The Heart of Spritely: Distributed Objects and Capability Security

Christine Lemmer-Webber, Randy Farmer

Table of Contents

NOTE: This is an early draft. Has yet to be reviewed by the technical review board.

This paper is the second in a three-part series explaining Spritely's thinking and design. The first paper, Spritely: New Foundations for Networked Communities, explains the problems which face contemporary social network design. This paper explains the necessary technical toolbox available

for programmers to build out Spritely's vision. The third paper in the series, Spritely for Secure Applications and Communities, ties the first two papers together by showing what how the architecture for user-facing software which fulfills the vision of the first paper can be built on top of ideas from this paper.

While Spritely's core tooling is generally useful and may be of interest to a wide variety of programming backgrounds, it is designed with a purpose: to give us the firm footing to be able to achieve the ambitious journey of fulfilling the full user-facing Spritely vision. If your goal is to understand Spritely's full vision, it is our recommendation that you read each paper in order.

1. Introduction

Building peer-to-peer applications on contemporary programming architecture is a complicated endeavor which requires careful planning. Building the kind of fully-decentralized social networking design that Spritely aspires for would be too hard on systems that assume traditional client-server architecture and authority models. If each of our needs runs contrary to the grain of expected paradigms, we will have a hard time achieving our goals. Still, we must provide a development model which is comfortable in ways which match programmer intuitions. Spritely's core layers of abstractions achieve each of these seemingly contradictory requirements by drawing together decades of research from object capability security and programming language design communities.

Spritely's core layers of abstractions make building secure peer-to-peer applications to be as natural as any other programming. Spritely provides an integrated system for distributed asynchronous programming, transactional error handling, time-travel debugging, and safe serialization. All this under a security model resembling ordinary reference passing, reducing most considerations to a simple slogan: "If you don't have it, you can't use it."

2. Capability security as ordinary programming

> The Principle of Least Authority (POLA) says that code should be granted only the authority it needs to perform its task and no more. Code has a lot of power. Code can read your files, delete your files, send your files (and all of the information within them) to someone else, record your keystrokes, use your laptop camera, steal your identity, hold your computer for ransom, steal your cryptocurrency, drain your bank account, and more. But most of the code that we write doesn't need to do any of those things – so why do we give it the authority to do so?
>
> POLA is ultimately about eliminating both ambient and excess authority. It's not a motto that is meant to be inspirational; POLA can actually be achieved. But how?
>
> – Kate Sills, POLA Would Have Prevented the Event-Stream Incident

The power of this model is best understood by contrast to the prevailing authority model, common to Unix and nearly everything which has followed since. If Alisha is logged in to her computer and wants to play Solitaire, she can run it like so:

```
# Applications run as Alisha!
# Can do anything Alisha can do!
> solitaire
```

Using an Access Control List permission system, Solitaire, the most innocuous-seeming of programs, can wreak the maximum amount of havoc possible to Alisha's computing life. Solitaire

could snoop through Alisha's love letters, upload her banking information to a shady website, and delete or cryptolock her files (possibly demanding a tidy sum on behalf of some shady group somewhere to release access).

What makes seemingly-innocent Solitaire so dangerous is the *ambient authority* of Access Control List operating systems. In such a computing environment, when Alisha types "solitaire" in a terminal window or double clicks on its icon, her computer runs Solitaire *as Alisha*. Solitaire can do everything Alisha can do, including many dangerous things Alisha would not like.

The contrast with an object capability environment is strong. Following the principle of least authority, programs, objects, and procedures are defined in an environment with no dangerous authority. In an object capability computing environment, Solitaire would only be able to run with the authority it has been handed.

Let's think of `solitaire` as being a procedure within an object capability secure language. (To make it obvious that these ideas can extend to a variety of language environments,[1] we will use a syntax which resembles something like Javascript or Python.) Solitaire, being run, cannot do anything particularly dangerous… but it can't do anything particularly useful either.

```
// Runs in an environment with no special authority...
// not even the ability to display to the screen!
> solitaire()
```

As-is, all solitaire can do is return a value… but Solitaire as a game requires interactivity: it should display to the screen, and it should be able to read input through the keyboard and mouse.

Let's introduce a capability which has been granted more power by the underlying system, `makeWinCanvas(windowTitle)`. Let's say that `solitaire` can take a first argument which takes a window + canvas representing object which is able to read keyboard and mouse input, but only while the window is active. We will be able to use the former to produce a value to pass to the latter, with exactly that authority and no more:

```
// Constructs a new window
> solitaireWin = makeWinCanvas("Safe Solitaire")
// Pass it to solitaire
> solitaire(solitaireWin)
```

If we want to allow Solitaire to be able to access a high score file, we could imagine that the `solitaire` procedure could accept a third procedure for exactly that purpose:

```
> scoreFile = openFile("~/.solitaire-hs.txt", "rw")
> solitaire(solitaireWin, scoreFile)
```

Consider the power of this: solitaire now has access to display to the `solitaireWin` window, it can read from the keyboard and mouse when the window is active, it can only write to the specific file we have given access to, but it cannot do anything else dangerous. It cannot access the network. It cannot read or write files from the filesystem arbitrarily (it only access the high score file it was given). It cannot act as a keylogger (it can only read keyboard and mouse events while the window is being actively used by the user).

We have built our object capability security model on completely ordinary reference passing, familiar to the kind of programming developers do every day. What can and cannot be done is clear: if you don't have it, you can't use it.

1 The requirements for a programming language to be considered object capability safe are reasonably minimal (no ambient authority, no global mutable state, lexical scoping with reference passing being the primary mechanism for capability transfer, and importing a library should not provide access to interesting authority). See A Security Kernel Based on the Lambda Calculus for more information.

3. Spritely Goblins: Distributed, transactional object programming

At the heart of Spritely is Goblins, its *distributed object programming* environment.[2] Goblins provides an intuitive security model, automatic local transactions for locally synchronous operations, and an easy to use and efficient asynchronous programming for *encapsulated objects* which can live anywhere on the network. Its networking model abstracts away these details so the programmer can focus on object programming rather than protocol architecture. Goblins also integrates powerful distributed debugging tools, and a process persistence and upgrade model which respects its security fundamentals.[3]

Within Goblins, when we say *distributed object*, we are referring to a model where many independent objects communicate with other objects on many different machines. In other words, when we refer to *distributed object programming*, we mean "a distributed network of independent objects".[4] Objects are built out of *encapsulated behavior*: an object is *encapsulated* in the sense that its inner workings are opaque to other objects, and (contrary to the focus of many systems today) objects are *behavior-oriented* rather than *data-oriented*. Goblins enables intentional collaboration between objects even though the network is assumed hostile as a whole.

Goblins utilizes techniques common to functional programming environments which enable cheap *transactionality* (and by extension, *time travel*). The otherwise tedious plumbing associated with these kinds of techniques is abstracted implicitly so the developer can focus on object behavior and interactions.

3.1. A taste of Goblins

The following examples will illustrate Goblins using its implementation in Guile (which is a type of Scheme, which is itself a type of Lisp).[5] While the ideas here could be ported across many kinds of

2 In recent years there has been enormous pushback against the term "object", stemming mostly from functional programming spaces and PTSD developed from navigating complicated Java-esque class hierarchies. However, the term "object" means many different things; Jonathan Rees identified nine possible properties associated with programming uses of the word "object". For Goblins, *objects* most importantly means addressable entities with encapsulated behavior. Goblins supports *distributed objects* in that it does not particularly matter where an object lives for asynchronous message passing; more on this and its relationship with *actors* later.

3 Goblins draws inspiration largely from two sources. The first is Scheme (on which its current implementations are built), and particularly the "W7" Scheme variant found in A Security Kernel Based on the Lambda Calculus, and the E programming language. (Both of these have rich histories of their own, particularly E's predecessor Joule, so of course Goblins inherits those too.) W7's primary contribution is the observation that a purely lexically scoped language, with Scheme in particular, is already an excellent candidate for an object capability security environment. E's primary contribution is the *distributed object* approach that Goblins largely adopts, including the first version of the *CapTP* protocol used by Goblins as the object communication layer abstraction of OCapN. Goblins can thus be seen as a combination of Scheme/W7 and E, with Goblins' primary innovative contribution being its transactional design.

4 This is not to be confused with "the objects themselves are distributed across different machines", which we do address as the *Unum Pattern* in the Distributed behavior and why we need it section.

Similarly we do not mean distributed *convergent machines* (such as *blockchains* or *quorums*), where a single abstract *machine*, with all of its contained objects, can be deterministically replicated by multiple independent machines on the network. While such designs can be composable with Spritely Goblins (or even easily built on top of its transactional architecture), they are not the essential infrastructure to achieve Spritely's goals. Further discussion of *convergent machines* is reserved for a future paper.

5 At present, Goblins has two implementations, one on Racket (the initial implementation), and one on Guile (which is newer). While both will be maintained and interoperable with each other in terms of distributed communication, the Guile implementation is becoming the "main" implementation on top of which the rest of Spritely is being built. Goblins' ideas are fairly general though and Goblins is implemented simply as a library on top of a host programming language, and Goblins' key ideas could be ported to any language with sensible lexical scoping (but it

programming languages, Scheme's minimalism and flexibility allow for cleanly expressing the core ideas of Goblins.

Prior knowledge of Scheme is not necessary, but some familiarity with programming in general is expected. We have chosen an unusual representation of Lisp syntax which is whitespace-based instead of parenthetical, named Wisp.[6] Experience has shown that while parenthetical representations of Lisp tend to feel alien to newcomers with prior programming experience, Wisp tends to look fairly pleasantly like pseudocode. Subexpressions are simply indented greater than their parents, or (more rarely) can be inlined by using a colon. Parenthetical notation is also supported but is used sparingly. Lines which start with a period means that the line does not start a subexpression but is part of its parent. (The lines beginning with periods are typically the ugliest part; the reader is encouraged to ignore these while reading and focus on the shape of the code.)

Code examples that have lines preceding with `REPL>` are meant to demonstrate examples of interactive use. Lines which follow and are preceded by underscores represent continued entries for the same expression.

We have aimed to have these examples be as simple as possible to understand just by reading them. Some readers will wish to try or play with the examples for themselves; we have provided two appendices if you are this kind of reader:

- Appendix: Following the code examples shows how to get the examples up and running (including how to get past some hand-waving we have done for the `REPL>` examples).
- Appendix: A small scheme and wisp primer gives more detailed explanations of the host language itself.

Again, neither of these is required to follow along.

Now you're ready to go. Read on!

3.1.1. A simple greeter

Here we will give an extremely brief taste of what programming in Goblins is like. The following code is adapted from the Guile version of Goblins.[6]

First, let us implement a friend who will greet us:

```
;; define with next argument wrapped in parentheses
;; defines a named function
define (^greeter _bcom our-name)   ; constructor (outer procedure)
  lambda (your-name)               ; behavior    (inner procedure)
    format #f "Hello ~a, my name is ~a!"     ; returned implicitly
           . your-name our-name
```

The outer procedure, defined by `define`, is named `^greeter`, which is its constructor.[7] The inner procedure, defined by the `lambda` (an "anonymous function"), is its behavior procedure, which implicitly returns a formatted string. Both of these are most easily understood by usage, so let's try instantiating one:

```
;; define with next argument *not* in parentheses
;; defines an ordinary variable
```

might not look as nice or be as pleasant to use or elegant).

6 Wisp's rules are defined in SRFI 119. Wisp's key feature is that it has all the same structural properties as a parenthetical representation and can be translated back and forth between the the parenthetical form and the whitespace-based form bidirectionally with few key rules.

7 The ^ character is conventionally prefixed on Goblins constructors and is called a *hard hat*, referring to the kind used by construction workers.

```
define gary
  spawn ^greeter "Gary"
```

As we can see, `spawn`'s first argument is the constructor for the Goblins object which will be spawned. For now, we'll ignore the `_bcom`, which is not used in this first example (an underscore prefix is the conventional way to note an unused variable; we'll see some examples where this *is* used soon). The rest of the arguments to `spawn` are passed in as the rest of the arguments to the constructor. So in our case, `"Gary"` is passed as the value of `our-name`.

The constructor returns the procedure representing its current behavior. In this case, that behavior is a simple anonymous `lambda`. We can now invoke our `gary` friend using the synchronous call-return `$` operator:[8]

```
REPL> $ gary "Alice"
;; => "Hello Alice, my name is Gary!"
```

As we can see, `"Alice"` is passed as the value for `your-name` to the inner `lambda` behavior-procedure. Since `our-name` was already bound through the outer constructor procedure, the inner behavior is able to pass both of these names to `format` to give a friendly greeting.

3.1.2. State as updating behavior

Let's introduce a simple cell which stores a value. This cell will have two methods: `'get` retrieves the current value, and `'set` replaces the current value with a new value.

```
define (^cell bcom val)
  methods             ; syntax for first-argument-symbol-based dispatch
    (get)             ; takes no arguments
      . val           ; returns current value
    (set new-val)     ; takes one argument, new-val
      bcom : ^cell bcom new-val  ; become a cell with the new value
```

Let's try it. Cells hold values, and so do treasure chests, so let's make a treasure chest flavored cell. Taking things out and putting them back in is easy.

```
REPL> define chest
_____   spawn ^cell "sword"
REPL> $ chest 'get
;; => "sword"
REPL> $ chest 'set "gold"
REPL> $ chest 'get
;; => "gold"
```

Now we can see what `bcom` is: a capability specific to this object instance which allows it to change its behavior! (For this reason, `bcom` is pronounced "become"!)

`methods` was also new to this version. It turns out that `methods` is simply syntax sugar, a macro which returns a procedure which supports symbol dispatch on its first argument. There is nothing special about `methods`: you could easily write your own version or use it outside of Goblins objects to build general symbol-based-method-dispatch.

8 Any code line preceded by `REPL>` represents the prompt for interactively entered code at a developer's REPL (Read Eval Print Loop). Lines following represent expected returned values or behavior, and those prefixed with `=>` represent an expected return value.

3.1.3. Objects which contain objects

Objects can also contain and define other object references, including in their outer constructor procedure.

Here is the definition of a "counting greeter" we call `^cgreeter`:

```
define (^cgreeter _bcom our-name)
  define times-called   ; keeps track of how many times 'greet called
    spawn ^cell 0      ; starts count at 0
  methods
    (get-times-called)
      $ times-called 'get
    (greet your-name)
      define current-times-called
        $ times-called 'get
      ;; increase the number of times called
      $ times-called 'set
        + 1 current-times-called
      format #f "[~a] Hello ~a, my name is ~a!"
             $ times-called 'get
             . your-name our-name
```

As we can see near the top, `times-called` is instantiated as an `^cell` like the one we defined earlier. The current value of this cell is returned by `get-times-called` and is updated every time the `greet` method is called:

```
REPL> define julius
_____   spawn ^cgreeter "Julius"
REPL> $ julius 'get-times-called
;; => 0
REPL> $ julius 'greet "Gaius"
;; => "[1] Hello Gaius, my name is Julius!"
REPL> $ julius 'greet "Brutus"
;; => "[2] Hello Brutus, my name is Julius!"
REPL> $ julius 'get-times-called
;; => 2
```

3.1.4. Asynchronous message passing

We have shown that the behavior of objects may be invoked synchronously with `$`. However, this only works if two objects are both defined on the same machine on the network and the same event loop within that machine. Since Goblins is designed to allow for object invocation across a distributed network, what can we do?

This is where `<-` comes in. In contrast to `$`, `<-` can be used against objects which live anywhere, even on remote machines. However, unlike invocation with `$`, we do not get back an immediate result, we get a promise:

```
REPL> <- julius 'greet "Lear"
;; => #<promise>
```

This promise must be listened to. The procedure to listen to promises in Goblins is called `on`:

```
REPL> on (<- call-me 'greet)
_____   lambda (got-back)
_____     format #t "Heard back: ~a\n"
_____            . got-back
;; Prints out, at some point in the future:
```

```
;;   "Heard back: [3] Hello Lear, my name is Julius!"
```

Not all communication goes as planned, especially in a distributed system. `on` also supports the keyword arguments of `#:catch` and `#:finally`, which both accept a procedure defining handling errors in the former case and code which will run regardless of successful resolution or failure in the latter case:

```
REPL> define (^broken-bob _bcom)
_____   lambda ()
_____     error "Yikes, I broke!"
REPL> define broken-bob
_____   spawn ^broken-bob
REPL> on (<- broken-bob)
_____    lambda (what-did-bob-say)
_____      format #t "Bob says: ~a\n" what-did-bob-say
_____    . #:catch
_____    lambda (err)
_____      format #t "Got an error: ~a\n" err
_____    . #:finally
_____    lambda ()
_____      display "Whew, it's over!\n"
```

3.1.5. Transactions make errors survivable

Mistakes happen, and when they do, we'd like damage to be minimal. But with many moving parts, accomplishing this can be difficult.

However, Goblins makes our life easier. To see how, let's intentionally insert a couple of print debugging lines (with `pk`, which is pronounced and means "peek") and then an error:

```
define (^borked-cgreeter _bcom our-name)
  define times-called
    spawn ^cell 0
  methods
    (get-times-called)
      $ times-called 'get
    (greet your-name)
      pk 'before-incr : $ times-called 'get
      ;; increase the number of times called
      $ times-called 'set
        + 1 ($ times-called 'get)
      pk 'after-incr : $ times-called 'get
      error "Yikes"
      format #f "[~a] Hello ~a, my name is ~a!"
             $ times-called 'get
             . your-name our-name
```

Now let's spawn this friend and invoke it:

```
REPL> define horatio
_____   spawn ^borked-cgreeter "Horatio"
REPL> $ horatio 'get-times-called
;; => 0
REPL> $ horatio 'greet "Hamlet"
;; pk debug: (before-incr 0)
;; pk debug: (after-incr 1)
;; ice-9/boot-9.scm:1685:16: In procedure raise-exception:
;;   Yikes
;; Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
```

Whoops! Looks like something went wrong! We can see from the `pk` debugging that the `times-called` cell should be incremented to 1. And yet…

```
REPL> $ horatio 'get-times-called
;; => 0
```

We will cover this in greater detail later, but the core idea here is that synchronous operations run with `$` are all done together as one transaction. If an unhandled error occurs, any state changes resulting from synchronous operations within that transaction will simply not be committed. This is useful, because it means most otherwise difficult cleanup steps are handled automatically.

This also sits at the foundation of Spritely Goblins' time travel debugging features. All of this will be discussed in greater detail in sections later in this document: The vat model of computation, Turns are cheap transactions, and Time-travel distributed debugging.

3.1.6. Promise pipelining

> "Machines grow faster and memories grow larger. But the speed of light is constant and New York is not getting any closer to Tokyo."
>
> — Mark S. Miller, Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control

Promise pipelining provides two different features at once:

- A convenient developer interface for describing a series of asynchronous actions, allowing for invoking the objects which promises will point to before they are even resolved (sometimes before the objects even exist!)
- A network abstraction that eliminates many round trips

Consider the following car factory, which makes cars carrying the company name of the factory:

```
;; Create a "car factory", which makes cars branded with
;; company-name.
define (^car-factory bcom company-name)
  ;; The constructor for cars we will create.
  define (^car bcom model color)
    methods                       ; methods for the ^car
      (drive)                     ; drive the car
        format "*Vroom vroom!*  You drive your ~a ~a ~a!"
               . color company-name model
  ;; methods for the ^car-factory instance
  methods                         ; methods for the ^car-factory
    (make-car model color)        ; create a car
      spawn ^car model color
```

Here is an instance of this car factory, which we will call `fork-motors`:

```
;; Interaction on machine A
REPL> define fork-motors
_____   spawn ^car-factory "Fork"
```

Since asynchronous message passing with `<-` works across machines, it does not matter whether interacitons with =fork-motors are local or via objects communicating over the network. We will treat `fork-motors` as living on machine A, and so the following interactions will happen with invocations originating from machine B.

Let's send a message to `fork-motors` invoking the `'make-car` method, receiving back a promise for the car which will be made, which we shall name `car-vow` (`-vow` being the conventional suffix given for promises in Goblins):

```
;; Interaction on machine B, communicating with fork-motors on A
REPL> define car-vow
_____   <- fork-motors 'make-car "Explorist" "blue"
```

So we have a *promise* to a future car reference, but not the reference itself. We would like to drive the car as soon as it rolls off the lot of the factory, which of course involves sending a message to the car.

Without promise pipelining, making use of the tools we have already shown (and following the pattern most other distributed programming systems use), we would end up with something like:

```
;; Interaction on machine B, communicating with A
REPL> on car-vow                    ; B->A: first resolve the car-vow
_____   lambda (our-car)            ; A->B: car-vow resolved as our-car
_____     on (<- our-car 'drive)    ; B->A: now we can message our-car
_____       lambda (val)            ; A->B: result of that message
_____         format #f "Heard: ~a\n" val
```

With promise pipelining, we can simply message the promise of the car directly. The first benefit can be observed from code compactness, in that we do not need to do an `on` of `car-vow` to later message `our-car`, we can simply message `car-vow` directly:

```
;; Interaction on machine B, communicating with A
REPL> on (<- car-vow 'drive)     ; B->A: send message to future car
_____   lambda (val)             ; A->B: result of that message
_____     format #f "Heard: ~a\n" val
```

While clearly a considerable programming convenience, the other advantage of promise pipelining is a reduction of round-trips, whether between our event-loop *vats* or across machines on the network.

This can be understood by looking at the comments to the right of the two above code interactions. The message flow in the first case looks like:

```
B => A => B => A => B
```

The message flow in the second case looks like:

```
B => A => B
```

In other words, machine B can say to machine A: "Make me a car, and as soon as that car is ready, I want to drive it!"

With this in mind, the promise behind Mark Miller's quote at the beginning of this section is clear.[9] If two objects are on opposite ends of the planet, round trips are unavoidably expensive. Promise pipelining both allows us to make plans as programmers and allows for Goblins to optimize carrying out those steps as bulk operations over the network.

3.1.7. Errors propagate through pipelines

TODO: Write this section. Less high of a priority as the others, though.

9 Like so many examples in this document, the designs of promise pipelining and the explanation of its value come from the E programming language, the many contributors to its design, and Mark S. Miller's extraordinary work documenting that work and its history. If you find this section interesting, both the the Promise Pipelining page from erights.org and sections 2.5 and 16.2 of Mark Miller's dissertation.

```
define (^lessgood-car-factory bcom company-name)
  define (^car bcom model color)
    lambda ()
      format #f "*Vroom vroom!*  You drive your ~a ~a ~a!"
             . color company-name model
  define (make-car model color)
    error "Your car exploded on the factory floor!  Ooops!"
    spawn ^car model color
  make-car

define forked-motors
  a-vat 'spawn ^lessgood-car-factory "Forked"

b-vat 'run
  lambda ()
    define car-vow
      <- forked-motors "Exploder" "red"
    define drive-noise-vow
      <- car-vow
    on drive-noise-vow
       lambda (val)
         format #f "Heard: ~a\n" val
       . #:catch
       lambda (err)
         format #t "Caught: ~a\n" err
```

3.2. Security as relationships between objects

> Cooperation between independent agents depends upon establishing a degree of security. Each of the cooperating agents needs assurance that the cooperation will not endanger resources of value to that agent. In a computer system, a computational mechanism can assure safe cooperation among the system's users by mediating resource access according to desired security policy. Such a mechanism, which is called a security kernel, lies at the heart of many operating systems and programming environments.
>
> – Jonathan A. Rees, A Security Kernel Based on the Lambda Calculus

In Capability security as ordinary programming we demonstrated how a programming language which uses lexical scoping and is strict about removing ambient authority is already likely an excellent foundation for a capability secure architecture. In A taste of Goblins we saw Goblins' powerful transactional distributed object programming system. This section shows the union of the two: that the relationships between Goblins objects is an excellent, expressive, and sufficient security model for networked programs.

To make this clear we will present a common tutorial: a blogging style system[10] (in our case, used by a community newspaper of an imagined town) with different users cooperating and performing different roles. Unlike most such tutorials, this is accomplished without an access control list: resources are protected from misuse without relying on checking the identity of the performing agent. Despite this, we will manage to introduce accountability and revocation features, the protection of misuse from unauthorized parties, and even the demonstration of a multiple-

10 This is not meant to be a "production-ready system", but an illustrative one. As one example limitation, the blog we will build is runtime-only and does not persist between processes to disk. However, the general ideas described are the foundation from which a more serious system could be built, and even persistence could be accomplished through the mechanisms described in Safe serialization and upgrade.

stakeholder cooperation pattern which has no direct parallel in an access control system.

3.2.1. Making and editing a blogpost

Lauren Ipsdale has decided to run a newspaper for her local community. The first thing Lauren will need is a way to construct individual posts which can be widely read, but edited only by trusted editors.

Lauren creates a new post:

```
REPL> define-values (day-in-park-post day-in-park-editor)
_____   spawn-post-and-editor
_____     . #:title "A Day in the Park"
_____     . #:author "Lauren Ipsdale"
_____     . #:body "It was a good day to take a walk..."
```

(We will show implementation details of these blogposts below, but first we will focus on narrative and use.)

`spawn-post-and-editor` returned two capabilities:

- `day-in-park-post`, which grants the authority to read Lauren's blogpost, but not to make changes to it.
- `day-in-park-editor`, which grants the authority to modify the blogpost.

Lauren wants the feedback of her friend Robert, but wants to decide whether or not to make or accept any changes herself. She shares `day-in-park-post` with Robert. Robert is able to view the post by running:

```
REPL> display-post day-in-park-post
```

Which prints out:

```
A Day in the Park
=================
  By: Lauren Ipsdale

It was a good day to take a walk...
```

Robert tells Lauren that he likes the blogpost, but that "a fine day" might sound more pleasant than "a good day" for the article's opening, and that maybe the name of the post should be "A Morning in the Park". Robert, not having access to `day-in-park-editor`, cannot make the changes himself.

Lauren deliberates on this feedback and decides that she agrees with the suggestion to change "good" to "fine" but that she thinks her title is good as-is. Lauren makes the change:

```
REPL> $ day-in-park-editor 'update
_____   . #:body "It was a fine day to take a walk..."
```

1. Implementation

   Since the "blog rendering" code is not essential to the demonstration of these security properties, that code is not shown in this section. However, it is available in Appendix: Utilities for rendering blog examples.

   The implementation of the post and editor pairs is fairly simple.

```
  define* (spawn-post-and-editor #:key title author body)
    ;; The public blogpost
```

```
    define (^post _bcom)
      methods
        ;; fetches title, author, and body, tags with '*post* symbol
        (get-content)
          define data-triple            ; assign data-triple to
            $ editor 'get-data          ;   the current data
          cons '*post* data-triple      ; return tagged with '*post*

    ;; The editing interface
    define (^editor bcom title author body)
      methods
        ;; update method can take keyword arguments for
        ;; title, author, and body, but defaults to their current
        ;; definitions
        (update #:key (title title) (author author) (body body))
          bcom : ^editor bcom title author body
        ;; get the current values for title, author, body as a list
        (get-data)
          list title author body

    ;; spawn and return the post and editor
    define post : spawn ^post
    define editor : spawn ^editor title author body
    values post editor      ; multi-value return of post, editor
```

This procedure takes three optional keyword arguments, the initial title, author, and body of the post.[11] (If not supplied, they will default to `#f`, meaning "false".) It returns two values, the `post` (which is the object which represents the readable blogpost), and the `editor`, which allows for editing what viewers of the `post` see.

In this system, the `editor` is the more powerful object. It contains two methods:

- `update`: Allows for changing the data associated with the post. The `bcom` operation calls `^editor` again, producing new behavior with the same `bcom` capability but updated (or not) versions of the `title`, `author`, and `body`.
- `get-data`: Retrieves the current title, author, and body associated with this post.

The `post` is considerably less powerful, and only has one method, `get-content`. Curiously, `get-content` is a thin wrapper around the `editor`'s `get-data`, merely tagging the returned data with the symbol `'*post*`.

2. Analysis

   With ordinary Goblins programming and a safe language environment, Lauren is able to construct separate post and editor capabilities which refer to the same blogpost. Lauren is able to choose who she hands these out to. Since Lauren shares the `post` capability with Robert but not the `editor` capability, Robert is able to read the blogpost, but there is no way for him to change its contents.

   All of this is accomplished without any attention by the underlying system to the identities of Lauren and Robert who are using the software, using ordinary reference passing behaviors. This is important, because in Capability security as ordinary programming we demonstrated that an identity-centric authority model is unsafe due to ambient authority and confused deputy problems. The solution we demonstrated of a capability security as ordinary argument passing extends into Goblins in a natural way. Since Goblins' object

11 Guile's `define` does not support keyword arguments, but `define*` does.

model is entirely built around behavior constructed from enclosed procedures, an object can only make use of the references to other objects it possesses in its scope.

We have also chosen in this example to have `post` be a comparatively thin object to `editor`, mostly proxying information which `editor` is in charge of, with a small type-tagging symbol added. This demonstrates how one one less powerful object can achieve most of its functionality by attenuating a more powerful object.

3.2.2. A blog to collect posts

Of course, a blogpost on its own is not itself a blog or newspaper. Lauren wants a collection of updated posts, not just a singular entry. Time to make the blog!

Lauren invokes `spawn-blog-and-admin`:

```
REPL> define-values (maple-valley-blog maple-valley-admin)
_____   spawn-blog-and-admin "Maple Valley News"
```

`spawn-blog-and-admin` returns two capabilities. The first is for the blog itself, which Lauren has locally bound to the variable `maple-valley-blog` only grants read access to the current set of posts. `maple-valley-admin` provides the ability to curate the set of posts itself. Lauren has a certain vision and standard of post quality she'd like to see held for Maple Valley News but would like it to be widely read, and thus she will share and encourage wide dissemination of the former capability but will more carefully guard the latter capability.

Since `maple-valley-blog` has just been initialized, it unsurprisingly reports having no posts:

```
REPL> $ maple-valley-blog 'get-posts
;; => ()
```

Since Lauren is now happy with `day-in-park-post`, she can add it via `maple-valley-admin`, and `maple-valley-blog` will now report the new post's addition:

```
REPL> $ maple-valley-admin 'add-post day-in-park-post
REPL> $ maple-valley-blog 'get-posts
;; => (#<local-object ^post>)
```

The blog can now also be read with `display-blog`:

```
REPL> display-blog maple-valley-blog
```

Which prints the following:

```
**********************
** Maple Valley News **
**********************

A Day in the Park
=================
  By: Lauren Ipsdale

It was a fine day to take a walk...
```

Robert tells Lauren he'd love to make an article of his own, and Lauren says she'd love to read it and see about including it. Robert pens a new post:

```
;; Run by Robert:
REPL> define-values (spelling-bee-post spelling-bee-editor)
_____  spawn-post-and-editor
_____    . #:title "Spelling Bee a Success"
```

```
_____     . #:author "Robert Busyfellow"
_____     . #:body "Maple Valley School held its annual spelling bee..."
```

Robert sends this to Lauren for review. Lauren says that it's good, but could use a catchier title. Robert's years of community newspaper reporting leaves him with exactly the right idea for a change:

```
;; Run by Robert:
REPL> $ spelling-bee-editor 'update
_____    . #:title "Town Buzzing About Spelling Bee"
```

Lauren checks the post and decides it's ready to go. She adds it to the blog:

```
REPL> $ maple-valley-admin 'add-post spelling-bee-post
```

Now `maple-valley-blog` is starting to look like it's got some real content going!

```
REPL> display-blog maple-valley-blog
**********************
** Maple Valley News **
**********************

Town Buzzing About Spelling Bee
===============================
  By: Robert Busyfellow

Maple Valley School held its annual spelling bee...


A Day in the Park
=================
  By: Lauren Ipsdale

It was a fine day to take a walk...
```

1. Implementation

   Here is the core implementation of spawn-blog-and-admin:

```
;; Blog main code
;; ==============

define (spawn-blog-and-admin title)
  define posts
    spawn ^cell '()

  define (^blog _bcom)
    methods
      (get-title)
        . title        ; return the title, as a value
      (get-posts)
        $ posts 'get   ; fetch and return the value of posts

  define (^admin bcom)
    methods
      (add-post post)
        define current-posts
          $ posts 'get
        define new-posts
          cons post current-posts  ; prepend post to current-posts
        $ posts 'set new-posts
```

```
    define blog : spawn ^blog
    define admin : spawn ^admin
    values blog admin
```

Here we see how lexical scope becomes a powerful feature for capability systems. `posts`, a cell which stores the current state of which articles are valid posts for this blog, is within the scope of the code for both `blog` and `admin`, which both utilize it within the scopes of their constructors `^blog` and `^admin` internally. However, while `blog` and `admin` are returned directly from `spawn-blog-and-admin`, `posts` never directly leaves the closure. Thus `posts` becomes a fully encapsulated coordination point between `blog` and `admin`.

2. Analysis

   The similarity between the patterns of `spawn-post-and-editor` and `spawn-blog-and-admin` is mostly clear, but what is interesting is in where they differ. While both return two capabilities, one effectively for reading and one effectively for writing, `spawn-post-and-editor` accomplished its job by having posts mostly proxy a subset of behavior of editors. In `spawn-blog-and-admin`, the roles are completely separated, and instead the encapsulated object of `posts` serves as the intermediary datastructure that the two other objects both use to coordinate reading current information (with `blog`) and writing current information (with `admin`).

3.2.3. Group-style editing

One implication from the way this code is currently written is that the blog is mostly a kind of aggregator of posts. While Lauren added Robert's post to Maple Valley News's collection of blogposts, since Robert did not share the edit capability with Lauren, Lauren cannot edit the post if she discovers a problem.

This can be an acceptable design, but Lauren has decided that she would like to ensure that any posts that are on the blog are editable by her or any other admins she gives access to. She also does not want to have to keep track of which edit capability is associated with which post: if she is looking at a post and catches an error, she wants to be able to jump straight into correcting it. Lauren wants to make sure her blogging administration software helps her ensure she is only adding objects which uphold these properties.

Under this rearchitecture, the admin interface is directly involved in constructing new posts and editors:

```
REPL> define-values (bumpy-ride-post bumpy-ride-editor)
_____   spawn-adminable-post-and-editor
_____     . maple-valley-admin
_____     . #:title "Main Street's Bumpy Ride"
_____     . #:author "Lauren Ipsdale"
```

Using this approach, Lauren could edit `bumpy-ride-post` using `bumpy-ride-editor`, but she does not need to since she can also use `maple-valley-admin` to edit:

```
REPL> $ maple-valley-admin 'edit-post
_____   . bumpy-ride-post
_____   . #:body "Anyone who's driven on main street recently..."
```

This new code also provides an assurance that any blogposts which are added are created through the internals of the code which runs "Maple Valley News". It will not be possible for any other

object to spoof being a post which will not grant a user of `maple-valley-admin` the ability to edit the post and still be added to the blog.

1. Pre-Implementation: Sealers and unsealers

   This example relies on a concept called "sealers and unsealers". *Sealers* and *unsealers* have an analogy with public key cryptography, where sealing resembles encryption, and unsealing resembles decryption. A third component, a *brand check predicate*, can check whether or not a sealed object was sealed by its corresponding sealer, and with a bit of work, we will show it can operate as the equivalent of signature verification. What is astounding is that all three of these operations can work without any cryptography at all, implemented purely in programming language abstractions. (The details of implementing sealers and unsealers can be seen in Appendix: Implementing sealers and unsealers.)

   To make this clearer, let us imagine a scenario where we are sealing lunchtime meals using sealers and unsealers. Our rival, who wishes to sabotage us, does the same:

```
REPL> define (our-lunch-seal our-lunch-unseal our-can?)
_____   make-sealer-triplet
REPL> define (rival-lunch-seal rival-lunch-unseal rival-can?)
_____   make-sealer-triplet
```

   We give our customer the unsealer, the delivery driver the brand predicate, and we keep the sealer privately to ourselves.

   The contents of sealed cans are private:

```
REPL> our-lunch-seal 'fried-rice
;; => #<seal>
```

   Our customer wants some chickpea salad, so we seal some for them:

```
REPL> define chickpea-lunch
_____   our-lunch-seal 'chickpea-salad
```

   Thankfully our truck driver is able to check that the food they are to deliver really is from us. (We have a reputation to uphold!)

```
REPL> our-can? chickpea-lunch
;; => #t (true)
REPL> our-can?
_____   rival-lunch-seal 'melted-ice-cream
;; => #f
```

   And the customer is able to open it just fine:

```
REPL> our-lunch-unseal chickpea-lunch
;; => 'chickpea-salad
```

   Whew!

2. Implementation

   We will have to re-architect our post/editor and blog/admin tooling to enable this new functionality, adding support for sealers and a few new methods.

   To accomplish this, we should first import pattern matching support, which we will use later:

```
use-modules
  ice-9 match      ; for pattern matching
```

We then implement our new version of post/editor spawning. This will no longer be used directly by users, so we also update its name, adding a `-internal` suffix.

```
define* (spawn-post-and-editor-internal blog-sealer #:key title author
body)
  ;; The public blogpost
  define (^post _bcom)
    methods
      ;; fetches title, author, and body, tags with '*post* symbol
      (get-content)
        define data-triple            ; assign data-triple to
          $ editor 'get-data          ;   the current data
        cons '*post* data-triple      ; return tagged with '*post*
      ;; *New*: get a sealed version of the editor from anywhere
      (get-sealed-editor)
        blog-sealer : list '*editor* editor
      ;; *New*: get a sealed version of self for self-attestation
      (get-sealed-self)
        blog-sealer : list '*post-self-proof* post

  ;; The editing interface
  define (^editor bcom title author body)
    methods
      (update #:key (title title) (author author) (body body))
        bcom : ^editor bcom title author body
      (get-data)
        list title author body

  ;; spawn and return the post and editor
  define post : spawn ^post
  define editor : spawn ^editor title author body
  values post editor
```

There are actually only three changes from our prior implementation, `spawn-post-and-editor`:

- This version takes one required argument, `blog-sealer`, which will be passed in by the admin object which creates the post/editor pair.
- We add two new methods to `post`:
  - `get-sealed-editor`: Uses `blog-sealer` to seal the corresponding `editor` object, allowing a relevant `admin` object to be able to unseal any post straight from the post itself (analogous to encryption). The `'*editor*` symbol is stored within the seal as a type tag indicating the *purpose* of the seal.
  - `get-sealed-self`: Uses `blog-sealer` to seal the post itself to attest to the `admin` that it was indeed created by the blog/admin code itself (analogous to a cryptographic signature). Like the previous method, it also stores a type tag within the seal indicating its purpose, here `'*post-self-proof*`.

We must also update our blog/admin spawning code so that it will be able to cooperate with the post/editor code we have just defined:

```
define (new-spawn-blog-and-admin title)
  ;; New: sealers / unsealers relevant to this blog
  define-values (blog-seal blog-unseal blog-sealed?)
    make-sealer-triplet

  define posts
```

```
      spawn ^cell '()

    define (^blog _bcom)
      methods
        (get-title)
          . title
        (get-posts)
          $ posts 'get

    define (^admin bcom)
      methods
        ;; *New:* A method to create posts specifically for this blog
        (new-post-and-editor #:key title author body)
          define-values (post editor)
            spawn-post-and-editor-internal
              . blog-seal
              . #:title title
              . #:author author
              . #:body body
          list post editor

        ;; *Updated:* check that a post was made (and is updateable)
        ;;   by this blog
        (add-post post)
          ;; (This part is the same as in the last version)
          define current-posts
            $ posts 'get
          define new-posts
            cons post current-posts  ; prepend post to current-posts
          ;; *New*: Ensure this is a post from this blog
          ;; This is accomplished by asking the post to provide the sealed
          ;; version "of itself".  The `blog-unseal` method will throw an
  error
          ;; if it is sealed by anything other than `blog-seal
          define post-self-proof
            $ post 'get-sealed-self
          match : blog-unseal post-self-proof
            ('*post-self-proof* obj)  ; match against tagged proof
              unless : eq? obj post   ; equality check: same object?
                error "Self-proof not for this post"
          ;; Checks out, let's update the set of posts
          $ posts 'set new-posts

        ;; *New:* A method to edit any post associated with this blog
        (edit-post post #:rest args)
          define sealed-editor
            $ post 'get-sealed-editor
          define editor
            match : blog-unseal sealed-editor
             ('*editor* editor)  ; match against tagged editor
               . editor
          apply $ editor 'update args

    values
      spawn ^blog
      spawn ^admin
```

Here we see several new additions:

- The blog calls `make-sealer-triplet` to instantiate `blog-seal` (the sealer), `blog-unseal` (the unsealer), and `blog-sealed?` (the brand-check predicate).

- `^admin` receives three key changes:
  - New method: `new-post-and-editor` is used to create post/editor pairs by running `spawn-post-and-editor-internal` (which was defined by the previous code block).
  - Updated method: `add-post` now checks that this is a post made by the blog itself. This is accomplished by asking the post for its supplied self-proof. This self-proof is returned sealed and must be unsealed by `blog-unseal`, which will throw an exception if not sealed by `blog-seal`, ensuring this is a post created by (and thus editable in the future by) the blog. The unsealed value should be a list tagged with the purpose of `'*post-self-proof*` and the object to check, the latter of which should have the same identity (compared via the identity-comparison procedure `eq?`) as `post`.
  - New method: `edit-post` allows for editing a post even without access to its corresponding editor object. This is accomplished by calling the `'get-sealed-editor` method on a post. The admin interface uses the `blog-unsealer` to extract the type-tagged editor. It uses `apply` to take the remaining arguments passed into edit-post and passes them along to the unsealed editor.

  Finally, this last bit is some convenience for consistency in our examples, since actors cannot return multiple values from their behavior

```
define (spawn-adminable-post-and-editor admin . args)
  define post-and-editor
    apply $ admin 'new-post-and-editor args
  match post-and-editor
    (post editor)          ; match against list of post and editor
      values post editor   ; return as values for consistency in examples
```

3. Analysis

   An administrator encountering a blogpost which is worth editing will want to edit it immediately. In an access control list style system, the way to accomplish this would be to assign users to an "editor" group, but we are building a system which aims to avoid the security problems associated with traditional access control list and related identity-centric authority systems.

   Instead, we take an approach called *rights amplification*: a sealed capability is attached to the post, giving access to the more powerful editor object, but this object can only be used through the corresponding unsealer. The only object empowered to make use of the unsealer is the blog's admin object, and so only by going through the admin is editing from the post possible.

3.2.4. Revocation and accountability

Lauren decides that it may be time for her to not be the only person running things, but she wants to make sure that she can hold anyone she gives access to accountable for the decisions they make and, if something inappropriate happens, revoke that access.

Lauren realizes she can extend her system to accommodate this plan *without rewriting any of the existing code*. Instead she will define some new abstractions that compositionally extend the system that exists.

The first thing she will need is a logger.

```
REPL> define admin-log
_____   spawn ^logger
```

Robert has been a great collaborator and has expressed interest in helping run things. Lauren decides it's time to take him up on it.

Lauren uses a new utility, `spawn-logged-revocable-proxy-pair`, which can proxy any object and log actions associated with a username meaningful to Lauren:

```
REPL> define-values (admin-for-robert roberts-admin-revoked?)
_____   spawn-logged-revocable-proxy-pair
_____     . "Robert"            ; username Lauren holds responsible
_____     . maple-valley-admin  ; object to proxy
_____     . admin-log           ; log to write to
```

The first of the two returned capabilities, `admin-for-robert`, is the one she sends Robert. The second, `roberts-admin-revoked?`, is the cell which defaults to false, but Lauren can set to be true at any time, at which point messages from Robert will no longer pass through.

Robert thanks Lauren for the capability and soon decides that Lauren's post would be better with a different title:

```
REPL> <- editor-for-robert 'edit-post bumpy-ride-post
_____    . #:title "Main Street Takes Some Bumps"
```

Later, Lauren suddenly notices with irritation that her blogpost isn't named what she remembered it being. She checks the log:

```
REPL> $ admin-log 'get-log
;; => ((*entry*
;;      user "Robert"
;;      object #<local-object ^admin>
;;      args (edit-post #<local-object ^post>
;;            #:title "Main Street Takes Some Bumps")))
```

Lauren decides that Robert shouldn't be editing her or anyone else's posts on the blog until they've had a serious conversation.

```
REPL> $ roberts-admin-revoked? 'set #t
```

Robert tries to make another edit to the blogpost and notices that it didn't go through. He sees a frustrated message in his inbox from Lauren and apologizes. The two of them agree on what the proper etiquette for editing someone else's post should be in the future and Lauren feels satisfied enough to renew Robert's access.

```
REPL> $ roberts-admin-revoked? 'set #f
```

1. Implementation

   The logger should look fairly familiar by now:

```
define (^logger _bcom)
  define log
    spawn ^cell '()   ; log starts out as the empty list

  methods
    ;; Add an entry to the log of:
    ;;  - the username accessing the log
    ;;  - the object they were accessing
    ;;  - the arguments they passed in
    (append-to-log username object args)
      define new-log-entry
```

```
          list '*entry* 'user username 'object object 'args args
        define current-log
          $ log 'get
        define new-log
          cons new-log-entry current-log  ; prepend new-log-entry
        $ log 'set new-log

      (get-log)
        $ log 'get
```

The revocable proxy pair takes the associated username, object to proxy, and log to write to:

```
define (spawn-logged-revocable-proxy-pair username object log)
  ;; The cell which keeps track of whether or not the proxy user's
  ;; access is revoked.
  define revoked?
    spawn ^cell #f

  ;; The proxy which both logs and forwards arguments (if not revoked)
  define (^proxy _bcom)
    lambda args
      ;; check if access has been revoked
      when ($ revoked? 'get)
        error "Access revoked!"
      ;; If not, first send a message to log the access
      <- log 'append-to-log username object args
      ;; Then proxy the invocation to the object asynchronously
      apply <- object args

  ;; The revoker only has one behavior.  It is called with no
  ;; arguments and revokes access to the proxy.
  define (^revoker _bcom)
    lambda ()
      $ revoked? 'set #t

  define proxy
    spawn ^proxy

  values proxy revoked?
```

It returns two cells, the proxy, and the cell which is used to control whether or not access is revoked.

2. Analysis

   Since Robert is never given access to the admin object directly, he has to operate using the `admin-for-robert` object which Lauren gives him. This object reports Robert's actions to a log which Lauren controls and will only operate if Lauren decides not to flip the `revoked?` cell to be true. Lauren is able to resume access through the capability should she so choose by flipping the `revoked?` cell's value back to false.

   Nothing is preventing Robert from sharing `admin-for-robert` with anyone else, but Lauren will hold Robert accountable for any actions taken with the `admin-for-robert` capability. This is a feature, and we will see it extended in the next section.

3.2.5. Guest post with review

Here's the story in shorthand, since I'm out of time:

- Robert has admin access resumed again

- He decides he's going on vacation but wants Matilda and her teacher to collaborate on a nice blogpost for the newspaper while he's gone.
- He spawns:

```
;; Robert's interactions
REPL> define-values (science-fair-post science-fair-editor science-fair-
reviewer)
_____   spawn-post-guest-editor-and-reviewer "Matilda Sample" admin-for-robert
```

- `science-fair-post` is given to Matilda and Teacher (come up with a name) both
- `science-fair-editor` is given to Matilda only
- `science-fair-reviewer` is given to the Teacher only
- Matilda writes her post's body and asks the teacher if it's good enough

```
;; Matilda's interactions
REPL> $ science-fair-editor 'set-body
_____   . "My name is Matilda and I am twelve. I won the science fair..."
```

- the teacher says "you forgot to give this a title, and make this more engaging, tell more of a story"

```
;; Matilda's interactions
REPL> $ science-fair-editor 'set-title
_____   . "Winning the Middle School Science Fair: A Personal Account"
REPL> $ science-fair-editor 'set-body
_____   . "At twelve years old, winning the local science fair has been..."
```

- the teacher looks at it again, looks good enough

```
;; Teacher's interactions
REPL> $ science-fair-reviewer 'approve
```

- and it goes live!

```
;; Runnable by any reader of the blog
REPL> display-blog maple-valley-blog
```

1. Implementation

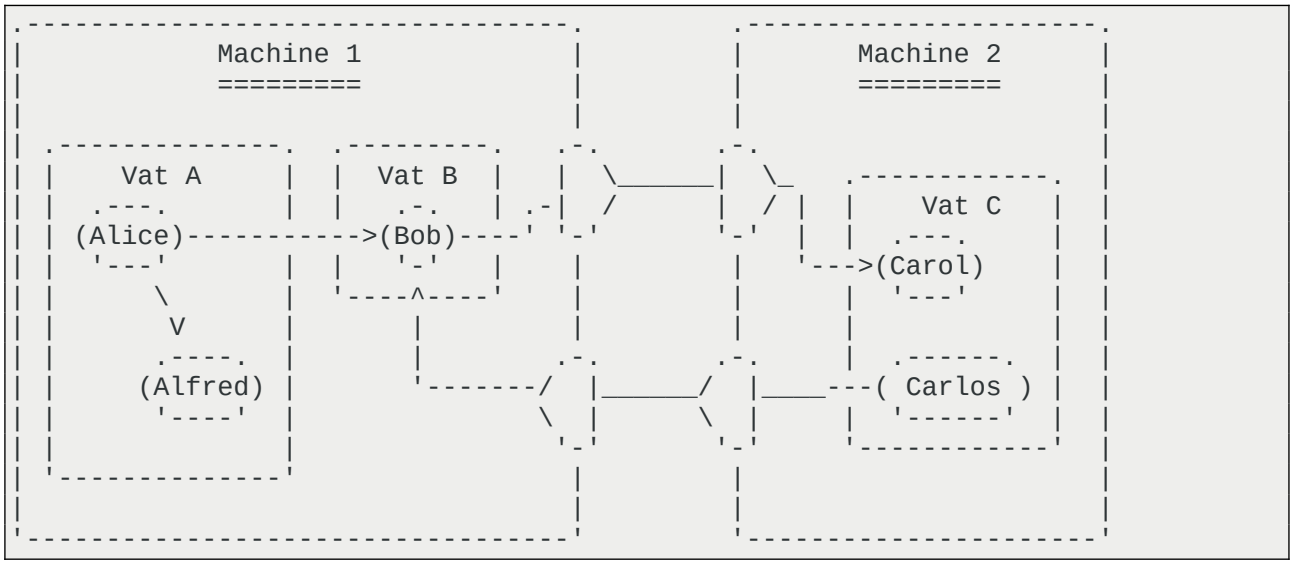
```
;;; Guest post with review
;;; ======================

;; The restricted-editor user can only change the title and body, but
;; not their name.
;; They cannot conspire with their teacher to be someone else on the
;; newspaper.
;;
;; The teacher cannot do anything but approve the student's post to
;; go live.  They cannot change the student's choice of language,
;; only ask them to change it before approval.

define (spawn-post-guest-editor-and-reviewer author blog-admin)
  define-values (post editor)
    spawn-post-and-editor #:author author

  define submitted-already?
    spawn ^cell #f

  define (ensure-not-submitted)
```

```
      when : $ submitted-already? 'get
        error "Already submitted!"

    define (^reviewer _bcom)
      methods
        (approve)
          ensure-not-submitted
          $ blog-admin 'add-post post
          $ submitted-already? 'set #t

    define (^restricted-editor _bcom)
      methods
        (set-title new-title)
          ensure-not-submitted
          $ editor 'update #:title new-title
        (set-body new-body)
          ensure-not-submitted
          $ editor 'update #:body new-body

    define reviewer : spawn ^reviewer
    define restricted-editor : spawn ^restricted-editor
    values post restricted-editor reviewer
```

2. Analysis

3.3. The vat model of computation

Goblins follows what is called the *vat model* of computation. A *vat* is simply an event loop that manages a set of objects which are *near* to each other (and similarly, objects outside of a vat are *far* from each other).



Objects which are *near* can perform synchronous call-return invocations in a manner familiar to most sequential programming languages used by most programmers today. Aside from being a somewhat more convenient way to program, sequential invocation is desirable because of cheap *transactionality*, which we shall expand on more later. In Goblins, we use the $ operator to perform synchronous operations.

Both *near* and *far* objects are able to invoke each other asynchronously using asynchronous message passing (in the same style as the *classic actor model*). It does not generally matter whether or not a *far* object is running within the same OS process or machine or one somewhere else on the

network for most programming tasks; asynchronous message passing works the same either way. In Goblins, we use the `<-` operator to perform asynchronous operations.

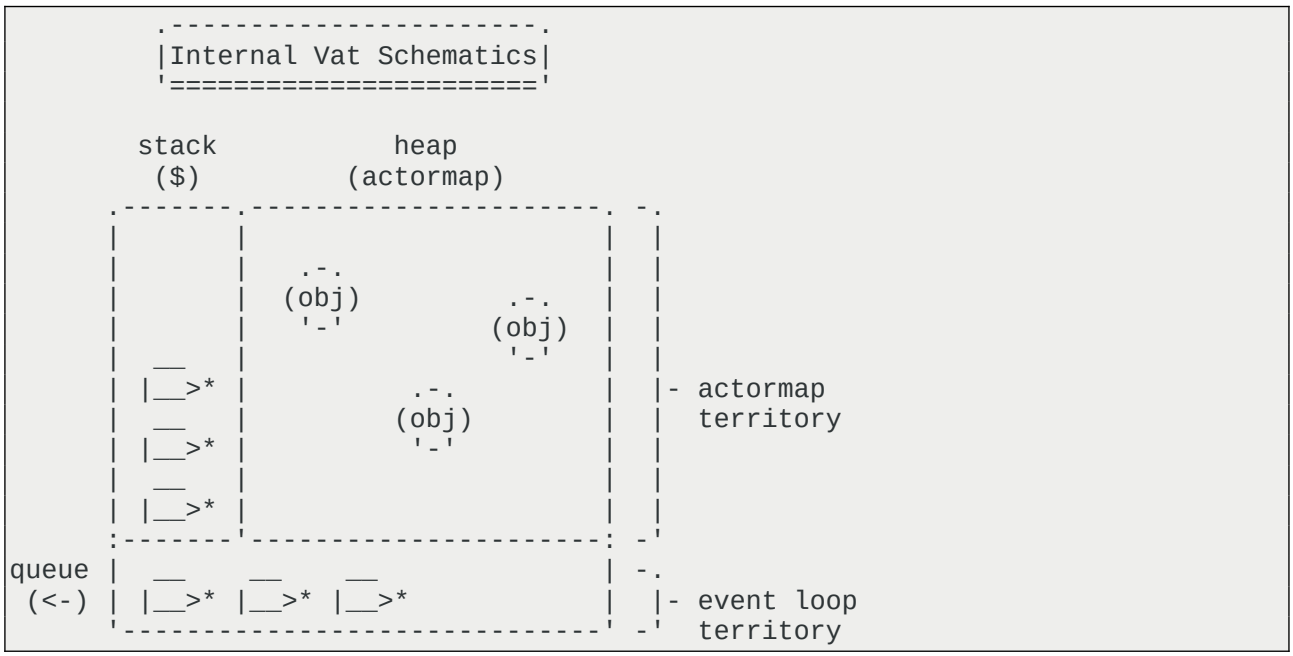
For both programmer convenience and for networked efficiency, Goblins supports *promise pipelining*: messages can be sent to promises which have not yet resolved, and will be forwarded to the target once the promise resolves. The sender of the message is handed back a promise to which it can supply callbacks, listening for the promise to be fulfilled with a value or broken (usually in case of an unexpected error).

As usual in the vat model of computation, individual message sends to a vat (event loop) are queued and then handled one *turn* at a time, akin to the way board game players take turns around a table (which is indeed the basis for the term *turn*).

The message, addressing a specific object, is passed to the recipient object's current behavior. This object may then invoke other *near* objects (residing within the same vat), which may themselves invoke other near objects in a synchronous and sequential call-return style familiar to most users of most contemporary programming languages. Any of these invoked objects may also change their state/behavior (behavior changes appear purely functional in Goblins; invocations of other actors do not), spawn new objects, invoke ordinary expressions from the host language, or send asynchronous messages to other objects (which are only sent if the *turn* completes successfully).

While the *vat model* of computation is not new (it originates in the E programming language and can trace some of its ideas back to E's predecessor Joule, and has since reappeared in systems such as Agoric's SwingSet kernel), Goblins brings some novel contributions to the table in terms of transactionality and time-travel debugging, enhancing an already powerful distributed programming paradigm.

3.4. Turns are cheap transactions



Special to Goblins is the transactional nature of *vat turns:* unhandled errors result in a *turn* being rolled back automatically (or more accurately, simply never being committed to the root transactional heap), preventing unintended data corruption. This cheap transactionality means that errors in Goblins are much less eventful and dangerous to deal with than in most asynchronous programming languages. Significantly less effort needs to be spent on cleanup when time is

reverted to a point where a mess never occured.[12]

3.5. Time-travel distributed debugging

The same transactional-heap design of Goblins can be used for other purposes. A distributed debugger inspired by E's Causeway is planned, complete with message-tracing mechanisms. This will be even more powerful when combined with already-demonstrated time travel features,[13] allowing programmers to debug a program in the state of an error when it occured.

3.6. Safe serialization and upgrade

```
Do you, Programmer,
 take this Object to be part of the persistent state of your application,
 to have and to hold,
 through maintenance and iterations,
 for past and future versions,
 as long as the application shall live?

  —Arturo Bejar
```

Processes crash or close and must be resumed. Behavior changes and representations must change to accommodate such change. Goblins has an integrated serialization mechanism which simplifies serialization and upgrade.

The need for state persistence and upgrade is hardly unique to Goblins programs. Much of programming traditionally involves reading and writing state of a program to a more persistent medium, generally files on a disk or some specialized database. Web applications in particular spend an enormous amount of effort moving between database representations and runtime behavior, but translating between runtime behavior and persistent state is typically disjoint and its solution space complicated.

Since Goblins' security model is encoded within the underlying runtime graph, manually scribing and restoring this structure would be a Sisyphean task in terms of labor and, should we naively trust objects' own self-descriptions, an entry point for vulnerability.

As an example, consider a multiplayer fantasy game might have to keep track of many rooms, the inhabitants of those rooms including various monsters and players, players' inventory, and many clever other objects and mechanisms which might even be defined while the game is running. Ad-hoc serialization of such a system would be too hard to keep our heads on straight about, and so we would like some way of having our system do the serialization of our process for us. Asking the objects to self-describe or manipulate the underlying database could also be dangerous, as objects

12 It is well known that the introduction of time and the introduction of local state are the same, introducing both benefits and costs. *Purely functional* systems model local state without introducing *side effects* by using *monads*, which re-introduces the benefits of time without being locked into changes which have occured. In other words: functional programming with monads grants freedom from time. Monads are powerful and beautiful constructs but are notorious for being difficult to learn to use (though learning to use them sometimes becomes a programmer point of pride), introducing enormous amounts of explicit plumbing outward to the user, threaded manually through a user's code. Goblins' design can be perceived as having an *implicit monad* which grants the user the benefits of time-travel without the explicit plumbing, allowing the user to focus on the core object behavior aspects of their program. The ability to be productively oblivious to the above is a goal: most users will never even know or consider the idea that Goblins contains an *implicit monad* unless they enjoy reading footnotes of architectural papers.

13 One early demonstration of this idea was shown in the runs-in-your-terminal space shooter game Terminal Phase, built as a demo to show off Spritely Goblins. The entire core game was built before even considering that time travel would be an easy feature to add, and a time travel demonstration was added within less than three hours changing no core game code but merely wrapping the toplevel of the program; its design fell out naturally from what Goblins already provided in the way it was used.

could claim to have authority that they do not… for example, in our game, we would not want player-built objects to be able to claim or dispense in-game currency or grant themselves powers which they did not originally have on restoration.

One option would be to use an underlying language runtime serialization system (many lisp and smalltalk systems have supported this for decades). However, this is wasteful; most serialized systems can be restored from a recipe of their construction rather than their current state at a fraction of the storage cost. Furthermore, the structure of our objects will be subject to change over time, and language-based process persistence misses out an opportunity to treat restoration as an opportunity for upgrade.

Spritely Goblins' solution is a serialization mechanism which asks objects how they would like to be serialized, but only allows objects to provide self-portraits utilizing the permissions they already have.[14,15] Goblins' serializer starts with root objects and calls a special serializer method on each object, asking each object for its depiction. This serialization mechanism is *sealed* off from normal usage; only the serializer can *unseal* it, preventing objects from interrogating each other for information or capabilities they should not have access to.[16]

Since walking the entire object graph is expensive, we can take advantage of reading turn-transaction-delta information to only serialize objects which have changed, making our serialization system performant.

The system is restored by walking the graph in reverse and applying each self-portrait to its build recipe. Restoring an object ends up being a great time to run upgrade code and as we build out Goblins we plan to capture many upgrade patterns into a common library.

The serialized graph can be used for another purpose: we can use it to create a running visualization of a stored ocap system, further helping programmers debug systems and understand the authority graph of a running system.

3.7. Distributed behavior and why we need it

In general when we have spoken so far in this paper of distributed objects, we have been referring to objects with one specific "location". But many systems are actually more complicated than this. For example, Alisha and Ben might both be in the same chatroom and there may be a distinct address for Alisha and Ben's personas; if we ask whether or not Carol means the same Alisha as Ben, she should have no problem saying "yes, this is the same person", and this can be as simple as address comparison.[17]

The Unum Pattern is a conceptual framework that encompasses the idea of a distributed abstract

14 The ideas for our serialization/upgrade mechanism stem from comments from comments by Jonathan A. Rees about "uneval" and "unapply" and the E programming language's Safe Serialization Under Mutual Suspicion paper (along with discussions between Randy Farmer and Mark S. Miller while at Electric Communities which preceded this).

15 Originally we had built this system as a separate mechanism we called *Aurie*, symbolized by a character made out of fire which was continuously extinguished and re-awakened like a phoenix. However we discovered that many programs, and even many of the standard library pieces which Goblins ships with, were in want of such a system, so Aurie's flame became folded into Goblins itself.

16 This is a common ocap pattern called *rights amplification* which is beyond the scope of this paper to explain but which is worth looking up.

17 Actually, saying that this is "as simple as address comparison" is the greatest misleading statement in this entire paper. Object identity through address comparison, frequently referred to as `EQ` based on the operator borrowed from lisp systems, is one of the most complicated talks debated in the object capability security community. See also the erights.org pages on Object Sameness and the Grant Matcher Puzzle. These are just the tip of the iceberg of `EQ` discussion and debate in the ocap community, and it's no surprise why: when identity is handled *incorrectly* it can accidentally behave as a *Access Control* List or inherit their problems of *ambient authority* and *confused deputies*. This is part of the value of finding patterns, to help prevent users from falling into these traps.

object with many different presences. One difference between the framing provided by the *unum pattern* and most other distributed pattern literature is that the *unum pattern* is particularly interested in *distributed behavior* rather than *distributed data*. Distributed data may be emergent from distributed behavior, but it is only one application. In the *unum pattern*, many different presences cooperate together performing different roles, sometimes even responding to messages in a manner semi-opaque to each other.

Consider a teacup sitting on a table in a virtual world. Where does it live? On the server? What about its representation in your client? What about the representation on another player's client? What about in your mind? While there is one *unum*, or "conceptual object", of the teacup, there are likely many *presences* representing it. Information and authority pertaining to the teacup may also be asymmetric;[18] you might know that the teacup has a secret note sealed inside it and I might not. While there may be one object which is the *canonical presence*, possibly serving as a source of shared identifier to refer to the object, the *canonical presence* is still a *presence*.[19]

Presences in Goblins typically correspond to Goblins objects.[20] The *unum pattern* is typically implemented via several messaging patterns: the reply pattern, the point-to-point pattern, the neighbor pattern, and the broadcast pattern. Keen observers might notice that a subset of the *unum pattern*, applied to data, is a publish-subscribe (PubSub) system, which is common in social media architecture design (ActivityPub is more or less a glorified data-centric publish-subscribe classic actor model implementation designed for social media on the web). For large-scale distribution of messages, the Amphitheater Pattern will be supported.

However, in recent times there have been advancements in convergent information architectures with research on conflict-free replicated data types. Goblins plans on implementing a standard library of CRDT patterns which can be thought of as a "unum construction kit".

4. OCapN: A Protocol for Secure, Distributed Systems

> Users have faced an impossible choice: between the full authority to get your work done and destroy your machine or authority so puny that you can't do anything useful with it. And if you grant full authority you are *toast!* Object capabilities enable you at many different scales to create easy-to-understand secure cooperation.
>
> If your cooperation has no security you will quickly find that the number of people you dare to cooperate with is limited. Unless you have security, you can only cooperate with your closest friends. By making this cooperation secure, we enable you to cooperate with people whom you do not fully trust. So if you want to do cooperation, you do indeed care about security.

18 Exploiting *asymmetric authority* is the very definition of the *confused deputy problem*. Its cause is usually emergent from ambient authority. Phishing attacks are an example of confused deputy problems where the confused deputy is a human being. Most object capability programming does not have confused deputy issues because to have a reference to a capability, in the general case, means to have authority to it. However, `EQ` and rights amplification (which bottoms out in a kind of `EQ`) both can re-introduce asymmetry, permitting confused deputies in careless designs, even to ocap systems. One might suggest removing identity comparison altogether from such systems, and for many ocap programs this is possible. However a *social system* is not very useful without identity, so we must develop patterns that treat identity with care.

19 The above explanation is modified directly from Chip Mornignstar's explanation of the Unum. Chip Morningstar co-founded both Lucasfilms Habitat and Electric Communities (with EC Habitat), both of which are enormous influences on Spritely's design.

20 Outside of Goblins, presences still may exist; it is still acceptable to consider your conception of a teacup to be a presence. Barring significant advancements in biomechanical integration, presences in your mind of a teacup probably are not represented directly by a Goblins object.

– Marc Stiegler, From Desktop to Donuts: Object-Caps Across Scales

OCapN (the Object Capability Network)

- CapTP: Distributed, secure networked object programming. Provides familiar message passing patterns, no distinction between asynchronous programming against local vs remote objects.
  - Distributed garbage collection: servers can cooperate to free resources that are no longer needed
  - Promise pipelining: massive parallelization and network optimization. provides convenience of sequential programming without round trips.
- Netlayers: abstract interface for establishing secure connections between two parties
  - Temporal connection abstraction: live session vs store and forward
  - Peer to peer and traditional client-server support
  - Quorum…! (Blockchains!)
- URI structure and certificates for establishing connections to servers and initial object connections

5. Application and library safety

- Language: All modules are untrusted by default. Loading a module doesn't mean it can do dangerous things.
- OS: All programs are untrusted by default. Loading a program doesn't mean it can do dangerous things.
- The same approach taken for ocap programming applies to language and OS safety: pass around capabilities to do things such as file operations, network access, reading from the keyboard, etc. (Solitaire example)
- "Sandboxing" is not enough if it doesn't allow for granting new permissions at runtime. You'll end up with too rigid of systems which means people will pass in more authority than they should. (Needs better framing, or link off-site to something.)
- Graphical interfaces for understanding and granting permissions

6. Portable, encrypted storage

- Some kinds of distributed files where the contents of the stored files aren't known or of concern to the hosting provider
- There is no "official" host for those files… anyone who has them can keep them alive
- We want both immutable file types and mutable types

7. Conclusions

8. Appendix: Related work

9. Appendix: Following the code examples

10. Appendix: A small scheme and wisp primer

11. Appendix: Utilities for rendering blog examples

```
;; Blogpost rendering utilities
;; ============================
use-modules
  ice-9 match  ; for pattern matching

define (display-post-content post-content)
  match post-content
    ('*post* post-title post-author post-body)
      let* ((title (or post-title "<<No Title>>"))
            (title-underline (make-string (string-length title) #\=))
            (author (or post-author "<<Anonymous>>"))
            (body (or post-body "<<Empty blogpost!>>")))
        display
          format #f "~a\n~a\n  By: ~a\n\n~a\n"
                 . title title-underline author body

define (display-blog-header blog-title)
  define header-len
    + 6 (string-length blog-title)
  define title-stars
    make-string header-len #\*
  display
    format #f "~a\n** ~a **\n~a\n"
           . title-stars blog-title title-stars

define (display-post post)
  display "\n"
  display-post-content
    $ post 'get-content
  display "\n"

define (display-blog blog)
  display-blog-header
    $ blog 'get-title
  for-each display-post
    $ blog 'get-posts
```

12. Appendix: Implementing sealers and unsealers

There are two ways to construct sealers and unsealers; one is the "coat check" pattern,[21] the other is the language-protected dynamic type construction pattern. The latter has less complications

21 The coat check pattern can be implemented and explained easily also: the coat is the value to be sealed, the sealer is the coat check desk, the ticket for later retrieval the sealed object, and the coat retrieval desk the unsealer. However this involves extra work to avoid garbage collection concerns amongst other issues; see "2.3.3 The Case for Kernel Support" in A Security Kernel Based on the Lambda Calculus.

surrounding garbage collection and leads to some real "a-ha" moments, so we will show that one here.

To make this work, we will use dynamically constructed type-records (as taken from the SRFI-9 scheme extension). In Guile, we must import the following:

```
use-modules
  srfi srfi-9
  srfi srfi-9 gnu
```

To understand how these records work in the general case, here is an example of a srfi-9 record used to define a 2d positional object which we'll call `<pos>`:

```
define-record-type <pos>   ; <pos>: name of the type
  make-pos x y     ; make-pos: constructor, takes two arguments
  . pos?           ; pos?: brand-check predicate (is it a pos?)
  x pos-x          ; pos-x: accessor for x
  y pos-y          ; pos-y: accessor for y
```

Use of this pos is simple enough:

```
REPL> define our-pos
_____   make-pos 2 3
REPL> pos-x our-pos
;; => 2
REPL> pos-y our-pos
;; => 3
```

Following from this, let's look at how `make-sealer-triplet` works:

```
;; Make a sealer, unsealer, and brand-check predicate using
;; dynamic type generation.
define (make-sealer-triplet)
  define-record-type <seal>
    seal val        ; constructor (sealer)
    . sealed?       ; predicate   (brand-check)
    val unseal      ; accessor    (unsealer)

  ;; Prevents snooping on contents at REPL, etc
  define (print-seal _rec port)
    display "#<sealed>" port
  set-record-type-printer! <seal> print-seal

  ;; Return sealer, unsealer, sealed? predicate
  values seal unseal sealed?
```

Within an invocation of `make-sealer-triplet`, we are defining a new `<seal>` type on the fly which will be completely distinct from any made during future invocations of `make-sealer-triplet`. The sealer is the constructor (accepting one argument, the sealed `val`), the brand-check is the type predicate, and the unsealer is the accessor of the sealed `val`. If running in a language argument which does not allow the user to piece apart a record without its corresponding accessor, there is no way to retrieve the associated value without the unsealer.

13. Appendix: Glossary

- **Actor:**
- **Ambient authority:**
- **Auditor:**

- **Authority:**
- **Behavior-oriented:**
- **Capability:**
- **CapTP:**
- **Causeway:**
- **Classic actor model:**
- **Confused deputy:**
- **Data-oriented:**
- **Distributed object programming:**
- **E:**
- **Guile:**
- **Joule:**
- **Lisp:**
- **Machine:**
- **Monad:**
- **Netlayer:**
- **Network:**
- **OCapN:**
- **Object:**
- **Object capability (security, discipline):**
- **Object graph:**
- **Permission:**
- **Presence:**
- **Promise pipelining:**
- **Racket:**
- **Rights amplification:**
- **Safe serialization:**
- **Sealers and unsealers:**
- **Scheme:**
- **Swingset:**
- **Spritely:**
- **Transaction:**
- **Turn:**
- **Unum:**
- **Uneval / Unapply:**

- **Vat:**
- **W7:**
- **Wisp:**

14. Appendix: Acknowledgments

An enormous number of people reviewed and provided feedback to this paper. Thank you to: Alan Karp, Baldur Jóhannsson, Chris Hibbert, Dan Connolly, Dan Finlay, Douglas Crockford, Jessica Tallon, Jonathan A. Rees, Jonathan Frederickson, Mark S. Miller, and Stephen Webber. (**NOTE:** if you think you should/shouldn't be on this list, let us know and we'll edit appropriately!)

Thank you to Mark S. Miller who personally spent enormous amounts of time walking Christine through object capability ideas through the years and provided guidance on how to properly represent granovetter diagrams (which, as applied to object capability systems, really are a powerful but underdocumented visual language). Thank you to Jessica Tallon who actively used Spritely Goblins during the production of this paper, allowing for feedback from direct experience, including many suggestions for improvements in the examples. Thank you to Arne Babenhauserheide, who developed the Wisp syntax for lisp used in this paper.

15. Appendix: ChangeLog

- `[2022-04-02 Sat]` release:
  - Promise pipelining examples added to A Taste of Goblins. This section was already planned but raised much interest in pre-review.
  - Make tagging list with cons in `^post` a bit easier to understand
  - First batch of the smaller of the changes suggested by Alan Karp (a whole bunch, should iterate…)
  - Incorporated feedback from Jessica Tallon
    - Explained how Solitaire gets access to keyboard and mouse
    - Switched reference from `^mcell` to `^cell`… oops, that's what I get from copy-pasting code from another document
    - Renamed `our-cgreeter` to `julius` in example
    - Fixed expected displayed message in "heard back" part
    - No longer use named let but the `^editor` contstructor, reflect that in surrounding text
    - Mention cons prepends to a list where appropriate
    - Fixed "Run by Robert" which had mistakenly said it was run by Lauren
    - Make it clearer that Lauren will hold Robert responsible for **anyone** who uses `admin-for-robert` (including someone Robert delegates authority to).
    - Moved sealers and unsealers implementation details to Appendix: Implementing sealers and unsealers