrestricted recursion in the form of initialization graphs forms a feasible basis for value recursion in a programming language. We first examine the above initialization puzzle in some other languages (§1.1) and discuss issues related to API design (§1.2). We give a detailed informal description of initialization graphs (§2), and a more formal treatment (§3). Turning to applications we discuss how the mechanism simplifies programs that use typical GUI APIs (§4). Two examples are presented to demonstrate how ML's limitations create problems for API design and usability and to show how initialization graphs help avoid these problems (§5-§6). Finally we discuss the relationship between initialization graphs and uses of `self` in OO languages (§7) and finish with related work and conclusions (§8-§9).

An implementation of initialization graphs is available as an extension to F# [26, 27], an ML-style language for the .NET platform. However, the mechanism could also be applied to other languages, especially to help avoid or control the presence of null-values in OO languages (null values are often used as initialization holes, as we will discuss). For example, the mechanism could be applied to OO languages with non-null types [10, 25]. Indeed, one theme of this report is that several problematic constructs in OO langauges such as null pointers and "`self` references during construction" appear to be less necessary once initialization graphs are available in a language. This is particularly relevant when "safe" languages must interoperate with APIs defined in OO languages, e.g. for languages that interoperate with the Java and .NET platforms, since it is undesirable to add the full range of OO constructs to such languages.

1.1 A Simple Example

We now continue with the example (A) above. Assume we are using the following call-by-value API for constructing graphs of GUI objects:[1][2]

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

Now consider how we would use this API from other languages. In Java or C# the programmer writes something akin to the following (we assume the language can access an appropriate translation of the above API, and that we adopt a standard way of writing callbacks of the form `λ(). code`.[3])

```
class MyForm {
   Form f;  Menu m;  MenuItem mi1, mi2;
   MyForm() {
     this.mi1 = createMenuItem("Item1", λ(). { toggle(this.mi2); });
     this.mi2 = createMenuItem("Item2", λ(). { toggle(this.mi1); });
     this.m = createMenu("File", new List<MenuItem> { this.mi1, this.mi2 });
     this.f = createForm("Form", new List<Menu> { this.m });
} }
```

Here the variables of a class form a kind of recursive scope accessed via references through `this` and initialized via the constructor. Note that the programmer assumes that the closures passed to `createMenuItem` will not be evaluated until some later point (otherwise `this.mi2` would result in a null pointer exception). Thus the use of `null` values as initialization holes permits members of `this` to be textually referred to before they are fully initialized.

From Scheme things are easier, and the program would appear as follows:

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

The only significant issue here is that the programmer must manually sort the declarations in dependency order, and little or no protection is given if this order

[1]In this report we use the term 'object' with its general meaning of a value encapsulating state, rather than as a value within an explicitly OO language.

[2]Throughout this report we use Caml-style syntax for APIs and ML programs, with the exception of using `()` to represent the `unit` (i.e. `void`) type and `λ().` to represent lambda expressions.

[3]The actual syntax for specifying closures might involve either a delegate/delegee (C#), an anonymous delegate (C#), the use of an inner class (Java) or the use of a new subclass with an event handler (C#, Java or many other languages) – the exact details do not make much difference here.

is incorrectly specified. This is because Scheme executes with values initially set to `undef`, again a form of initialization hole.

1.2 API, Data or Language?

One approach to mutually referential values is to assume that the problem lies with the API, rather than the programming language. For example, the API above relies on an unstated invariant that functions such as `createMenuItem` do not apply their closure arguments while the object graph is being constructed (i.e. event processing does not start until some later point), and an annotation to this effect would allow a compiler to declare the value bindings "safe". Progress has been made recently on type systems where APIs can be annotated with this information, e.g. Dreyer's work [7]. Type-theoretic solutions are no doubt crucial in the long term, but are 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 [26, 3]. Dreyer admits the likely need for a mechanism for unrestricted recursion (see §8).

Another approach is to describe systems using recursive data rather than programatic calls. Recent versions of OCaml support directly-recursive data without the use of null values or mutation: both constructed data and delayed values are permitted on the right of the recursive bindings [16]. This approach lacks abstraction properties — even simple functions that generate concrete data cannot be used as part of such a program. It also means wrapping all external APIs and making them entirely data-driven, which is extremely problematic.

A final approach is to equip APIs with a set of "fixed point" operators for describing recursion: see [18] for an example. If mutual recursion is involved then many different operators may be required, and API clients use the correct operator according to the number of items involved in their recursive cycle. However the approach appears to require a level of sophistication which may not be feasible to expect from programmers and it is difficult to see this scaling to the sophisticated APIs used for specifying GUIs and other reactive machines.

2 Towards Initialization Graphs in the Language

Consider the following systematic 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' ])
```

[4]In OCaml `lazy` and `force` generate and consume delayed computations of the type α `lazy`. These correspond directly to Scheme's `delay` and `force`. The former is syntactic sugar for the construction of a lazy value. The type of lazy computations can be defined in the OCaml language itself using an appropriate discriminated union and a reference cell.

```
    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*. Within a recursive scope a value is interpreted as an on-demand lazy computation (`f'` etc.), but outside that scope it is the result of the eager forcing of that computation (`f` etc.). We call such a set of eagerly-evaluated lazy computations an *initialization graph*.

To be more precise, the execution of a recursive binding constructs a graph with one node for each binding. Each node is in one of the following states:

1. A closure recording an initializing computation yet to be performed;
2. A marker to indicate that the initializing computation is in progress;
3. A value containing the results of a successful initializing computation.

These correspond to the `lazy` values in the motivating code above. Execution proceeds by evaluating the computations at each node, placing the node in state 2 before the computation begins. If a reference to a variable is encountered then the action depends upon the state of the node for that variable:

- If in state 1 then the corresponding initializing computation is executed eagerly, i.e. earlier than scheduled;
- If in state 2 then an initialization error is raised;
- If in state 3 then evaluate to the given result value.

Upon completion of an initializing computation a node is placed in state 3. Laziness is used because we cannot statically be sure when values will be first required. In the above example evaluating the binding for `f` requires the evaluation of `m` which requires the evaluation of `mi1` and `mi2`, and so the evaluation of the bindings will complete in order `mi1`, `mi2`, `m`, `f`. Thus we deliberately abandon a strict adherence to left-to-right evaluation ordering in order to embrace more general value recursion.

Before proceeding we first note that this approach has some somewhat 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 intitialization-time dependencies cause runtime errors. Runtime checks are needed for this condition.

An ideal initialization graph will never raise an initialization error, and furthermore will not depend on the declaration order of the elements. However these are properties the programmer must ensure, and in this report are not statically enforced.

2.1 Terminology: Immediate and Delayed Dependencies

Initialization graphs support a broad class of recursive bindings as long as no cycles occur amongst *immediate* dependencies between values. Let's assume we augment an ML-style language to permit arbitrary self references in recursive value definitions. This includes well-behaved definitions such as (A) from §1, and also nonsensical recursive definitions such as the following

```
let rec x1 = x1 + 1
let rec x2 = not x2
```

Also assume we execute these bindings left-to-right, and that when we encounter a reference to a recursively bound variable we somehow resolve the reference to a value, perhaps by additional computation. It is useful to make the following distinctions:

- When this execution evaluates a reference to a recursively bound variable we record an *immediate dependency*, i.e. if we evaluate a reference to v that syntactically occurs in a binding for u then an immediate dependency is recorded from u to v.
- After the bindings have been completed, a number of objects or closures may have escaped that still include references to these variables. The subsequent evaluation of these references generate *delayed dependencies*, i.e. if we evaluate a reference to v that syntactically occurs in a binding for u then a delayed dependency is recorded from u to v.

Immediate and delayed dependencies are dynamic notions: in general it is not possible to statically determine if a given syntactic occurence of a recursively bound variable results in immediate or delayed dependencies, or even both (it is undecidable which parts of the initialization bindings will execute at all). Delayed dependencies are irrelevant as far as initialization-soundness is concerned: they are purely part of the emergent behaviour of the object values being defined. This is not a new observation: every programmer who writes a callback for an event-oriented system knows it intuitively, and every Haskell programmer soon learns that *all* dependencies are delayed.

3 Initialization Graphs: Basic Semantics

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

v	$=$	id	Variable
e	$=$	v	Value/Node Reference
	$=$	$e\ e$	Function Application
	$=$	`let rec` b_1 `and` ... b_n `in` e	Recursive Binding
	$=$	`fun` v `->` e	Lambda Abstraction
	$=$	c	Constant (e.g. Integers)
	$=$	`print`(string)	A simple effectful construct (See discussion in text)
b	$=$	v `=` e	Binding

Figure 1: Syntax for λ_I

allowing arbitrary expressions on the right of recursive bindings. We incorporate a simple effectful action `print(s)` to ensure that programs have observable behaviour. 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 in a state of (possibly delayed) evaluations rather than maps to values. We have omitted rules propagating errors (see §3.1). The given calculus can be extended to include conditionals, non-recursive structured data, pattern matching, mutable state and I/O in completely standard ways – most examples in this report 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 [16], which is beyond the scope of this report.[6] We consider the more problematic issue of exceptions later.

The evaluation of a recursive binding initially assigns a new delayed computation for each variable, and then evaluates each variable, which ensures that the thunk for that variable now contains a computed value. Hence the execution of a recursive binding leaves no unresolved delayed computations, and thus the delayed computations do not "escape" their lexical scope. The evaluation of simple variables may result in further computation and/or errors.

We observe the following about this semantics:

- Expressions never evaluate to delayed initialization thunks. The arguments to functions are values, hence the language is call-by-value. Initialization thunks are present in the evaluation machinery but are not first-class.

- Locations in the initialization graph are never aliased, since they are not directly referred to by expressions. They are each referenced by at most a

[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.

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

$$\begin{array}{rcll}
\Gamma & = & id \to l & \text{Environment}\\
\sigma & = & l \to V & \text{Initialization Graph State}\\
V & = & v & \text{Evaluated Initialization Thunk}\\
 & | & (\Gamma, \lambda_0 e) & \text{Delayed Initialization Thunk}\\
 & | & \texttt{error} & \text{Initialization Error}\\
v & = & c & \text{Constant Value}\\
 & | & (\Gamma, \lambda x.e) & \text{Closure Value}\\
 & & & \\
 & & f \oplus (x \mapsto y) & \text{Function extension}
\end{array}$$

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

$$\frac{}{\Gamma, \sigma, o \vdash \texttt{print}(s) \leadsto 0, \sigma, (o + s)}
\qquad
\frac{}{\Gamma, \sigma \vdash c \leadsto c, \sigma}$$

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

$$\frac{\begin{array}{c}l_i \text{ fresh}\\ \Gamma' = \Gamma \oplus \overline{(x_i \mapsto l_i)}\\ \sigma_0 = \sigma \oplus \overline{(l_i \mapsto (\Gamma', \lambda_0 e_i))}\\ \Gamma', \sigma_{i-1} \vdash x_i \leadsto v_i, \sigma_i \quad (1 \leq i \leq n)\\ \Gamma', \sigma_n \vdash e \leadsto v, \sigma'\end{array}}{\Gamma, \sigma \vdash (\texttt{let rec } \overline{x_i = e_i} \texttt{ in } e) \leadsto v, \sigma'}$$

Figure 2: Semantic objects and Operational Semantics for λ_I

unique binding in the environment (or a corresponding binding in closure environments).

- Under the above semantics the state collects locations related to prior recursive bindings. Closure environments continue to refer to them, although they will map to values rather than delayed computations.
- 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. (The delayed computation is immediately evaluated to a value.)
- If the expressions on the right of a `let rec` are all λs then we have the traditional semantics for `let rec`s limited to recursive functions. (Execution immediately reduces initialization thunks to closure values).

3.1 Initialization Errors and Exceptions

λ_I does not permit initialization thunks in the `error` state to be dereferenced (computations encountering initialization errors simply can't be derived). The simplest semantics is to change `error` to be a value and propagate it as an exception in the standard way (say `throw InitializationFailure` – we omit the rules here). This may leave unexecuted initialization thunks in the computation state, and in a richer language these may escape, e.g.:

```
let myCell = ref (fun x -> x) in
let notDelayed f x = f x in
try let rec a = myCell := (fun x -> b); a + 1
        and b = print "b"; 3 + 4
    in failwith "we don't reach here"
with InitializationFailure -> !myCell 5
```

Here the binding for `a` stores a closure that includes a delayed dependency on `b`. The execution of `a + 1` then raises an initialization error. This is caught and the execution of the closure now forces the evaluation of `b`, which prints `b` and succeeds. The initializing-computation has thus been restarted, though we retain the property that successful initialization ensures an absence of later initialization errors related to that particular initialization graph.

An alternative semantics would be that if any of the bindings in an initialization graph generates either an exception or `error` then remaining uninitialized bindings should be recorded to be `error`. This would be captured by the following rule:

$$\frac{\begin{array}{c} l_i \text{ fresh} \\ \Gamma' = \Gamma \oplus \overline{(x_i \mapsto l_i)} \\ \sigma_0 = \sigma \oplus \overline{(l_i \mapsto (\Gamma', \lambda_0 e_i))} \\ \Gamma', \sigma_{i-1} \vdash x_i \rightsquigarrow v_i, \sigma_i \;\; (1 \leq i < k, v_i \text{ not } \texttt{error}) \\ \Gamma', \sigma_{k-1} \vdash x_k \rightsquigarrow \texttt{error}, \sigma_k \\ \sigma = \sigma_k \oplus \overline{l_i \mapsto \texttt{error}} \;\; (k \leq i \leq n) \end{array}}{\Gamma, \sigma \vdash (\texttt{let rec } \overline{x_i = e_i} \texttt{ in } e) \rightsquigarrow \texttt{error}, \sigma'}$$

Expressing this exception-catching behaviour in a transformation like that of §3.4 requires the use of a *try-in-unless* construct [2] where the "try" protects a set of declarations rather than an expression.

3.2 Reasoning Principles

The following theorem holds for λ_I.[7]

Theorem 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 a simple 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.

Beyond this result, the key semantic considerations are the effects that can be performed during initialziation. For example, can initialization mutate state, perform I/O, throw exceptions or start threads? We discuss the latter two of these in more detail below. But before we do this, we note the standard Scheme and ML response to this problem: the language defines a precise order of execution within a single thread. Beyond this effects are fundamentally the problem of the programmer and library designer.

3.3 Useful Static Warnings and Errors

The calculus from the previous section permits nonsensical definitions such as `let rec x = x + 1` where the evaluation of `x` on the right-hand-side of the bind-

[7]Here we assume that we are interested in observing the existence and state of initialization thunks themselves, in order to understand the actual execution of the underlying mechanism. Thus the theorem is in terms of the quantity and state of initialization thunks in the thunk heap: something that cannot normally be observed, except through a debugger. This reflects the expository role the semantics plays in this paper.

ing will cause an immediate exception. It is obviously desirable to statically approximate the set of immediate dependencies in order to rule out such programs. It is also sensible to statically detect if the use of value-recursion may result in bindings being executed on-demand, instead of strictly left-to-right, and to warn the programmer in this case: this may be exactly what is desired, but initialization graphs are sufficiently novel that they should be used deliberately, rather than accidentally. Furthermore, a programmer may wish to be warned of all the places where runtime checks may occur when evaluating self referential variable bindings. Finally, we do not want to give a warning for every `let rec`. Bindings made up entirely of recursive function definitions should give no errors or warnings. However functions can be mixed with values, e.g.:

```
let rec f x = (x+x, λ(). fst(y)*x)
    and y = f 3
```

evalutes `y` to `(6,λ().6*3)`. The above is a non-trivial initialization graph, and as such a warning should be emitted.

A simple analysis of the bindings is sufficient to detect most such conditions. For example the following analysis is used by F# (here considering only the constructs found in λ_I):

- For each binding x = e where e is not a λ:
  - For each occurrence in e of a recursively-defined value y not appearing under a λ add a *definite immediate dependency* from x to y to a graph.[8]
  - For occurrences of recursively-defined values y under a λ add a *possible immediate dependency* from x to y to the graph.
- Iteratively repeat the above analysis for each unanalyzed binding x = e where e is a λ and where x appears as the target of a possible or definite dependency.

If any possible or definite immediate dependencies exist then a warning is emitted that an initialization graph is being used. If a loop exists amongst definite immediate dependencies then a compile-time error is given. If any forward dependencies exist amongst possible or definite immediate dependencies then a warning is given that bindings will be executed on-demand rather than in a strictly left-to-right order (a forward dependency is one from x_i to x_j for a binding `let rec` x_1 = e_1 ... x_n = e_n where $i < j$).

The above analyses are imperfect: you can still write programs that cause cycles amongst immediate dependencies, e.g. in the following x has an immediate dependency on itself:

```
let notReallyDelayed f x = f x
let rec v = notReallyDelayed (fun x -> v + x) 3
```

[8]This actually over-approximates the set of immediate dependencies since the branches of conditionals are not necessarily executed – for simplicity the analysis assumes they are.

It is clear that far more extensive inference algorithms are possible for initialization conditions, and the above only serves to hint at the techniques that may be applied to eliminate spurious warnings. See also [4] for an in-depth treatment of inference issues related to one particular static type systems for value recursion.

3.4 Implementation Techniques

Fortunately, initialization graphs are very 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`. The transformation works as follows: every expression of the form

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

is transformed to

```
let $x_1, ..., x_n$ =
   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 variables are replaced by `force` operations. This converts the value recursion to a form where all expressions on a `let rec` are now delayed computations and is also extremely easy to implement in a compiler. Optimizations to this scheme are possible: for example initialization thunks are not required for bindings x = e where e is a λ or some other delayed computation. This ensures that the performance of recursive functions is not impaired.

3.5 Concurrency and Escaping Values

As a final semantic issue, we consider what happens in the presence of the "extreme" effects such as thread-style concurrency (which in principle also induce the problem of continuations [19]). Two problems arise:

- The techniques from 3.1 can be used to store initialization graph nodes in shared state, making them accessible to other threads before the computations have completed.
- The bindings of an initialization graph may start new threads.

In both cases race conditions can arise when reading/writing the changes of state within the lazy thunks used to implement the delayed computations, potentially resulting in multiple executions of the computations if no mutual exclusion is ensured, a problem that also affects escaping instances of `delay`/`lazy` in Scheme/OCaml. Three approaches are:

1. A language can exclude this possiblity by tightly controlling shared state and when threads can be started, e.g. through a type system to control effects;

2. The obligation can be placed on the programmer to ensure that initializing computations do not escape to shared state before the bindings have been completed and that threads are not started during initialization;[9]

3. The initialization section can be treated as a single critical-region, and any threads that attempt to access an unevaluated binding would block until the initialization of all bindings has been completed.

The prototype F# implementation follows (2). If (3) were used the costs of entering the critical region could be reduced by first checking if thunks have been evaluated, along with other techniques from Concurrent Haskell [17].

4 Initialization Graphs and GUI APIs (Continued)

Initialization graphs are used when value bindings give rise to immediate and/or delayed dependencies. GUIs provide an excellent source of examples where immediate dependencies feature prominently: typically the widget containment hierarchy must be specified at the point of creation of the GUI objects through the use of immediate dependencies. We have already shown simple examples of self references amongst simple GUI components. This story repeats itself on a larger scale in a typical hand-programmed or machine-generated GUI.

For example the GUI components of the `ConcurrentLife` sample from the F# distribution involve a form, a menu, 7 menu items, a background worker thread and a bitmap to record the state of the display. Immediate dependencies arise from the widget containment hierarchy and many delayed-dependency loops exist between the GUI components. The author's experience was that use of initialization graphs made a large difference when developing this program. Above all no explicit initialization holes were required, and there was no need to manually sort the declaration order of objects according to the DAG of immediate dependencies. As soon as initialization graphs were used the author was able to scale-up the sample substantially, concentrating on GUI design issues rather than fighting against the programming language.

Event Loops and Self references. GUIs are interesting for a further reason: event loops give rise to an additional source of delayed dependency loops. For

[9]We note that this obligation effectively already exists for every alternative technique to the problems described in this paper, i.e. initialization holes and/or null pointers, since all involve potentially shared state. In other words, with regard to threads things are at least no worse.

example, consider an application that runs a background computation on a worker thread where the results of this computation must be fed back to 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. (In the context of the .NET WindowsForms 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.) This leads to "long-distance" dependency loops: a form refers to a menu item whose action causes a thread to serialize computed results back via the form. Initialization graphs permit programs containing such loops to be programmed in a natural way.

"Create and Configure" APIs. Most GUI programming APIs support a combination of direct-specification and an additional style of intitialization called *create-and-configure*. For example, the API from §1.1 could in practice be structured as follows:

```
val createForm: string -> Form
val createMenu: string -> Menu
val createMenuItem: string -> MenuItem
val toggle: MenuItem -> unit
val setMenus: Form * Menu list -> unit
val setMenuItems: Menu * MenuItem list -> unit
val setAction: MenuItem * action -> 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.[10]

Create-and-configure APIs can be used in conjunction with initialization graphs by adding the configuration actions to the graph as bindings whose results are immediately discarded. These bindings can be placed alongside the creation actions for the objects or in a separate section of the recursive scope. For example, example (A) 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)
```

[10]OO APIs also allow configuration of components via method overriding. For the purposes of this paper overriding can be thought of as a convenient way to directly specify functional parameter values during initialization.

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 initialization graphs for initialization actions that are essentially commutative, i.e. the programmer should ensure that the same result would be achieved if configuration actions are executed after all bindings have been established.

5 Abstract APIs for Automata

We now show how initialization graphs can be used to describe the mutually referential states of automata without resorting to the use of explicit initialization holes. We are particularly interested in cases where the implementation of automata states is hidden.

Assume we wish to run a controlled computation of a game on a worker thread via an automaton that transitions between control states `paused`, `running` and `finished` in response to signals `stopSignal`, `stepSignal`, `runSignal`, `resetSignal` and `exitSignal`. We assume a type `Game`, supporting functions `resetGame, oneStep: Game → Game` and the value `initialGame: Game`.

The use of such an automaton on a worker thread is standard, however states and transitions are usually encoded as explicit calls to platform primitives (e.g. .NET's `WaitHandle.WaitAny` or Unix's `select`). However there are advantages to using combinators and abstract values (objects) to represent the control-states: e.g. the implementation of states can be uniformly augmented with additional tracing, caching and/or profiling functionality. So instead of coding the states directly we will assume we have to use the following abstract API:

```
type α State
type α Transition = Signal * α NextState
type α NextState = () → α State
val waitAll: Signal list * α NextState → α State
val waitOne: α Transition list → α State
val peekOne: α Transition list * α NextState → α State
val doThen: (α → α) * α NextState → α State
val finish: α State
val run : α State → (α → α)
```

An automaton in a `waitAll` state waits until all given signals have been set; in a `waitOne` state it performs a select amongst the given signals and commits to the selected transition; in a `peekOne` state it performs a `waitOne` with a zero-timeout, else transitions to a default state; and in state `finish` it does nothing further. The execution of an automata passes a value of type α from state to state: 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 to represent `NextState` values. This allows the API to support the dynamic generation of the objects that represent new states and makes the API well-suited for use with initialization graphs.

The transitions of the worker automaton can now be specified using the following initialization graph:

```
let rec initial = resetThenRun
and running = peekOne([ stopSignal,  (λ(). paused);
                        stepSignal,  (λ(). running);
                        runSignal,   (λ(). running);
                        resetSignal, (λ(). resetThenRun);
                        exitSignal,  (λ(). finished) ],
                      (λ(). stepThenRun))
and resetThenRun = doThen(resetGame, (λ(). running)
and stepThenRun = doThen(oneStep, (λ(). running)
and paused = waitOne [ stopSignal,  (λ(). paused);
                       stepSignal,  (λ(). stepThenPause);
                       runSignal,   (λ(). running);
                       resetSignal, (λ(). resetThenPause);
                       exitSignal,  (λ(). finished) ]
and stepThenPause = doThen(oneStep, (λ(). paused))
and resetThenPause = doThen(resetGame, (λ(). paused))
and finished = finish
let thread = newThread (run initial initialGame)
```

The above is compact declaration of a set of mutually dependent objects along with the specifications of how they each react to the different signals.[11]

For declarations of this kind the partial static checking described in §3.3 will be very effective at detecting loops amongst immediate dependencies. This is because all λ constructs in the above code represent truly delayed computations.

5.1 Automata in ML without initialization graphs

In traditional ML one approach is to program a state machine is to use a set of recursive functions, e.g. using the following modifications to the above API:

```
type α State = α -> α
type α NextState = α State
```

Again α is instantiated to `Game`, and the first part of the automata becomes:

```
let rec initial s = resetThenRun s
and running s = peekOne([ stopSignal, paused;
                          stepSignal, running;
                          runSignal, running;
                          resetSignal, resetThenRun;
                          exitSignal, finish ],
                        stepThenRun) s
and resetThenRun s = doThen (resetGame, running) s
```

[11]One exception is that the closure syntax for next-state functions is a little obscure. A keyword such as `perform` for closures of this kind would help significantly.

This avoids ML's value-recursion restrictions by defining functions rather than arbitrary values. This has the significant drawback that states are *not* abstract: they are known to be functions $\alpha \to \alpha$, and that only functions can be declared in the given initialization block. For example, it is desirable if `peekOne` states cache some intermediary data which is regularly handed to the operating system (`peekOne` calls happen frequently and it is sensible to avoid any allocation here). In principle function values can hide caches, but the value-recursion restriction means we cannot create these at the same time as specifying the functions. In ML the caller must allocate caches prior to the recursive binding and use these within the bodies of recursive functions, but this breaks abstaction boundaries: the caches should ideally be fully private within the implementation of automata states. Similarly, the automaton API could be augmented with a method that reports the number of times a state is entered. The mutable cells required to store this data cannot be allocated within a ML-style recursive binding without breaking abstraction boundaries. We return to this issue in §6.

5.2 Automata and Immediate Dependencies

Removing the binding `initial = resetThenRun` from the above example eliminates all immediate dependencies from the definition, and thus makes it amenable to definition via Scheme's `letrec` construct. However immediate dependencies have their uses even in this setting: it is useful if the specification continue to function even under pseudo-abstractions such as replacing

```
and resetThenRun = doThen(resetGame, (λ(). running))
and resetThenPause = doThen(resetGame, (λ(). paused))
```

by the innocuous looking

```
and resetThen state = doThen(resetGame, (λ(). state))
and resetThenRun = resetThen running
and resetThenPaused = resetThen paused
```

This has introduced two immediate dependencies (e.g. state `resetThenRun` now has an immediate dependency on `running`), but sufficient delays still exist to ensure all cycles are broken (e.g. all references to `resetThenRun` are delayed). It would seem advantageous if the introduction of harmless immediate dependencies did not immediately invalidate a definition. However, a better way to introduce the above combinator would be to maintain the `NextState` discipline as follows, to ensure that no additional immediate dependencies are introduced.

```
and resetThen nextState = doThen(resetGame, nextState)
and resetThenRun = resetThen (λ(). running)
and resetThenPaused = resetThen (λ(). paused)
```

6 Abstract Compositional Marshalling Objects (Picklers)

The example from §5 showed how restrictions on value recursion in Standard ML can force programmers to break abstraction boundaries in order to support compact recursive specifications. We now turn to a slightly more sophisticated instance of this problem. We draw the example from [18], where Kennedy introduces a functional-language combinator library for specifying *picklers*, a compositional way of specifying objects that manage both the marshalling and unmarshalling of data structures, an approach that can also be applied in an OO setting. The library lets the programmer build up marshallers for data structures while still controlling 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 value-recursion restrictions in strict functional langauges such as Standard ML. For example, consider the following recursive data type (We add some helper functions to make the following 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.[12] Even if you reveal the implementation of marshallers (as we did for the abstract type of states in §5), you still can't use Standard ML's value recursion, which can only define functions, and not records containing functions. The problem comes up often in the combinatorial approach to programming, especially when the objects generated by the combinators have non-trivial behaviour. To quote Kennedy

> This problem is overcome in ML implementations of parser combinators [22] by exposing the concrete function type of parsers, and then abstracting on arguments... 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. This declaration would normally be rejected in ML.

[12] 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.

7 Initialization Graphs and self in Object Oriented Languages

OO languages use the value self for self referential access. One of the recurring problems in these languages is the potential for unsoundnesses that arise 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. Good design dictates that no use of self be made until all fields are known to have been initialized to an appropriate value. The complexities of specifying initialization-soundness conditions for constructors in OO and OO-bytecode languages have even caused a number of security bugs in the virtual machine verifiers [13, 5].

An object's constructor forms a recursive initialization scope somewhat similar to an initialization graph with one node. Indeed, initialization graphs let you encode self references in methods 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.[13]

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

Here one method (lengthOfName) is defined in terms of another (getName). 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.

self is a form of recursion that runs into limitations when defining mutually referential objects. In this paper we have seen several examples of such objects built through initialization graphs and combinator patterns. At a more contrived level, consider the following pair of objects:

```
type object = { getName: () -> string; }
let mkLinkedObjects (name1,name2) =
  let toggle = ref false in
  let rec obj1 = { getName=λ(). toggle := not !toggle;
                               if !toggle then name1 else obj2.getName() }
      and obj2 = { getName=λ(). toggle := not !toggle;
                               if !toggle then name2 else obj1.getName() }
  in obj1,obj2
let myObject1,myObject2 = mkLinkedObjects("abc","def")
```

Here the behaviours of obj1 and obj2 are intertwined: each call to getName on one will effect the behaviour of the other. The textually corresponding C#

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

or Java program would declare two subclasses of a base class `object`, and the mutual references could not be encoded by using `self` alone: extra fields would be need to hold the cross references between `obj1` and `obj2`, and these would be initialized via a create-and-configure pattern.

There are other interesting parallels between constructors for OO languages initialization graphs. 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 §3.3, so initialization graphs give more protection against this kind of error than do typical OO languages.

Although initialization graphs and constructors/`self` bear striking similarities, initialization graphs are the exception rather than the norm – indeed it is envisaged that only a handful of such graphs will occur in a typical program. This means that the vast majority of a program will 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. We also note that Moby [11] and some other OO languages disallow access to `self` during object construction, at least until all fields have been known to be initialized into a good state.

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 and will provide added motivation for the development of type systems in this area.

It seems likely that initialization graphs would have been considered as the semantic machinery to underpin Scheme's `letrec` at some point. As currently defined Scheme's `letrec` can't represent initializations that involve immediate dependencies. We have shown how immediate dependencies arise quite naturally, especially when there is a containment relation between the objects being defined, as in the case of GUIs. They also arise in combinator-generated objects, e.g. as the marshallers defined in §6, where delayed dependencies are the exception rather than the rule. Scheme's `letrec` could be added directly to an ML-style language with a guaranteed left-to-right evaluation. Additionally, programmers would have to manually sort their declarations according to the DAG of immediate dependencies.

Recursive Modules and Type Systems Considerable work exists regarding type systems for controlling effects within recursion, e.g. by annotating function types with levels that indicate whether functions arguments are evaluated or not, e.g. [16]. Dreyer's work gives an excellent overview [7], also [15, 4].

Recursive initialization considerations arise in the context of proposals for recursive modules in ML-style languages. In [7] Dreyer defines a distinction that captures the essence of immediate v. delayed dependencies. To quote a follow up description by Dreyer[6]:

> I design the type system so that it ensures that when evaluating a recursive definition (like `let rec x = e`), `x` will not be *dereferenced* when evaluating `e`, although it may be *referenced.*

Referencing and dereferencing correspond to the delayed and immediate dependencies from §2.1. Dreyer's primary aim is a type system for a form of restricted recursion, i.e. to statically exclude the possibility of failures. This would of course be extremely useful for reducing the number of warnings given in relation to initialization graphs, as discussed in §1.2.

A form of unrestricted recursion is defined by both Russo and Dreyer [23, 7] 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 ramifcations 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 §3 in that the evaluation of mere values can result in errors.

Laziness in strict languages Wadler et al. describe different methods to add on-demand computations to strict languages [28], and explain how doing it in the "wrong" way can easily result in problems. For example here is the "right" way to define an infinite stream via laziness:

```
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 does not help with value recursion. For example, a single value that represents an infinite stream of the value "3" cannot be defined using `let rec threes = lazy(Cons(3, threes))` due to value-recursion restrictions.[14] If we follow the approach of §6 then a `delay` stream constructor is needed, along with either an explicit initialization hole or an initialization graph:

```
(* delay : (() -> 'a stream) -> 'a stream *)
let delay s = lazy (force (s()))
let rec threes =  lazy (Cons(3, delay (fun () -> threes)))
```

Top-level intialization in the presence of dynamic loading Somewhat related is the recurring problem of *top-level static initialization of dynamically loaded components.* C# and Java support top-level initialization through class-initializers. A C# execution engine (e.g. the CLR [20]) 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 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.[15]

This indicates that top-level initialization for dynamic loading is a fundamentally more complex problem than the localized mutual references dealt with by initialization graphs. A theory for the dynamic linking of mutually-dependent compilation units has been developed by Flatt and Felleisen [12], where uninitialized references at `letrec` are used to help permit on-demand dynamic linking.

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. Haskell's way to control effects is through the use of *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 [9]. Friedman and Sabry [14] 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.

[14]A recursive function can be used instead, e.g. `let rec repeat n = lazy (Cons(n, repeat n))`. This will generate a new stream object each time "3" is consumed. An interesting possibility would be to explore the automatic introduction of initialization holes for the purposes of optimization, though this will not be possible for variations on streams that store caches or other additional state, c.f. automata states in §5.

[15]In reality mutual references between class initializers are avoided by programmers.

Moggi and Sabry have given a semantics for a monadic meta-language incorporating this construct [21]. Both approaches incorporate a notion of runtime initialization failure, and also limit the role of forward immediate references. However, initialization graphs permit effects to be executed on-demand, and this appears distinct from the treatment of effects in these versions of monadic recursion. An axiomatization of the monads for which initialization graphs are appropriate would be of great interest.

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 [24, 8].

9 Discussion and Future Work

It has been observed elsewhere that value recursion yields a tension between expressiveness, efficiency, simplicity and soundness [16]. In particular, a language that admits many self referential programs may also admit unsound programs whose execution may result in a runtime exception. Likewise, languages may reject many sound programs and require artificial coding techniques in order to express programs.

This paper has presented an alternative approach to value recursion called *initialization graphs*. Most significantly the mechanism helps eliminate the use of explicit initialization holes, `null` values, mutation-based APIs and/or `self` references for crucial programming tasks. We have argued that "solving" initalization puzzles by these techniques is simply relying on fundamentally unsafe language constructs in an unregulated fashion, with consequences for the usability of the language. Instead we have proposed that a weaker notion of initialization soundness may be appropriate, even in the context of "safe" strict languages, and may produce better results overall. However care must be taken to ensure that programmers are warned of the possible dangers and limitations of the mechanism.

We have drawn examples from the setting of GUI and reactive programming and presented initialization graphs in the context of core ML. The examples from §5 and §6 examine two abstract combinatorial APIs for building graphs of related objects. They indicate how the value-recursion restrictions in ML-style languages lead to substantial problems for API design: either APIs must be non-abstract (revealing objects to be functions) or else client programs are forced to use explicit mutable intialization holes.

Section §7 has shown the connection between initialization graphs and `self` and indicates that the mechanism may also allow us to introduce a notion of initialization soundness in OO languages – a notion that currently barely features in major OO languages. This may be important because it is clearly difficult for an OO language to accept the full restrictions of an ML-like mechanism.

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. Future work will examine appropriate mechanisms to support functions that generate partial initialization graphs more conveniently.

Acknowledgements We would like to thank Andrew Kennedy, Nick Benton, Simon Peyton-Jones, Byron Cook, Georges Gonthier, Gavin Bierman and Claudio Russo for helpful discussions related to this work. We also think the reviewers for their helpful remarks and lively discussion.

References

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

[2] N. Benton and A. Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, 2001.

[3] 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.

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

[5] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java security: Web browsers and beyond. pages 241–269, 1998.

[6] D. Dreyer. Message to the Types email list, September 2004. Quoted at `http://lists.seas.upenn.edu/pipermail/types-list/2004/000352.html`.

[7] D. 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.

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

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

[10] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312. ACM Press, 2003.

[11] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 37–49. ACM Press, 1999.

[12] M. Flatt and M. 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.

[13] S. N. Freund and J. 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.

[14] D. P. Friedman and A. Sabry. Recursion is a computational effect. Technical Report 459, Indiana University, December 2000.

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

[16] T. Hirschowitz, X. 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.

[17] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308. ACM Press, 1996.

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

[19] S. Kumar, C. Bruggeman, and R. K. Dybvig. Threads yield continuations. *Lisp Symb. Comput.*, 10(3):223–236, 1998.

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

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

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

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

[24] M. 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.

[25] C. Smith. Java pointifications: Nullability constraints, June 2001. At `http://cdsmith.twu.net/professional/java/pontifications/nonnull.html`.

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

[27] D. Syme. ILX: Extending the .NET Common IL for functional language interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[28] P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language without even being odd, September 1998.