The Heart of Spritely: Distributed Objects and Capability Security

Christine Lemmer-Webber, Randy Farmer, Juliana Sims

Table of Contents

1. Introduction
2. Capability security as ordinary programming
3. Spritely Goblins: Distributed, transactional object programming
    - 3.1. On language and syntax choice
    - 3.2. A taste of Goblins
        - 3.2.1. A simple greeter
        - 3.2.2. State as updating behavior
        - 3.2.3. Objects which contain objects
        - 3.2.4. Asynchronous message passing
        - 3.2.5. Transactions make errors survivable
        - 3.2.6. Promise pipelining
        - 3.2.7. When schemes go awry: failure propagation through pipelines
    - 3.3. Security as relationships between objects
        - 3.3.1. Making and editing a blogpost
            - 3.3.1.1. Implementation
            - 3.3.1.2. Analysis
        - 3.3.2. A blog to collect posts
            - 3.3.2.1. Implementation
            - 3.3.2.2. Analysis
        - 3.3.3. Group-style editing
            - 3.3.3.1. Pre-Implementation: Sealers and unsealers
            - 3.3.3.2. Implementation
            - 3.3.3.3. Analysis
        - 3.3.4. Revocation and accountability
            - 3.3.4.1. Implementation
            - 3.3.4.2. Analysis
        - 3.3.5. Guest post with review
            - 3.3.5.1. Implementation
            - 3.3.5.2. Analysis
        - 3.3.6. Lessons learned
    - 3.4. Spritely Goblins as a society of networked objects
    - 3.5. The vat model of computation
    - 3.6. Turns are cheap transactions
    - 3.7. Time-travel distributed debugging
    - 3.8. Safe serialization and upgrade
    - 3.9. Distributed behavior and why we need it
4. OCapN: A Protocol for Secure, Distributed Systems

5. Application safety, library safety, and beyond
6. Portable encrypted storage
7. Conclusions
8. Appendix: On the choice of Scheme
9. Appendix: Lisp and Wisp
10. Appendix: Setting up Guile, Wisp, and Goblins
    10.1. Obtaining Guile
    10.2. Obtaining Wisp
    10.3. Obtaining Goblins
11. Appendix: Using vats in files
12. Appendix: Utilities for rendering blog examples
13. Appendix: Implementing sealers and unsealers
14. Appendix: Glossary
    14.1. Goblins and capability terminology
    14.2. Core goblins operations
    14.3. Portable encrypted storage specific terminology
15. Appendix: Acknowledgments
16. Appendix: ChangeLog
    16.1. [2023-09-26 Tue]
    16.2. [2022-07-01 Fri]
    16.3. [2022-06-30 Thu]
    16.4. [2022-06-28 Tue]
    16.5. [2022-06-27 Mon]
    16.6. [2022-06-26 Sun]
    16.7. [2022-06-24 Fri]
    16.8. [2022-06-23 Thu]
    16.9. [2022-06-22 Wed]
    16.10. [2022-06-21 Tue]
    16.11. [2022-06-20 Mon]
    16.12. [2022-06-18 Sat]
    16.13. [2022-06-17 Fri]
    16.14. [2022-06-16 Thu]
    16.15. [2022-06-15 Wed]
    16.16. [2022-06-14 Tue]
    16.17. [2022-06-11 Sat]
    16.18. [2022-06-10 Fri]
    16.19. [2022-06-09 Thu]
    16.20. [2022-06-08 Wed]
    16.21. [2022-06-07 Tue]
    16.22. [2022-04-02 Sat]
17. License

NOTE: This is an early draft, still under technical review.

This paper is the second in a three-part series outlining 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 details the core technical toolbox provided by Spritely Goblins and how it supplies the necessary features to feasibly build out Spritely's broader vision. The third paper in the series, Spritely for Secure Applications and Communities, ties the first two papers together by showing how the architecture for user-facing software fulfills the vision of the first paper and can be built on top of ideas from this paper.

Spritely's core tooling is generally useful and this paper may be independently of interest to people with a wide variety of programming backgrounds. The architecture of this paper is also designed with a purpose: to give us the firm footing to be able to achieve the ambitious journey of fulfilling the Spritely's user-facing vision. If your goal is to understand Spritely's full vision, it is our recommendation that you read each paper in order, however this is not a requirement.

This paper, like all of the Spritely Institute's work, is based on and is a contribution to open source and open standards.

1. Introduction

Building *peer-to-peer* applications on contemporary programming architecture is a complicated endeavor which requires careful planning, development, and maintenance. Building the kind of fully-decentralized design for healthy social community networks 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 the object capability security and programming language design communities.

Spritely's core layers of abstraction make building secure *peer-to-peer* applications as natural as any other programming model. 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, encrypt 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 Access Control Lists (ACL), the prevailing authority model, common to (and popularized by) Unix and nearly everything which has come before and 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!
SHELL> solitaire
```

In an ACL 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.[1]

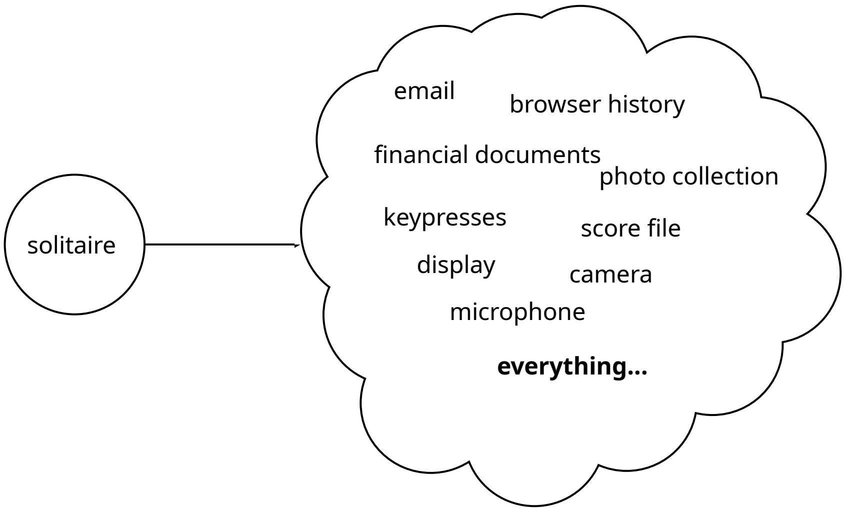

Figure 2.1:

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.

Imagine `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,[2] examples 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!
REPL> solitaire()
```

1 A malicious version of Solitaire is an example of the confused deputy problem. This is an issue which exists in Access Control List security where one program or a user can trick a program or user with greater privileges into doing something malicious. By delegating to or deputizing Solitaire with her privileges, Alisha opens the door to Solitaire abusing them.

2 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.

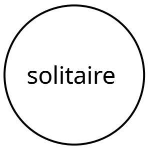

Figure 2.2:

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.

Consider a capability which has been granted more power by the underlying system, `makeWinCanvas(windowTitle)`. Say that `solitaire` can take a first argument which takes a window + canvas representing an object which is able to read keyboard and mouse input, but only while the window is active. A user 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
REPL> solitaireWin = makeWinCanvas("Safe Solitaire")
# Pass it to solitaire
REPL> solitaire(solitaireWin)
```

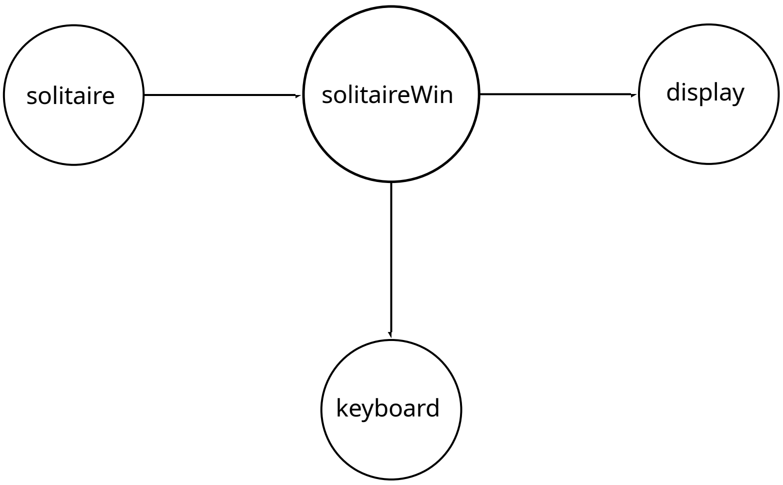

Figure 2.3:

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

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

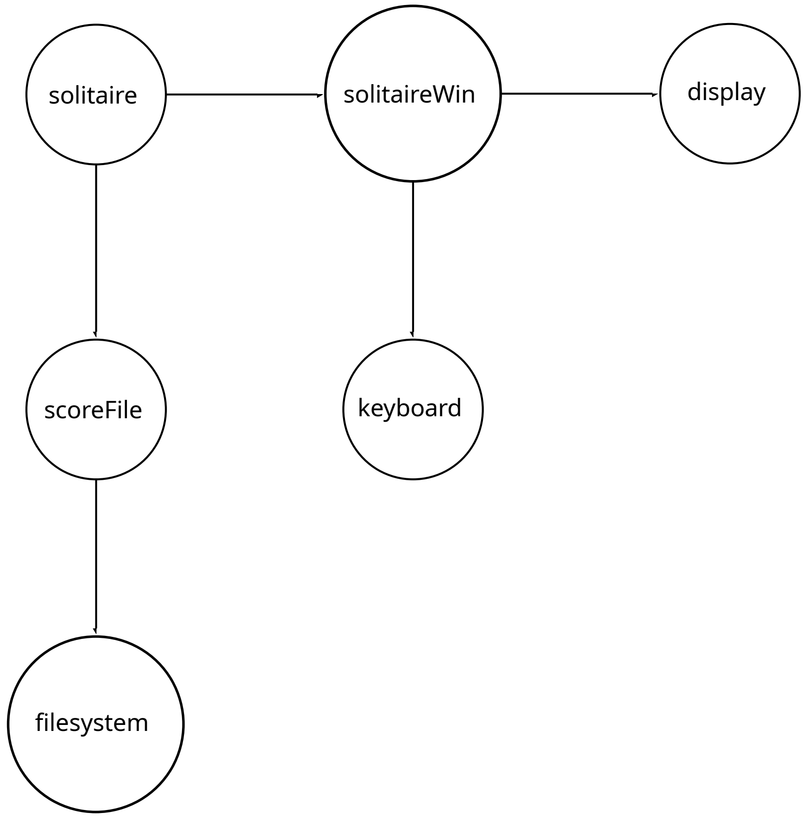

Figure 2.4:

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 it has been given access to, but it cannot do anything else dangerous.[3] It cannot access the network. It cannot read or write files from the filesystem arbitrarily (it can 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).[4]

This object capability security model is built 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.

3 Those experienced with Unix-like operating systems may be familiar with POSIX file handles. These are integer references to (open) files created by the kernel unique to the process requesting them which can be passed between processes and which give access to the underlying file. There is no need to search for the underlying file when given a file handle; the handle has all the information needed to access it. These handles can be thought of as capabilities.

4 Creating all of these various capabilities by hand each time a program is launched would be extremely tedious. Instead, in an object capability context, creating and handing off references would be handled automatically by existing capabilities so that users would not have to think about it. In this example, the executing enviroment may implicitly pass in `scoreFile` and `solitaireWin` to an invocation of `solitaire()`.

3. Spritely Goblins: Distributed, transactional object programming

At the heart of Spritely is Goblins, its *distributed object programming* environment.[5] Goblins provides an intuitive security model, automatic local *transactions* for locally synchronous operations, and easy to use and efficient asynchronous communication with *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 network protocol design. Goblins also integrates powerful distributed debugging tools, and a process persistence and upgrade model which respects its security fundamentals.[6]

Within Goblins, discussion of a *distributed object* is referring to a model where many independent objects communicate with other objects on many different machines. In other words, *distributed object programming* means "a distributed network of independent objects".[7] 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 away so the developer can focus on object behavior and interactions.

3.1. On language and syntax choice

The following examples will illustrate Goblins using its implementation in *Guile* (which is a *dialect* of *Scheme*, which is itself a *dialect* of *Lisp*).[8] While the ideas here could be ported across many

5 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.

6 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.

7 This is not to be confused with "the abstract conceptual objects themselves are distributed/replicated across different machines", addressed as the *Unum Pattern* in the Distributed behavior and why we need it section.

Similarly this does 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.

8 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

kinds of 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. See A Scheme Primer if you'd like an introduction to Scheme.

This document uses an unusual representation of Lisp syntax which is whitespace-based instead of parenthetical, named Wisp.[9] 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. These examples aim to be as simple as possible to understand just by reading them. If you'd like a more thorough explanation of how Lisp and Wisp relate, see Appendix: Lisp and Wisp.

Lisp and Scheme programming (not only, but especially) tends to involve a cycle between experimenting at the interactive *REPL* and code you keep around in a file. This paper follows the same convention. 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:

```
REPL> define name "Doris"
REPL> string-append "Hello " name "!"
; => "Hello Doris!"
REPL> display "Hello screen output!\n"
; prints: Hello screen output!
```

To get your *REPL* set up properly for live programming, you will need to setup Guile, Goblins, and Wisp.

3.2. A taste of Goblins

The following section gives a high-level demonstration of Goblins through practical use.

If you do choose to follow along by entering the code from this section, you can define this as a full-fledged module, say perhaps `taste-of-goblins.w`, like so:

```
define-module : taste-of-goblins
  . #:use-module : goblins
  . #:use-module : goblins actor-lib methods
  . #:export (^cell ^greeter ^cgreeter ^borked-cgreeter
              ^car-factory ^borked-car-factory)
```

Code examples that are *not* interactive can/should be entered into this file.

Now you're ready to go. Read on!

3.2.1. A simple greeter

This is an extremely brief taste of what programming in Goblins is like. The following code is adapted from the Guile version of Goblins.[9]

To work with Goblins objects, a vat is required. Vats will be explained more later, but for now, think of a vat as a context for Goblins objects. Here's how to make and enter one interactively:

```
;; define with next argument *not* in parentheses
```

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

9 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 parenthetical form and the whitespace-based form bidirectionally with few key rules.

```
;; defines an ordinary variable
REPL> define a-vat
_____    spawn-vat
REPL> ,enter-vat a-vat
REPL [1]>
```

Entering a vat introduces a new layer of abstraction to the REPL and creates a sub-REPL, indicated by `[1]`. Errors may result in the user entering a new debugger sub-REPL, also indicated by `[1]`. The number in brackets is incremented for each new sub-REPL. Any code examples in this document which use `[1]` should be run inside a vat - that is, after an `,enter-vat` command like the one above - even if this command is not written out. You can exit a vat (or any sub-REPL) with `,q`:

```
REPL [1]> ,q
REPL>
```

Note that this only works in an interactive context at a REPL. For how to work with vats statically - inside a file - see Appendix: Using vats in files.

Now, implement a friend who will greet you:

```
;; 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.[10] 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 try instantiating one:[11]

```
;; define with next argument *not* in parentheses
;; defines an ordinary variable
REPL [1]> define gary
_________    spawn ^greeter "Gary"
```

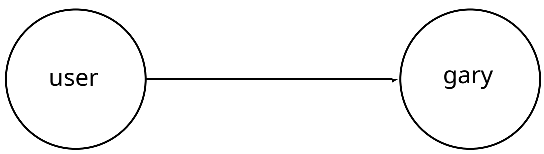

Figure 3.2.1:

As you can see, `spawn`'s first argument is the constructor for the Goblins object which will be spawned. For now, ignore the `bcom`, which is not used in this first example. The rest of the

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

11 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.

arguments to spawn are passed in as the rest of the arguments to the constructor.[12] So in this 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. You can now invoke your gary friend using the synchronous call-return $ operator (not to be confused for a Unix command prompt):

```
REPL [1]> $ gary "Alice"
;; => "Hello Alice, my name is Gary!"
```

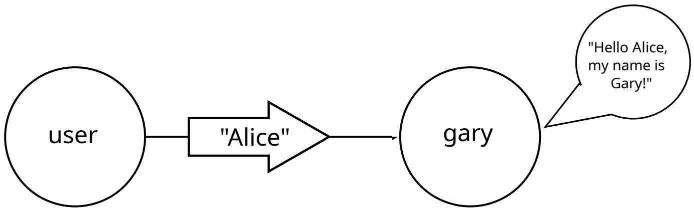

Figure 3.2.2:

As you 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.

3.2.2. State as updating behavior

Here is 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
```

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

```
REPL [1]> define chest
_________    spawn ^cell "sword"
REPL [1]> $ chest 'get
;; => "sword"
REPL [1]> $ chest 'set "gold"
REPL [1]> $ chest 'get
;; => "gold"
```

12 spawn invokes a constructor for an object and returns a reference to that object, which may lead one to think of it in terms of new or make from object-oriented languages. However, unlike those keywords, spawn does not exist primarily to indicate new heap values; rather, it wraps the construction of an object in an object capability secure manner. spawn creates and manages the bcom capability (unique to each object), as well as places the object into the actor map of the vat where it is created. These topics will be discussed more later.

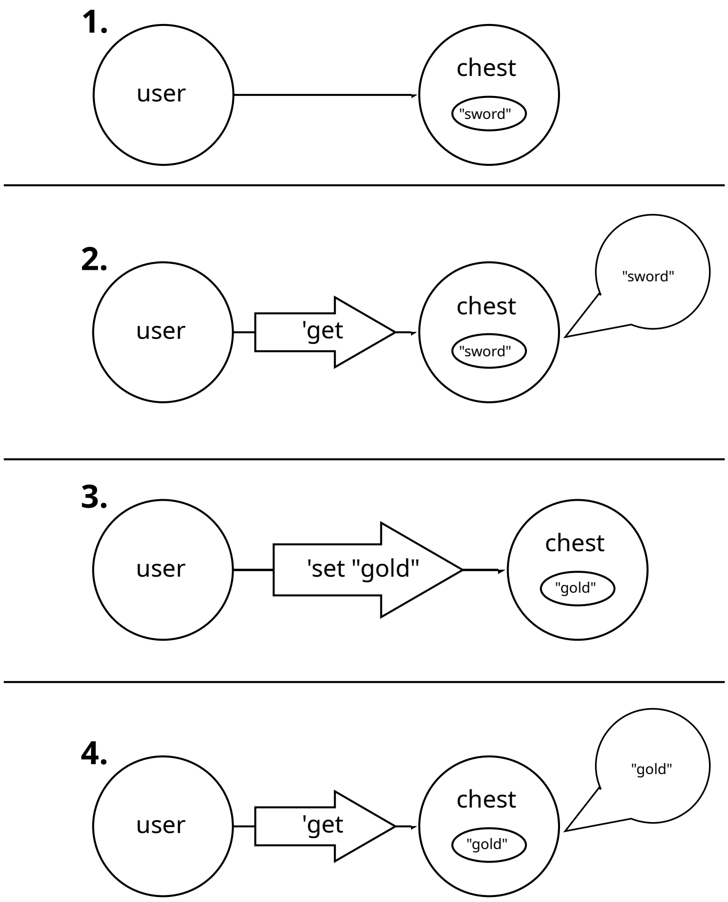

Figure 3.2.3:

Now you 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"!)[13]

`methods` was also new to this version. It turns out that `methods` is simply syntax sugar. 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.[14]

3.2.3. 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" called `^cgreeter`:

```
define (^cgreeter bcom our-name)
  define times-called   ; keeps track of how many times 'greet called
```

13 Objects in Goblins derive their functionality from "behaviors", which are simply procedures. `bcom` allows an object to specify what functionality it would like to have - what behavior it would like to *become* - the next time it is invoked. Together, these features allow Goblins to be quasi-functional, and enable transactionality and time travel capabilities.

14 `methods` is a macro which returns a procedure which supports symbol dispatch on its first argument. Macros are one of the key features of Scheme that make it such a powerful and expressive language. You can read more about them in the Scheme Primer section "On the extensibility of Scheme (and Lisps in general)", which even includes an implementation of `methods` itself!

```
    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 you can see near the top, `times-called` is instantiated as an `^cell` like the one 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 [1]> define julius
            spawn ^cgreeter "Julius"
REPL [1]> $ julius 'get-times-called
;; => 0
REPL [1]> $ julius 'greet "Gaius"
;; => "[1] Hello Gaius, my name is Julius!"
REPL [1]> $ julius 'greet "Brutus"
;; => "[2] Hello Brutus, my name is Julius!"
REPL [1]> $ julius 'get-times-called
;; => 2
```

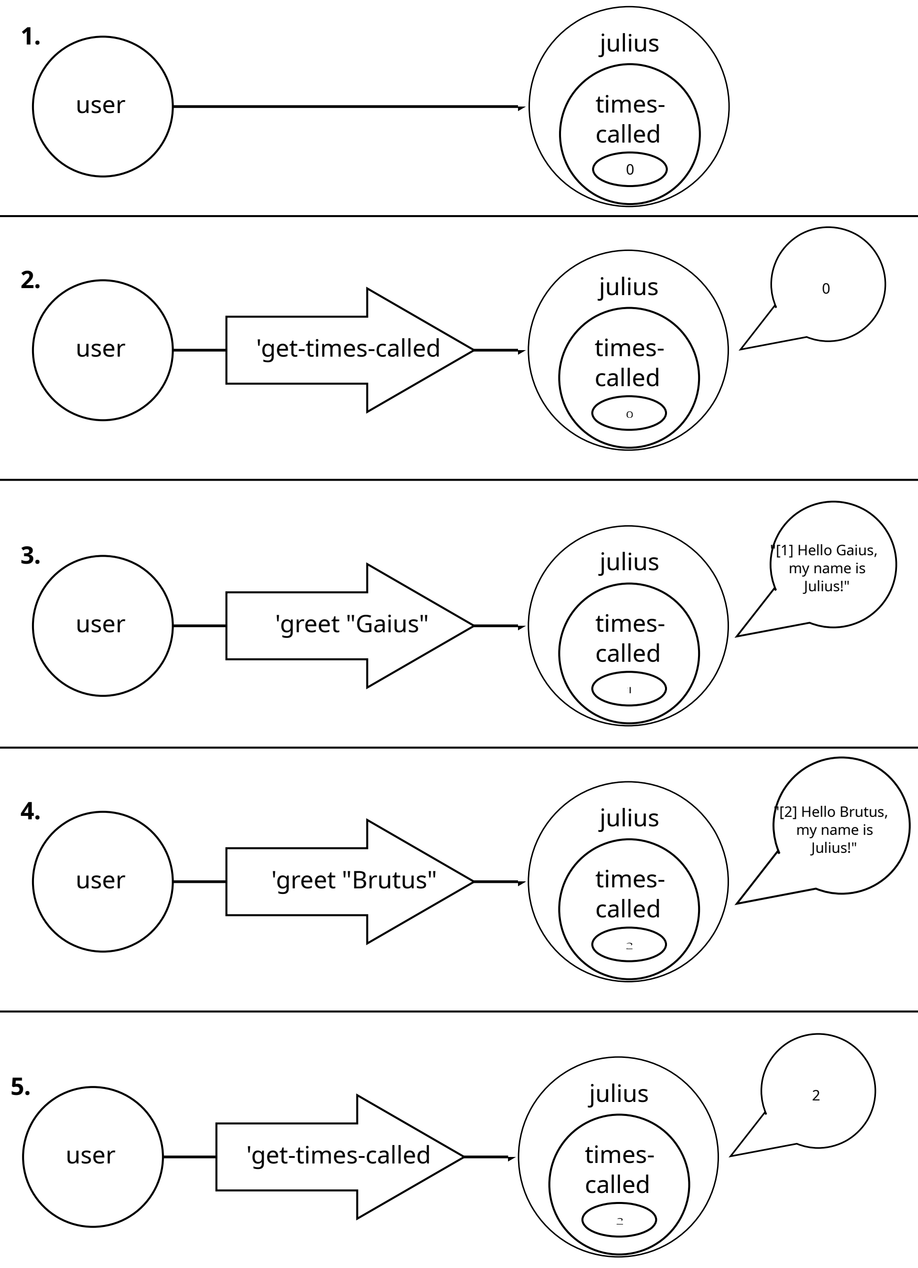

Figure 3.2.4:

3.2.4. Asynchronous message passing

You have seen 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, how is that done?

You can simulate a distributed context by creating and entering a new vat:

```
REPL [1]> ,q
REPL> define b-vat
_____   spawn-vat
REPL> ,enter-vat b-vat
```

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

```
REPL [1]> <- julius 'greet "Lear"
;; => #<promise>
```

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

```
REPL [1]> on (<- julius 'greet "Lear")
_________   lambda (got-back)
_________     format #t "Heard back: ~a\n"
_________            . got-back
; prints (eventually):
;   Heard back: [4] Hello Lear, my name is Julius!
```

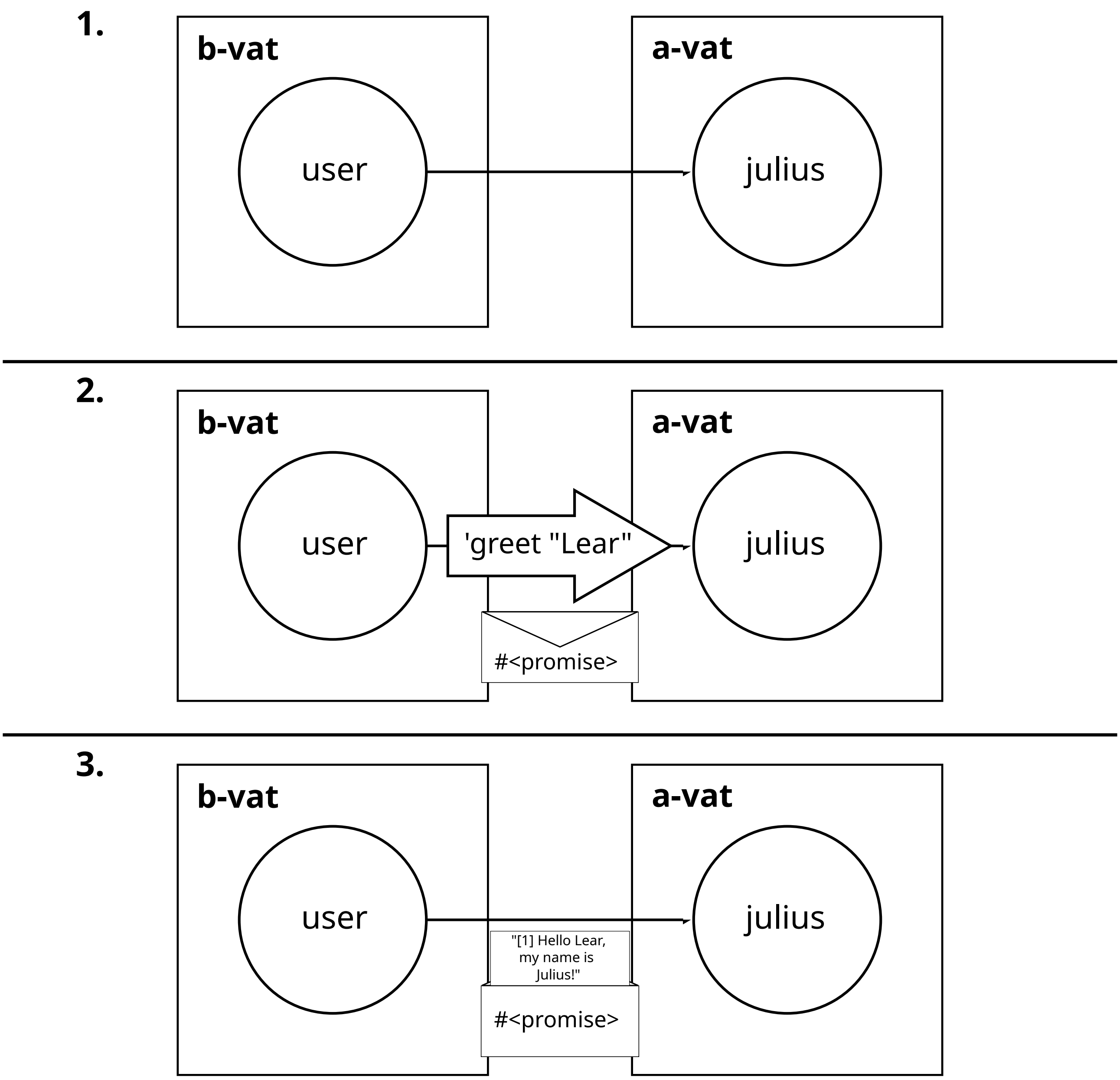

Figure 3.2.5:

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> ,enter-vat a-vat
REPL [1]> define broken-bob
_________   spawn ^broken-bob
REPL [1]> ,q
REPL> ,enter-vat b-vat
REPL [1]> 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"
; prints (eventually):
;   Got an error: <error ...>
;   Whew, it's over!
```

3.2.5. Transactions make errors survivable

Mistakes happen, and when they do, the damage should be minimal. But with many moving parts, accomplishing this can be difficult.

However, Goblins makes life easier. To see how, 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 spawn this friend and invoke it:

```
REPL> define horatio
_____   spawn ^borked-cgreeter "Horatio"
REPL> ,enter-vat a-vat
REPL [1]> $ horatio 'get-times-called
;; => 0
REPL [1]> $ 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! You can see from the `pk` debugging that the `times-called` cell should be incremented to 1. And yet…

```
REPL [1]> $ horatio 'get-times-called
;; => 0
```

This is covered 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.

3.2.6. 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[15] 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[16]

15 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 Promise Pipelining page from erights.org and sections 2.5 and 16.2 of Mark Miller's dissertation.

Note that if you are familiar with promises in Javascript, those are also inspired by E (and its predecessor Joule)'s promises. However, the full version of promises, including promise pipelining (or its most powerful use combined with network programming) were never included in Javascript proper. E's full vision of promises are present in Spritely Goblins, as outlined here.

16 Promises without promise pipelining are already an improvement over raw callbacks but are still insufficiently ergonomic for convenient programming. "Callback hell" and the annoyance of "`.then()` chaining" have lead many developers to prefer coroutines via `async` and `await` type operators. Goblins does have support for coroutines, but their use is somewhat cautioned against, and they are not prioritized. Coroutines give the illusion of straightahead call-return style programming by flattening callback structures. Unfortunately, while call-return programming is synchronous, coroutines are really "splitchronous"… each invocation of `await` splits time. `await` makes it very easy to accidentally mistake splitchronous code as being synchronous code, but the difference is severe: the world can change around the user during the time between a coroutine's suspension and resumption, opening up a class of vulnerabilities known as "re-entrancy attacks". This risk was observed during E's development and lead E to not include coroutines at all. A couple of decades later, re-entrancy attacks became the number one way money has been stolen in Ethereum due to bugs in smart contracts.

But there is another reason to prefer promise pipelining over coroutines: the reduction of round-trips! A coroutine requires waiting for a response to come back before deciding upon the next action, which is not a requirement for a promise pipelining based system.

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 to create.
  define (^car bcom model color)
    methods                        ; methods for the ^car
      (drive)                      ; drive the car
        format #f "*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 factor called `fork-motors`. If you type this in, you'll see an error:

```
;; Interaction on machine A
REPL> ,enter-vat a-vat
REPL [1]> define fork-motors
_________   spawn ^car-factory "Fork"
```

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

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> ,enter-vat b-vat
REPL [1]> define car-vow
```

```
__________   <- fork-motors 'make-car "Explorist" "blue"
```

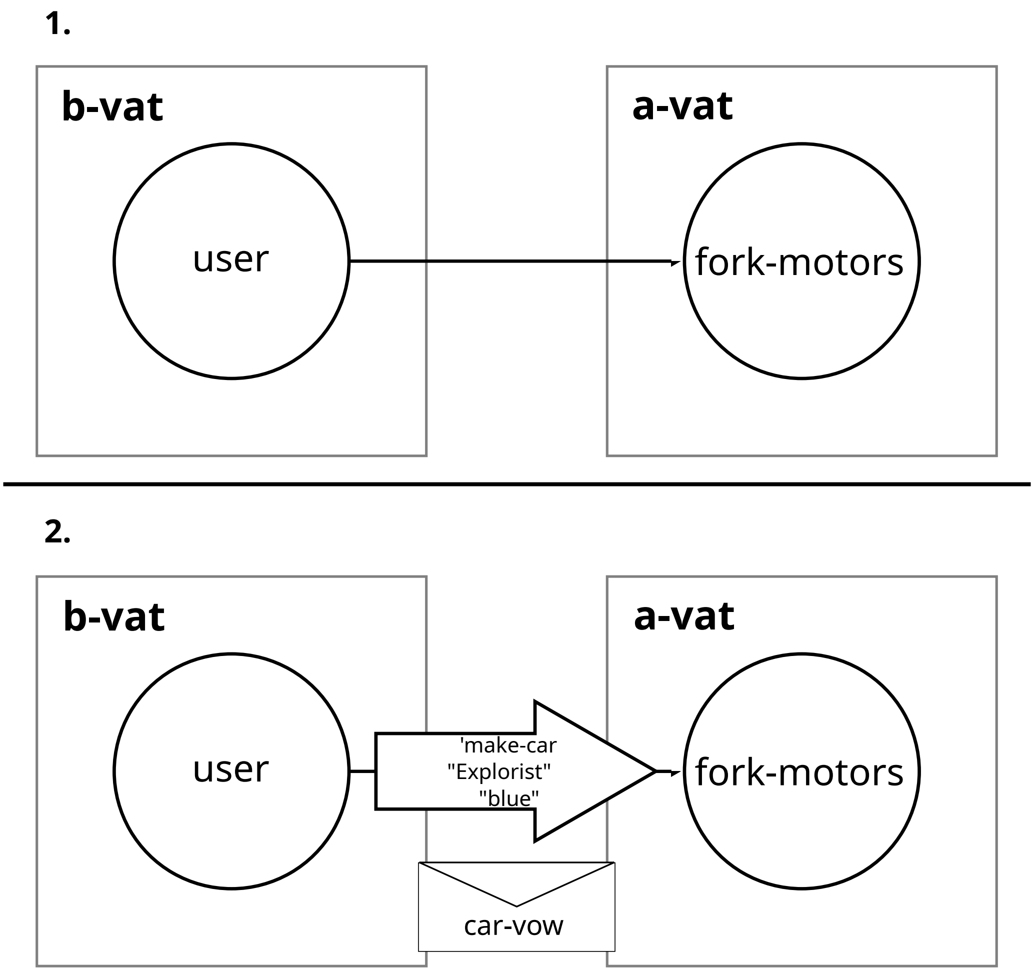

Figure 3.2.6:

So you have a *promise* to a future car reference, but not the reference itself. You 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 already shown (and following the pattern most other distributed programming systems use), you would end up with something like:

```
;; Interaction on machine B, communicating with A
REPL [1]> 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 you can message our-car
__________       lambda (val)           ; A->B: result of that message
__________         format #t "Heard: ~a\n" val
; prints (eventually):
;   Heard: *Vroom vroom!*  You drive your blue Fork Explorist!
```

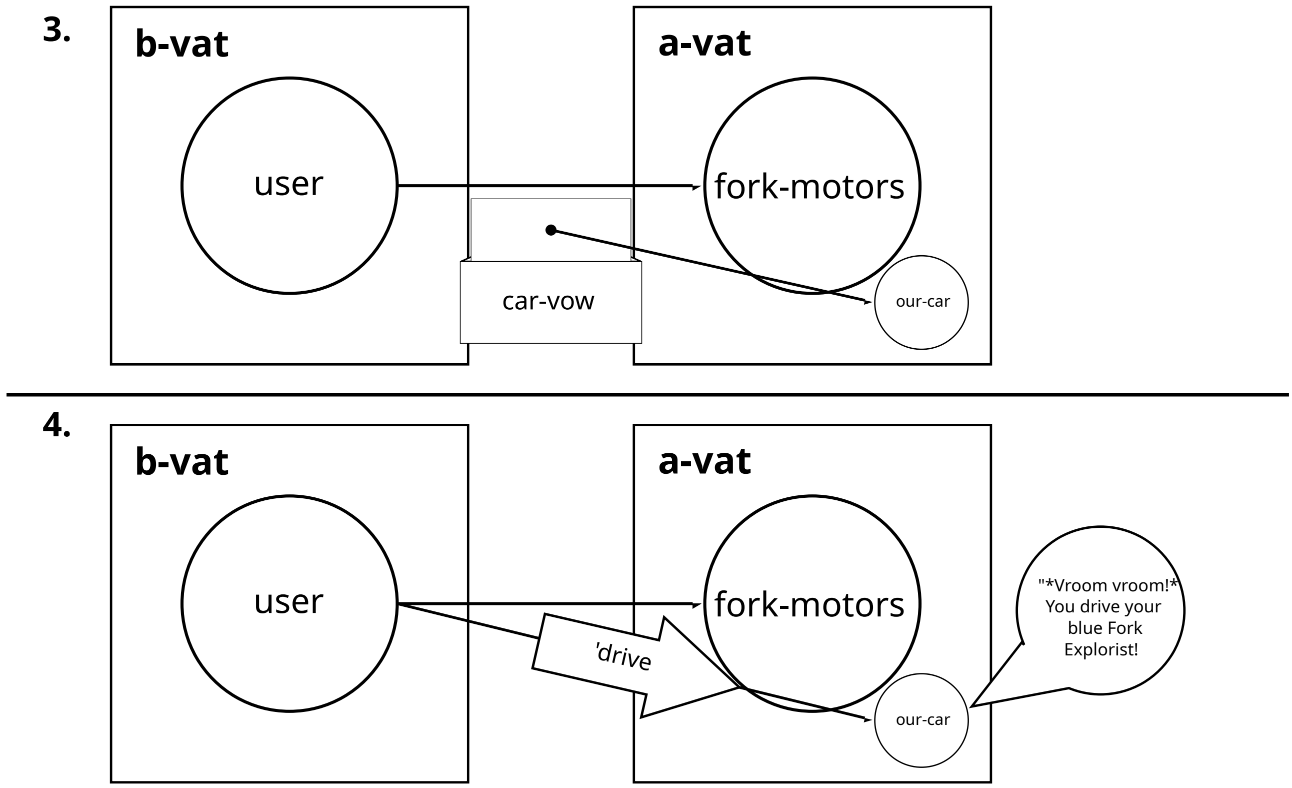

Figure 3.2.7:

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

```
;; Interaction on machine B, communicating with A
REPL [1]> on (<- car-vow 'drive)      ; B->A: send message to future car
__________   lambda (val)             ; A->B: result of that message
__________     format #t "Heard: ~a\n" val
; prints (eventually):
;   Heard: *Vroom vroom!*  You drive your blue Fork Explorist!
```

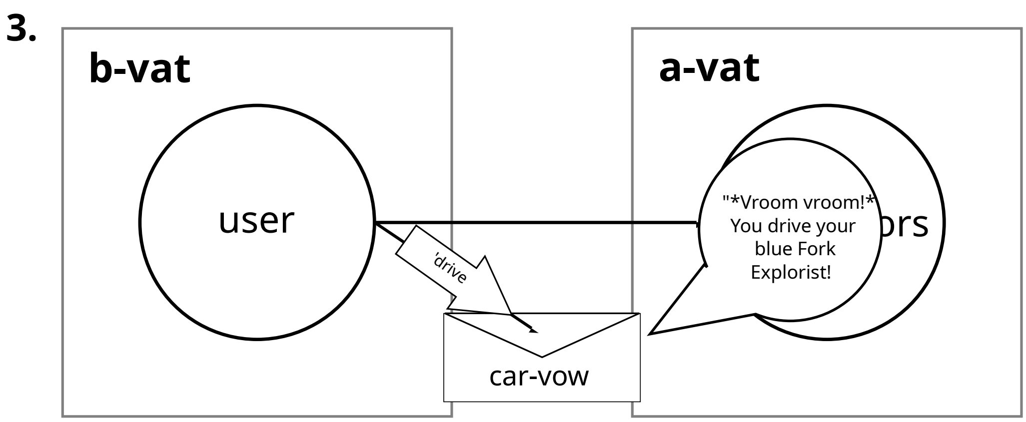

Figure 3.2.8:

While clearly a considerable programming convenience, the other advantage of promise pipelining is a reduction of round-trips, whether between 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. 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.

3.2.7. When schemes go awry: failure propagation through pipelines

```
Thy wee bit heap o' leaves an' stibble, Has cost thee mony a weary nibble! Now
thou's turn'd out, for a' thy trouble, But house or hald, To thole the winter's
sleety dribble, An' cranreuch cauld!

 But, Mousie, thou art no thy-lane, In proving foresight may be vain; The best-
laid schemes o' mice an' men Gang aft agley, An' lea'e us nought but grief an'
pain, For promis'd joy!

 Still thou art blest, compar'd wi' me The present only toucheth thee: But, Och!
I backward cast my e'e. On prospects drear! An' forward, tho' I canna see, I
guess an' fear!

 — From "To a Mouse, on Turning Her Up in Her Nest With the Plough" by Robert
Burns, 1785
```

Unexpected behavior can cause a cascade of failures. In a synchronous call-return system with exceptions, raising an exception causes not only the current procedure invocation to fail, but further invocations up the chain until the exception is caught (and if uncaught, possibly by allowing the program as a whole to fail). While potentially frustrating to encounter as a programmer or user, the alternative of proceeding without mitigating unhandled behavior could be equally disastrous. Still, if you interpret each procedure as voluntarily "sending a message to its caller" that something has gone awry, you can see the great service that each callee performs for its caller (such a pattern is common when a language does not provide implicit exception support), allowing the caller to make new plans, or at least not move forward under assumptions that no longer hold. Even unhandled exceptions, observed by the programmer, can be an opportunity to study and make new plans so that things may work better next time.

In a highly asynchronous networked environment, the likelihood of unanticipated failures grows substantially. Even with the most well implemented, bug-free *locally implemented* code (itself usually less likely a possibility than its authors may think), network connections are fickle, and remote objects may misbehave. As such, if a promise is broken, a pipelined message to that promise will have nowhere to go. This too should be interpreted as a failure and handled correctly.

As an example of this, consider this broken implementation of a car factory:

```
define (^borked-car-factory bcom company-name)
  define (^car bcom model color)
    methods                       ; methods for the ^car
      (drive)                     ; drive the car
        format #f "*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
      error "Your car exploded on the factory floor!  Ooops!"
      spawn ^car model color
```

What would happen if you tried making a car using this factory and then pipeline a message to drive it?

```
REPL [1]> define forked-motors
_________   spawn ^borked-car-factory "Forked"
REPL [1]> define car-vow
_________   <- forked-motors 'make-car "Exploder" "red"
REPL [1]> define drive-noise-vow
            <- car-vow 'drive
_________
REPL [1]> on drive-noise-vow
_________    lambda (val)
_________      format #t "Heard: ~a\n" val
_________    . #:catch
_________    lambda (err)
_________      format #t "Caught: ~a\n" err
; prints (eventually):
;   Caught: <error...>
```

Even though it is `car-vow` which is initially broken, its exception propagates to `drive-noise-vow`. Since there would be no useful way to drive a broken promise of a car anyhow, this is the correct design, and the situation can be detected and dealt with.

3.3. 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

Capability security as ordinary programming 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. A taste of Goblins showed Goblins' powerful *transactional* distributed object programming system. This section shows the union of the two: that the relationships between Goblins objects is a powerful, expressive, and robust security model for networked programs.

What follows is a common tutorial to make this clear: a blogging style system[17] (in this case, used

17 This is not meant to be a "production-ready system", but an illustrative one. As one example limitation, the blog is runtime-only and does not persist between processes to disk. However, the general ideas described are the

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, it will manage to introduce accountability and revocation features, the protection of misuse from unauthorized parties, and even the demonstration of a multiple-stakeholder cooperation pattern which has no direct parallel in an access control system.

3.3.1. 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 [1]> 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..."
```

(Implementation details of these blogposts will follow, but first this tutorial 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:

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.

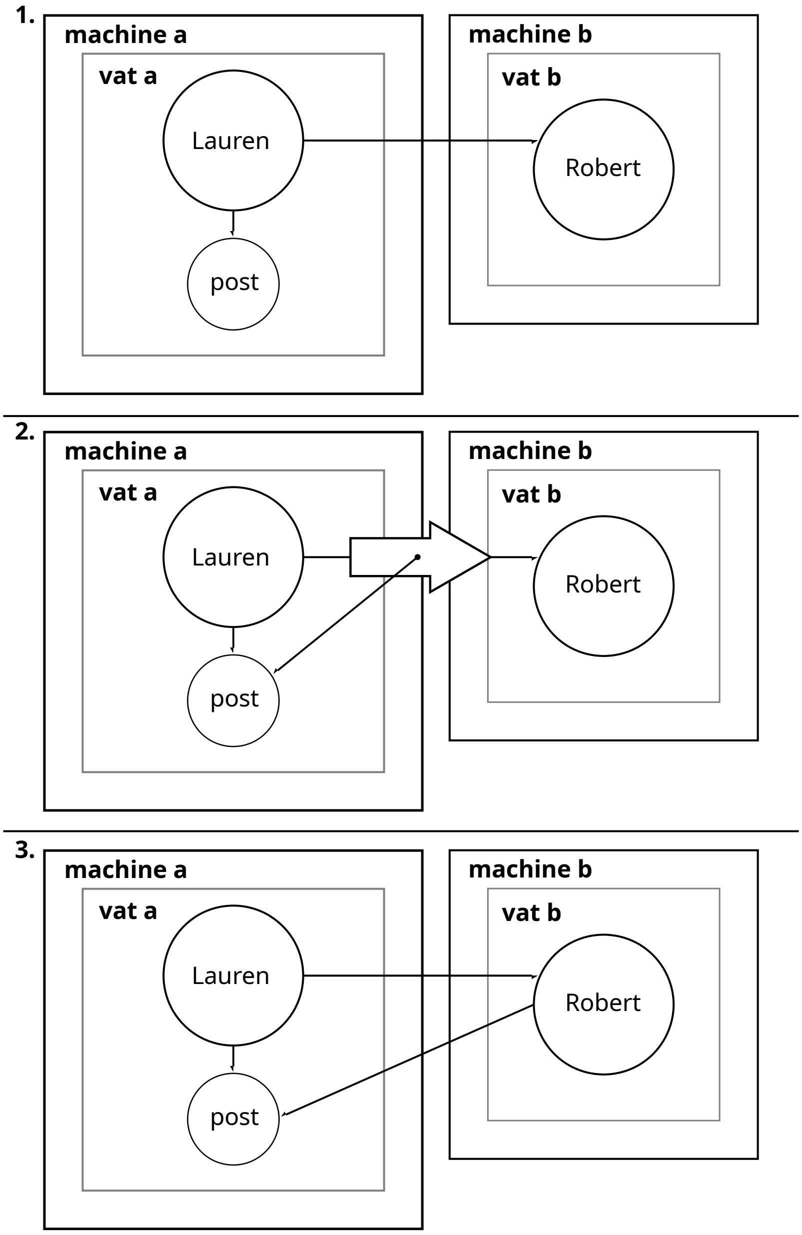

Figure 3.3.1:

```
REPL [1]> 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 [1]> $ day-in-park-editor 'update
_____        . #:body "It was a fine day to take a walk..."
```

3.3.1.1. 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 final header for this module will look like so:

```
define-module : goblins-blog
  . #:use-module : goblins
  . #:use-module : goblins actor-lib methods
  . #:use-module : ice-9 match
  . #:use-module : srfi srfi-9
  . #:use-module : srfi srfi-9 gnu
  . #:use-module : simple-sealers
  . #:use-module : method-cell
  . #:export (spawn-post-and-editor spawn-blog-and-admin
              new-spawn-blog-and-admin spawn-adminable-post-and-editor
              ^logger spawn-logged-revocable-proxy-pair
              spawn-post-guest-editor-and-reviewer
              display-post-content display-blog-header
              display-post display-blog)
```

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.[18] (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*`.

3.3.1.2. Analysis

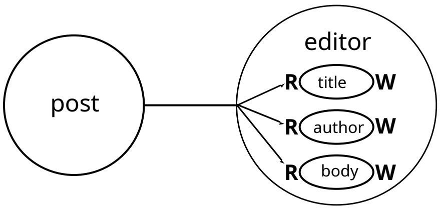

Figure 3.3.2:

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 [Capability security as ordinary programming] demonstrated that an identity-centric authority model is unsafe due to ambient authority and confused deputy problems. The solution there of a capability security as ordinary argument passing extends into Goblins in a natural way. Since Goblins' object 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.

18 Guile's `define` does not support keyword arguments, but `define*` does. Keyword arguments are simply those introduced with the `#:key` syntax. They can be supplied much as they are defined, using `#:name value`. For example:

```
REPL> define* (hello #:key [who "world"])
_____   display : string-append "Hello, " who "\n"
REPL> hello
;; => Hello, world
REPL> hello #:who "Todd"
;; => Hello, Todd
```

Although this is a common feature of Schemes, it is not actually standard. However, the `cons` syntax, also introduced here, is; for more on that, see the Scheme Primer section "Lists and cons".

This example makes `post` 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 less powerful object can achieve most of its functionality by attenuating a more powerful object.

3.3.2. 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 [1]> 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`, and which 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 [1]> $ 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 [1]> $ maple-valley-admin 'add-post day-in-park-post
REPL [1]> $ maple-valley-blog 'get-posts
; => (#<local-object ^post>)
```

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

```
REPL [1]> 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 [1]> 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 [1]> $ 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 [1]> $ 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 [1]> 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...
```

3.3.2.1. 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
```

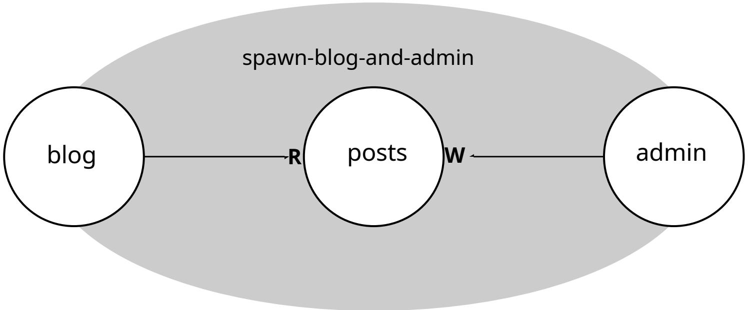

Figure 3.3.3:

Here you 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`.

3.3.2.2. Analysis

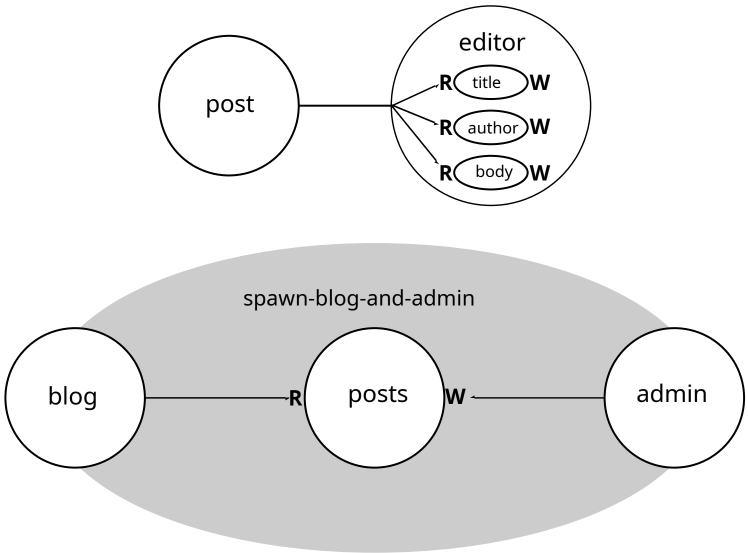

Figure 3.3.4:

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 data structure that the two other objects both use to coordinate reading current information (with `blog`) and writing current information (with `admin`).

3.3.3. 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 [1]> 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 [1]> $ 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.

As a contrived example, pretend that a malicious actor wants to ruin Lauren's reputation and has somehow gotten access to a few pieces of the blog infrastructure, but only has access to the `add-post` method of `maple-valley-admin`. This is the best they can do:

```
REPL [1]> define-values (mallet-seal mallet-unseal mallet-sealed?)
            make-sealer-triplet
REPL [1]> define-values (shmaple-shmalley-post shmaple-smalley-editor)
            spawn-post-and-editor-internal
              . mallet-seal
              . #:title "Maple Valley More Like Shmaple Shmalley"
              . #:author "Lauren Ipsdale"
              . #:body "I hate Maple Valley..."
REPL [1]> <- spoofed-maple-valley-admin 'add-post shmaple-shmalley-post
;;     => error: Self-proof not for this post ...
```

As you can see, while Mallet is able to make their own sealers and unsealers, these do not correspond to the admin object's sealer and unsealer. Because of this, they cannot be used for anything dangerous. In other words: Mallet does not have it, so they cannot use it!

3.3.3.1. 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, the example 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, imagine a scenario where you are sealing lunchtime meals using sealers and unsealers. Your rival, who wishes to sabotage you, does the same:

```
REPL [1]> define-values (our-lunch-seal our-lunch-unseal our-can?)
            make-sealer-triplet
REPL [1]> define-values (rival-lunch-seal rival-lunch-unseal rival-can?)
            make-sealer-triplet
```

You give your customer the unsealer, the delivery driver the brand predicate, and keep the sealer privately to yourself.

The contents of sealed cans are private:

```
REPL [1]> our-lunch-seal 'fried-rice
; => #<seal>
```

Your customer wants some chickpea salad, so seal some for them:

```
REPL [1]> define chickpea-lunch
            our-lunch-seal 'chickpea-salad
```

Thankfully your truck driver is able to check that the food they are to deliver really is from you. (You have a reputation to uphold!)

```
REPL [1]> our-can? chickpea-lunch
; => #t (true)
REPL [1]> our-can?
            rival-lunch-seal 'melted-ice-cream
; => #f
```

And the customer is able to open it just fine:

```
REPL [1]> our-lunch-unseal chickpea-lunch
; => 'chickpea-salad
```

Whew!

3.3.3.2. Implementation

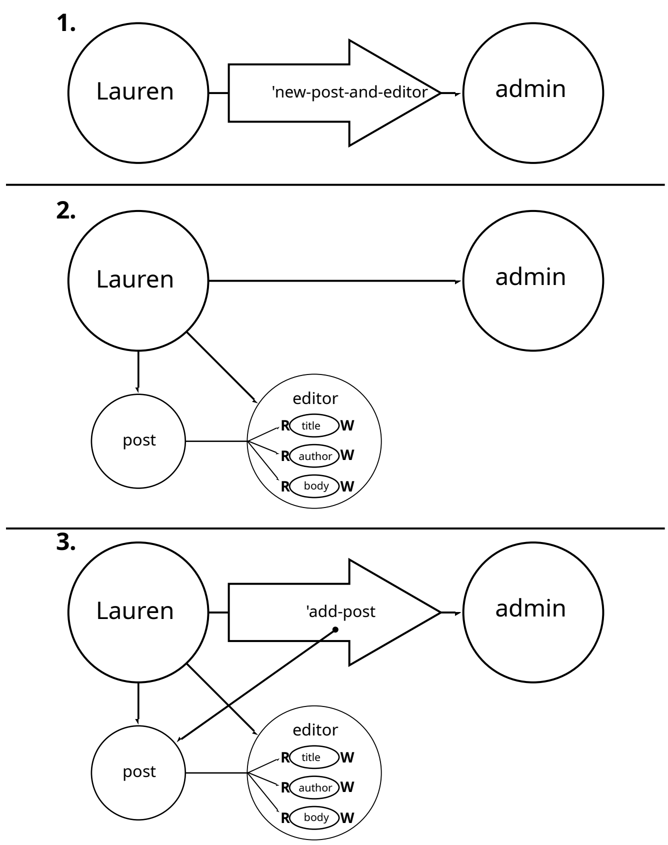

Figure 3.3.5:

This will require re-architecting the post/editor and blog/admin tooling to enable this new functionality, adding support for sealers and a few new methods.

The new version of post/editor spawning will no longer be used directly by users, so 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 the 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.
- It adds 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*`.

The blog/admin spawning code also needs to be updated so that it will be able to cooperate with the new post/editor code:

```
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, 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
```

There are 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.

Lauren would create a new `maple-valley-admin` and `maple-valley-blog` with this procedure as before:

```
REPL [1]> define-values (maple-valley-blog maple-valley-admin)
_________   new-spawn-blog-and-admin "Maple Valley News"
```

Finally, this last bit is some convenience for consistency in the 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
```

3.3.3.3. 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 this is a system which aims to avoid the security problems associated with traditional access control list and related identity-centric authority systems.

Instead, there is 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.

3.3.4. 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 [1]> 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 [1]> 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 [1]> <- admin-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 [1]> $ 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 [1]> $ 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 [1]> $ roberts-admin-revoked? 'set #f
```

3.3.4.1. 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
```

```
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.

3.3.4.2. Analysis

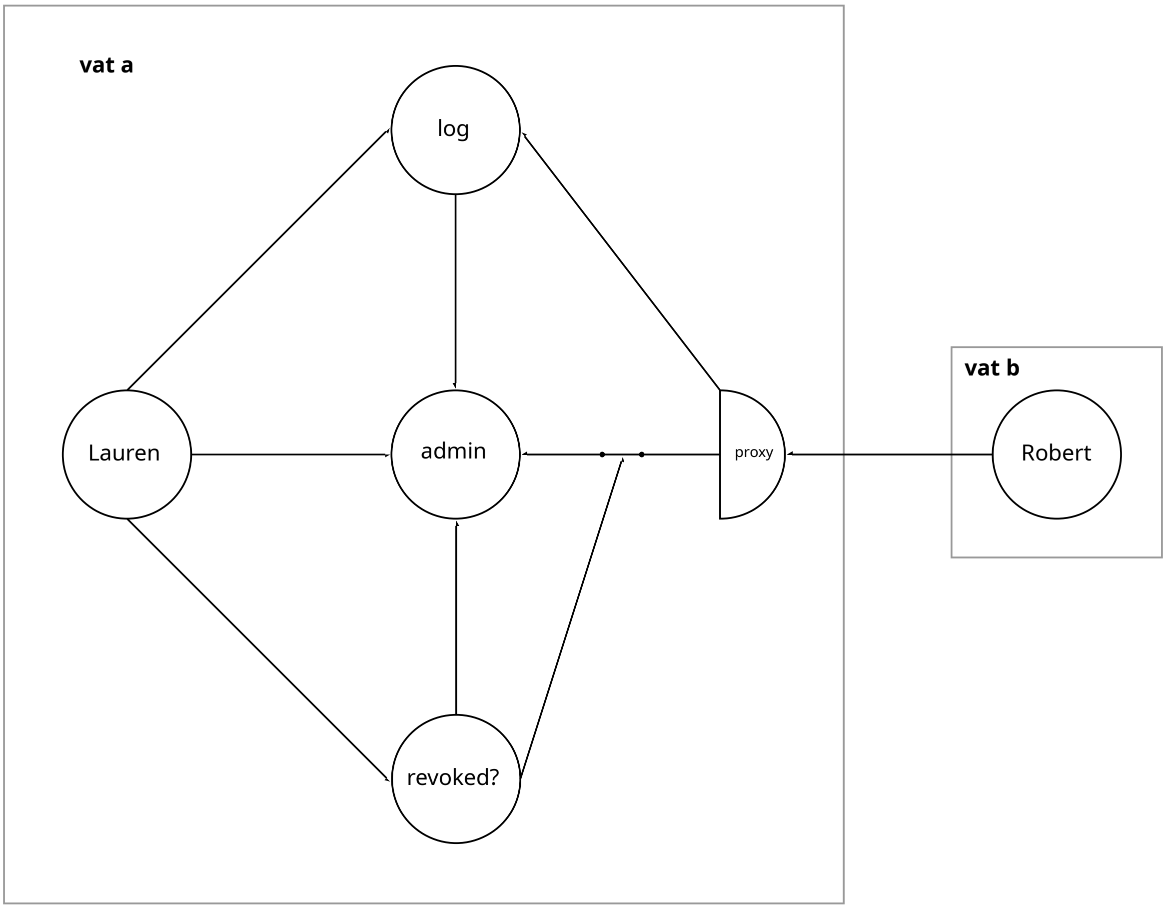

Figure 3.3.6:

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.

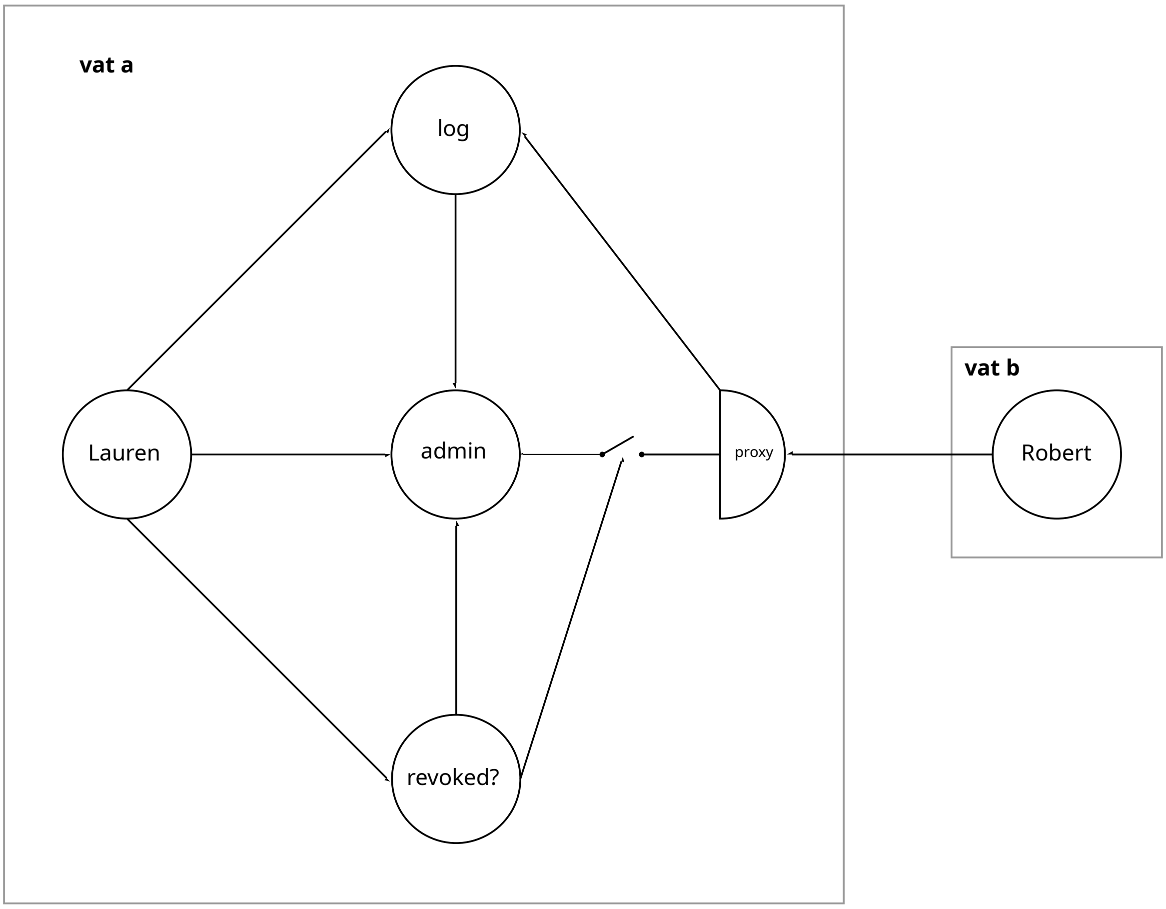

Figure 3.3.7:

Nothing is preventing Robert from sharing `admin-for-robert` with anyone else (including programs or services Robert runs), but Lauren will hold Robert accountable for any actions taken with the `admin-for-robert` capability. This is a feature, and it is extended in the next section.

3.3.5. Guest post with review

Some time has passed and Maple Valley News is doing well. Robert and Lauren have been knocking out a lot of well celebrated articles covering their community. Lauren is busy figuring out next steps for the newspaper, but Robert is exhausted and needs to go on the vacation he has long promised his family they would take. But Robert has an idea for a guest post article that could be published in his absence without having to interrupt Lauren.

Robert has a friend who works at the local school, Maple Valley Elementary, and has told Robert about how a young student named Matilda Sample won a distinguished prize in the regional science fair, assisted with the mentorship of her science teacher Mx. Beaker. Robert thinks this would be a great idea for a story. Robert asks if Matilda would be willing to write a story about her experience and whether Mx. Beaker would be willing to review and determine if and when the article would be good enough to publish.

Everyone agrees, so Robert sets everything up. Robert runs the following:

```
;; Robert's interactions
REPL [1]> define-values (science-fair-post science-fair-editor
                         science-fair-reviewer)
            spawn-post-guest-editor-and-reviewer "Matilda Sample" admin-for-
robert
```

Robert now sends out a message to Matilda and another to Mx. Beaker with the capabilities they will need:

- `science-fair-post` is given to both Matilda and Mx. Beaker and allows either of them to read the current state of the post.
- `science-fair-editor` is given to Matilda only; this allows Matilda to edit and author the post. This capability only allows Matilda to change the title and body, but *not* the author (which Robert has already set to "Matilda Sample"). However, this capability *does not* give Matilda the authority to publish the post.
- `science-fair-reviewer` is given to Mx. Beaker; this allows Mx. Beaker to approve and publish the post (but also will prevent future edits in the process). However, this capability *does not* give Mx. Beaker the authority to modify the post.

Matilda begins writing the post:

```
;; Matilda's interactions
REPL [1]> <- science-fair-editor 'set-body
              . "My name is Matilda and I am twelve. I won the science fair..."
```

Matilda asks Mx. Beaker if it's good enough to publish. Mx. Beaker tells Matilda, not yet! The post needs a title, and Matilda's teacher explains how to make the post tell a more engaging and personal narrative.

Matilda updates the title and rewrites the body:

```
;; Matilda's interactions
REPL [1]> <- science-fair-editor 'set-title
              . "Winning the Middle School Science Fair: A Personal Account"
REPL [1]> <- science-fair-editor 'set-body
              . "At twelve years old, winning the local science fair has
been..."
```

After another prompt for review, Mx. Beaker decides that the post now looks great and will be a great representation of both Matilda and the school. Feeling proud of their student, Mx. Beaker presses approve:

```
;; Teacher's interactions
REPL [1]> <- science-fair-reviewer 'approve
```

And the post goes live!

Readers of the blog will see the new post, and will be able to share it widely:

```
;; Widely runnable (by blog readers and those they share it with)
REPL [1]> display-blog maple-valley-blog
```

Robert, still on vacation, receives a message from Lauren. "Hey, I just saw that blogpost go live! It looks great! But I see in the log that you posted it… didn't you promise you weren't going to work while you left on vacation?"

Robert smiles and types up a response. "Funny thing that… nice things can happen that serve multiple peoples' interests, and if you think far ahead enough, sometimes when you aren't even around…"

3.3.5.1. Implementation

Robert's clever solution is custom code. He was able to write it without even having to change anything about how the core blogging code worked:

```
;;; 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-adminable-post-and-editor
      . blog-admin
      . #: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
```

This uses patterns already shown. The code above has an encapsulated `post` and `editor` but only exports `post` directly. The post is already configured at `spawn-post-and-editor` time with the relevant author. `restricted-editor` is configured to allow changing the title and the body, but not the author.

Once `reviewer`'s `'approve` method is called, the encapsulated `blog-admin` will be invoked to add the post to the blog. This also flips the encapsulated `submitted-already?` cell to true. At this point, `reviewer` and `restricted-editor` will be revoked, throwing an error if someone tries to use them.

3.3.5.2. Analysis

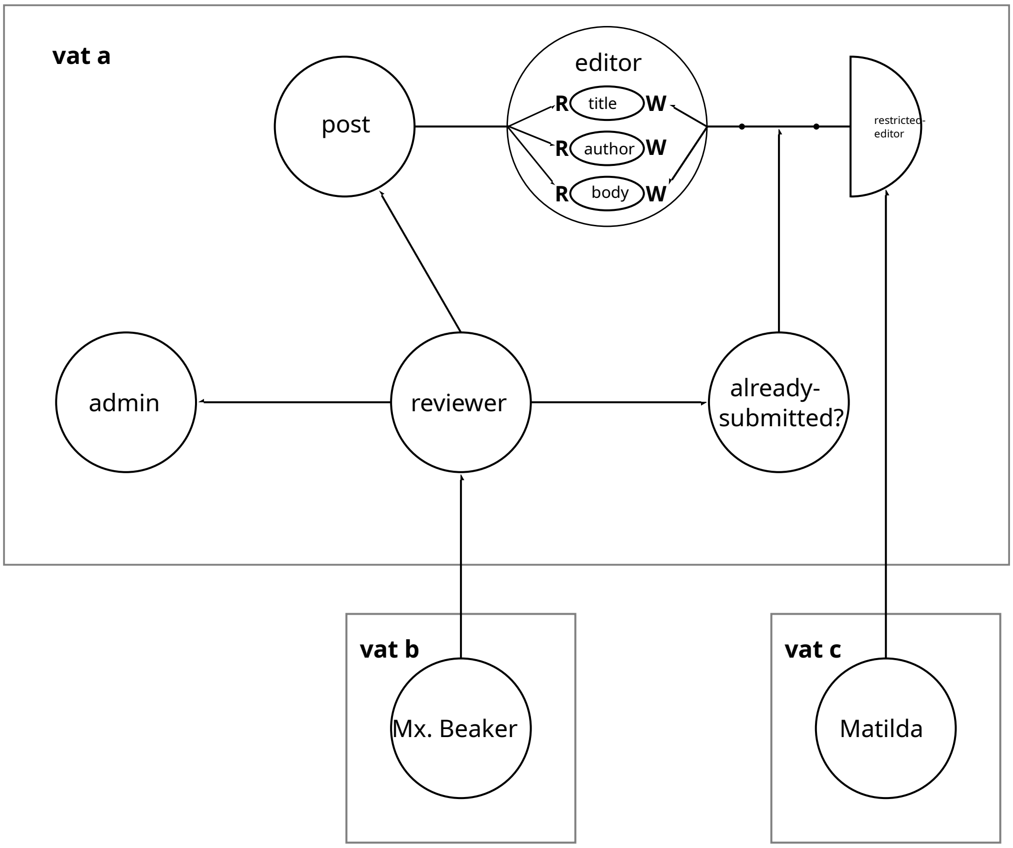

Figure 3.3.8:

While variants of all the techniques shown in this example have already been shown, the astounding thing is that the way they are arranged permits cooperation between multiple parties:

- Lauren wishes to hold Robert responsible for any updates to the blog Robert makes. Since Robert uses his admin capability, he is still held accountable for whatever actions are taken.
- Robert wishes to have interesting new content added to the blog while both he and Lauren are unavailable to actively participate. By bringing multiple stakeholders to the table, he feels confident that quality both he and Lauren would feel comfortable with will be maintained.
- Matilda wants to be able to talk about her experiences, and wants to be able to tell them in her own words and not be misrepresented. She is willing to receive mentorship from her teacher and apply this feedback to produce an improved article, even though she wants to write the article herself.
- Mx. Beaker wants a quality article that reflects well on their school, their student, and

themselves. However, Mx. Beaker can only approve the post, meaning that they must convince Matilda of any changes they would like made.

- Robert is assured that neither Mx. Beaker nor Lauren can post the article on the blog falsely claiming authorship from someone else.
- When Lauren and Robert return from being busy, they will both still be able to use their admin capabilities to edit the post should they feel it appropriate (though Lauren will still hold Robert accountable for his changes).

But the most astounding thing of all: this entire arrangement was possible without changing any of the pre-existing blog code. Robert was able to encode an arrangement that kept everyone's interests in play, without having to even be present!

3.3.6. Lessons learned

This tutorial has skipped over some important steps intentionally: there is no demonstration of setting up the network connections between parties, no demonstration of how to produce capability references which can be passed along offline, and no demonstration of how these posts might be persisted to long-term storage or upgraded.

Nonetheless, it is has shown some powerful things:

- Distributed objects defined by *behavior* and bound together through *capabilities* are sufficient to represent sophisticated and useful social interactions between multiple parties.
- The authorization mechanism relies on capability references and follows the "if you don't have it, you can't use it" philosophy. Sharing access remains as simple as reference passing. Everything is understandable as ordinary code.
- Despite the fact that the authorization mechanism itself is ambivalent about the identity of its participants, attribution of actions can be encoded into the system. Combined with a revocation mechanism, this permits accountability. This example also added broader group-style access to administer certain objects. All this without needing an access control list mechanism or the inherent ambient authority and confused deputy risks associated with such an approach.
- Goblins is able to encode rich, multi-stakeholder arrangements that benefit everyone. The guest post with a review example demonstrated that special use cases like this can occur layered on top of an existing system rather than requiring a messy rewrite of existing behavior.

3.4. Spritely Goblins as a society of networked objects

The relationship between Spritely Goblins' abstracted distributed object layers can be understood visually. Consider the following relationship graph representing communicating objects:

```
.----------------------------------.         .----------------------.
|            Machine 1             |         |       Machine 2      |
|            =========             |         |       =========      |
|                                  |         |                      |
| .--------------.  .---------.   ,-.       ,-.                     |
| |    Vat A     |  |  Vat B  |  _|  \______|  \_   .------------.  |
| |  ,-----.     |  |  ,---.  | | |  /      |  / |  |    Vat C   |  |
| | ( Alice )-------->( Bob )---' `-'       `-'  |  |  ,-----.   |  |
| |  `-----'     |  |  `---'  |    |         |   '--->( Carol )  |  |
| |      \       |  |    ^    |    |         |      |  `-----'   |  |
```

```
| |         V      |  `----|----'    |         |      |            |  |
| |    ,------.  |       |        ,-.       ,-.     |            |  |
| |   ( Alfred ) |       |_______/  |______/  |__   |  ,------.  |  |
| |    `------'  |               \  |      \  | '----( Carlos ) |  |
| |              |                `-'       `-'     |  `------'  |  |
| '--------------'                 |         |      |            |  |
|                                  |         |      '------------'  |
'----------------------------------'         '----------------------'
```

In the above diagram:

- Two machines (`Machine 1` and `Machine 2`) are running separately from each other, but connected to each other over the network via OCapN and CapTP.
- `Vat A` and `Vat B` are event loops which live on `Machine 1`, and `Vat C` is an event loop which lives on `Machine 2`.
- The individual objects (represented by circles) live in *vats*, aka event loops which contain objects. `Alice` and `Alfred` live in `Vat A`, `Bob` lives in `Vat B`, and `Carol` and `Carlos` live on `Vat C`. (While these objects have human-like names, they're just Goblins objects.)
- The arrows between the objects represent references these objects have to each other. `Alice` has references to both `Alfred` and `Bob`. `Bob` has a reference to `Carol`. `Carlos` has a reference to `Bob`.
- Two objects which are in the same vat are considered *near* each other, and thus can invoke each other synchronously, whereas any objects not in the same vat are considered *far* from each other. Any objects can invoke each other by asynchronous message passing as long as they have a reference to each other.
- Not pictured: each vat has an *actormap*, an underlying *transactional* heap used for object communication. This is what permits *transactionality* and time travel. (*Actormaps* can also be used independently of vats for certain categories of applications.)

Another way to think about this is via the following abstraction nesting dolls:

```
(machine (vat (actormap {refr: object-behavior})))
```

- **Machines**, which are computers on the network, or more realistically, operating system processes, which contain…
- **Vats**, which are communicating event loops, which contain…
- **Actormaps**, transactional heaps, which contain…
- A mapping of **References** to **Object Behavior**.

3.5. The vat model of computation

Goblins follows what is called the *vat model* of computation. A *vat* is 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), as well as messages between them and from other *vats*.

```

           Internal Vat Schematics
          =========================

        stack           heap
        ( $ )        ( actormap )
```

```
      .-------.---------------------. -.
      |       |                     |  |
      |       |    ,-.              |  |
      |       |   (obj)       ,-.   |  |
      |       |    `-'       (obj)  |  |
      |   ___ |               `-'   |  |
      | |___) |         ,-.         |  |- actormap
      |  ___  |        (obj)        |  |  territory
      | |___) |         `-'         |  |
      |  ___  |                     |  |
      | |___) |                     |  |
      +-------'---------------------+ -'
queue |  ___    ___    ___          | -.
 (<-) | |___)  |___)  |___)         |  |- event loop
      '-----------------------------' -'  territory
```

The above schematic of a vat models its constituent parts. The actormap (the heap) holds the objects (actors) local to that vat. Each object in a given vat's actormap are *near* to each other. The stack holds synchronous messages, which are conventional calls; these are invocations of *near* objects. The queue holds asynchronous messages, which can either be between *near* objects, or from other vats and thus from *far* objects.

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 is expounded more later. Goblins uses 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*)[19]. 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[20]; asynchronous message passing works the same either way. Goblins uses the `<-` operator to perform asynchronous operations.

The sender of an asynchronous 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). For both programmer convenience and for network efficiency, Goblins supports *promise pipelining*: messages can be sent to promises which have not yet been resolved, and will be forwarded to the target once the promise resolves.[21]

While the *vat model* of computation is not new[22], Goblins brings some novel contributions to the

19 In the actor model, objects called *actors* pass fully asynchronous messages to perform computation. An *actor* in the classic model processes one incoming message at a time as defined by its current *behavior*. It may respond in one of three ways: create and receive a reference to a new *actor*; send messages to other *actors* including introducgin them to other actors it knows about; or specify a change in its behavior for the next message it receives. As there are other variants of the actor model, this core, general subset is sometimes called the *classic actor model*. Object capabilities generally and Spritely in particular build on this model.

20 One situation in which it does matter whether a far object is running on the same machine is in the case of network session failure. Whereas objects in the same process and, to a lesser extent, on the same machine will be able to rely on the ability to communicate, those connected across a network have no such guarantees should the connection between their host machines be broken. This problem can be mitigated with a variety of techniques including store-and-forward networks.

21 Those familiar with JavaScript may already know the term from JavaScript. Indeed, JavaScript promises are descended from E and Joule. However, JavaScript does not implement the full version of promises from these languages; notably, they do not support promise pipelining. Goblins' promises, however, do.

22 The *vat model* 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. All of these projects are the

table in terms of *transactionality* and time-travel debugging, enhancing an already powerful distributed programming paradigm.

3.6. Turns are cheap transactions

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 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).

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 occurred.[23]

3.7. 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,[24] allowing programmers to debug a program in the state of an error when it occured.

3.8. 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
```

brainchildren of Mark Miller.

23 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.

24 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.

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 the 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 it 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 track of cognitively, and so the system should serialize the process. Asking the objects to self-describe or manipulate the underlying database could also be dangerous, as objects could claim to have authority that they do not. For example, in the game, player-built objects should not be able to claim or dispense in-game currency or grant themselves powers which they did not originally have on restoration.

Spritely Goblins' solution[25] 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.[26,27] Goblins' serializer starts with root objects and calls a special serializer method on each object, asking each object for its self-portrait. 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.[28]

Since walking the entire object graph is expensive, serialization can take advantage of reading turn-transaction-delta information to only serialize objects which have changed, making it 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 Goblins is built out it will collect many upgrade patterns into a common library.

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

25 An alternative 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 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.

26 The ideas for Spritely's serialization/upgrade mechanism stem 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).

27 This system was originally a separated mechanism called *Aurie*, symbolized by a character made out of fire which was continuously extinguished and re-awakened like a phoenix. However 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 was folded into Goblins itself.

28 This is a common ocap pattern called *rights amplification*, explored in Group-style editing.

3.9. Distributed behavior and why we need it

In general so far when this paper has spoken of distributed objects, it has 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 asked 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.[29] Alisha and Ben may have their own local representations of Carol with their own local behavior and state, particular to the interests, needs, or knowledge of Carol as she is known locally to each networked participant.

The Unum Pattern is a conceptual framework that encompasses the idea of a distributed abstract 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;[30] you might know that the teacup has a secret note sealed inside it and someone else 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*.[31]

29 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* (*ACL*) 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.

30 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 Spritely must develop patterns that treat identity with care.

31 The above explanation is modified directly from Chip Morningstar'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. He also generously agreed to let us use the unum diagrams above.

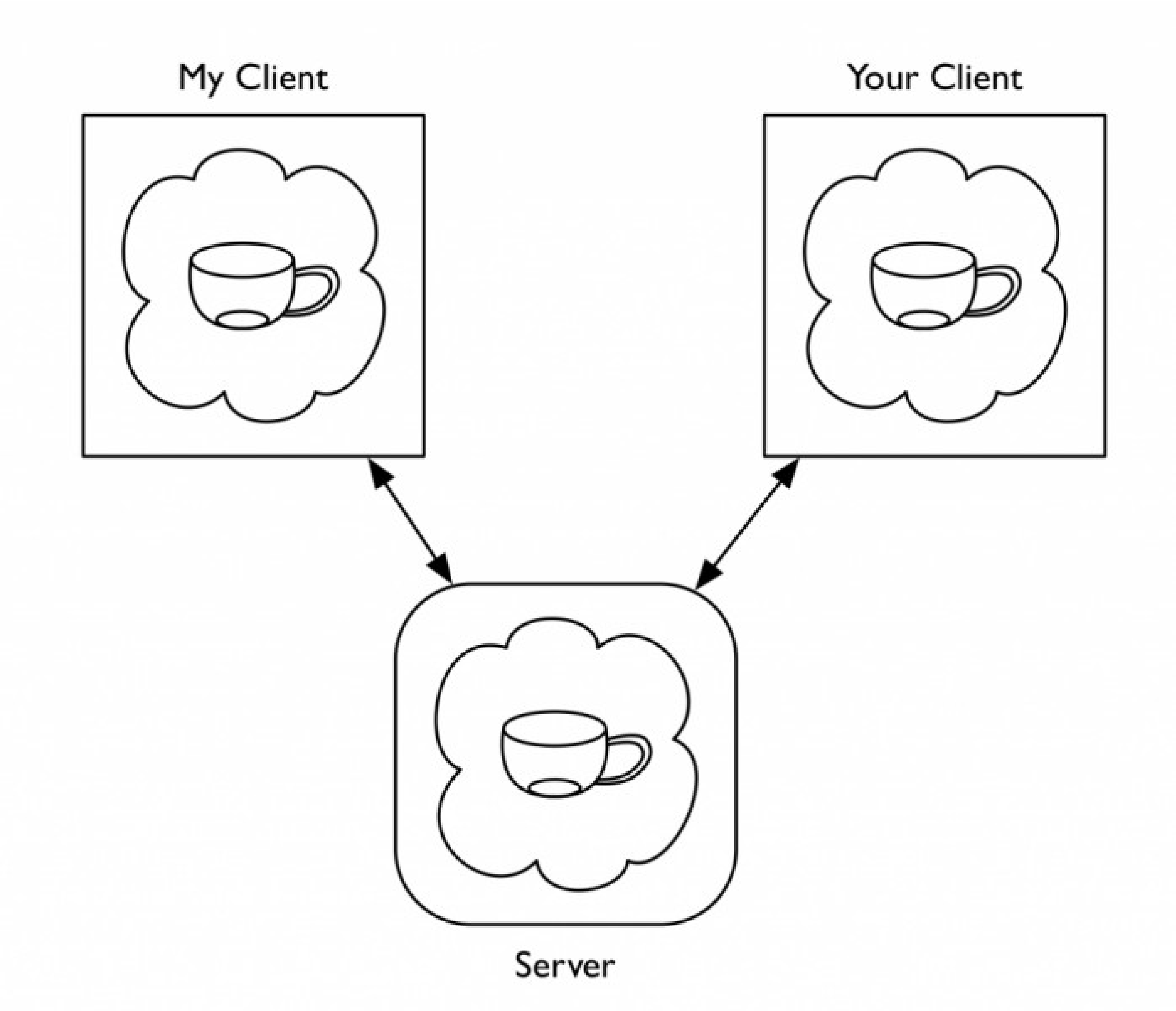

Figure 3.9.1:

Presences in Goblins typically correspond to Goblins objects.[32] 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.

32 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.

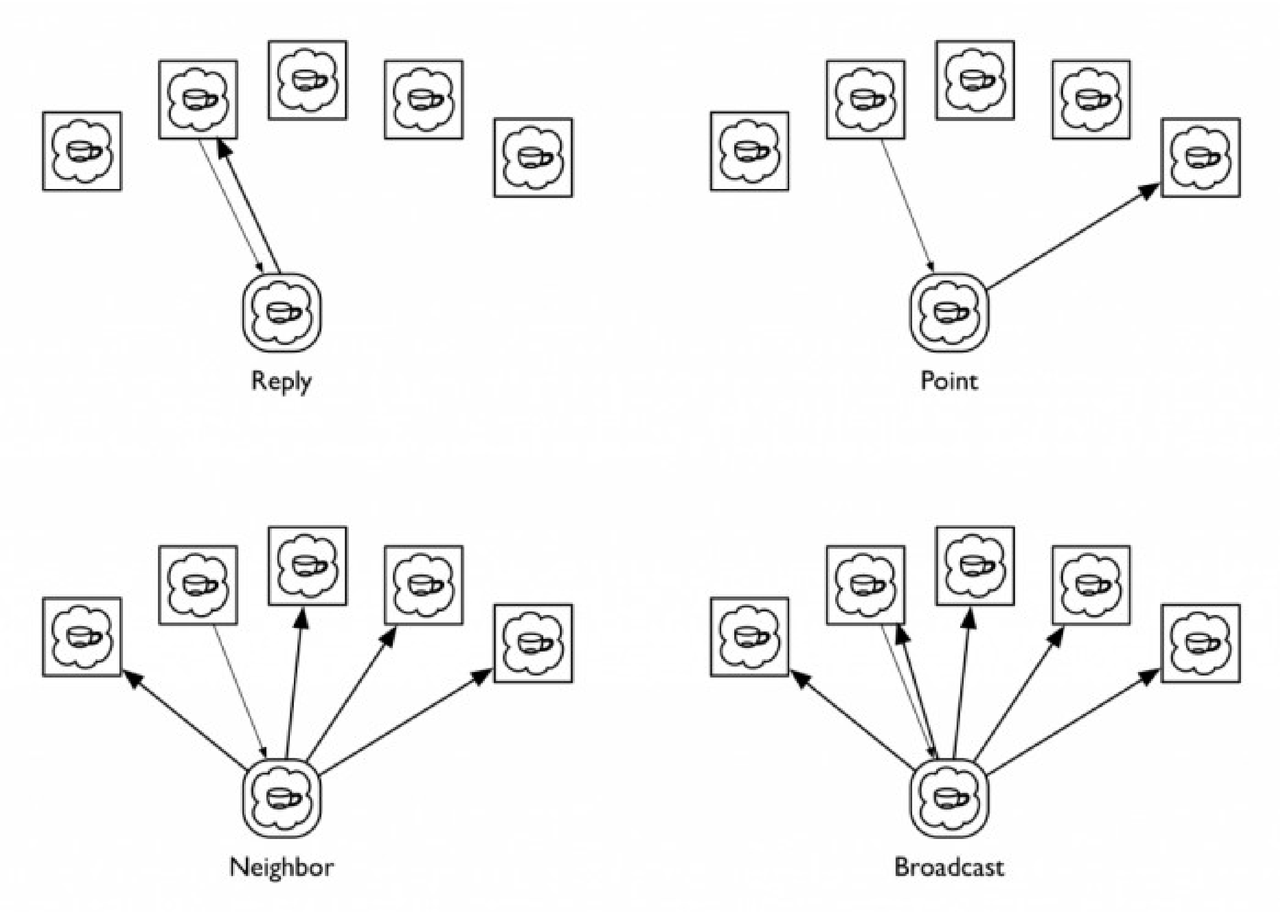

Figure 3.9.2:

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

Here, in brief, is a discussion of *OCapN* (the *Object Capability Network*), which Spritely Goblins implements. Spritely is leading the effort to define and produce a draft specification of *OCapN* to kickoff a standardization process, the progress of which you can follow in its GitHub repository. What *OCapN* provides is a set of layered abstractions so that very little code needs to be aware of "where" objects live for asynchronous programming, fully capable of functioning with no central authorities, even on *peer-to-peer* networks with the default assumption of hostile participants. While *OCapN* is already supported by Spritely Goblins, the protocol is general and could be broadly implemented across programming languages, providing interoperable networked cooperation.

The layers of OCapN are:

- **CapTP:** The *Capability Transport Protocol* (also known as *CapTP*) provides a distributed, secure networked object programming abstraction. *CapTP* provides familiar message passing patterns with no distinction between asynchronous programming against local vs remote objects and features:
    - **Distributed garbage collection:** Servers can cooperate to free resources which are no

longer needed.

  - **Promise pipelining:** Massive parallelization and network optimization. Provides convenience of sequential programming without round trips.

- **Netlayers:** CapTP sits on top of the *netlayers* abstract interface, which allows for establishing secure connections between two parties. The *netlayers* abstraction provides:

  - **Transport protocol agnosticism:** Multiple types of netlayers are supported. Fully *peer-to-peer* networks such as Tor Onion Services, I2P, and libp2p can work alongside more contemporary networks such as DNS + TLS. Even encrypted *sneakernets* are possible.

  - **Temporal connection abstraction:** Both live sessions for high-performance socketed connections and high-delay, intermittently offline/online store-and-forward systems are supported.

- **URI structure and certificates:** Entrance to the network must be bootstrapped and object locations identified; a unification of URI schemes provides the information an OCapN-aware language/library can use to engage connectivity. Certificates provide similar functionality but with different tradeoffs: less simplicity in sharing, but also less vulnerability to leakage.

5. Application safety, library safety, and beyond

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

While all the examples in this paper follow object capability security discipline, this paper has hand-waved past one critical detail. Even if Goblins follows object capability security discipline, Goblins is implemented as a library. Goblins can provide capability security properties in the network through *OCapN*, but it needs more:

- It must trust the security environment that Goblins itself runs on, so that Goblins-enabled programs will not be subverted in the security properties they are designed to provide. In other words, it needs a **trusted computing base**.

- It must also be possible to preserve the safety of code which runs *on top of Goblins* (both externally potentially malicious or buggy vulnerable code, but even preserving the safety of Goblins code, to help reduce bugs which manifest as vulnerabilities), which means it needs a **safe evaluation environment**.

There are many layers of a *trusted computing base*, and Spritely would like to provide as many as possible:

- **User experience level safety:** The end user experiences of everyday users should uphold the users' intuitions of security through the interfaces they use for their work, entertainment, social communication, and community interactions. (This is the topic of a forthcoming paper, Spritely for Secure Applications and Communities.)

- **Network cooperation level safety:** Spritely wishes to be able to cooperate with objects hosted across the network and preserve capability passing semantics at the network abstraction level. Objects should be able to cooperate with objects on another host, but another host should be able to hold no more dangerous authority over them than the capabilities which have been granted it (by the object, or by those who have delegated capabilities to it). Thankfully, Goblins is able to provide this layer through *OCapN* already, so this constitutes part of the *trusted computing base* (assuming, of course, lower components have not been subverted).
- **Library level safety:** All modules are untrusted by default. Loading a module doesn't mean it can do dangerous things. Instead of libraries being able to "reach out" and grab access to whatever dangerous operations they would like (such as accessing the filesystem, the network, etc), libraries should have to be passed explicit capabilities to do these things, not unlike passing capabilities into the invocation of a function.
- **Language level safety:** Related to the above, the language needs to uphold the security properties encode in programs, and the runtime itself should be well programmed and to have good object capability enabling semantics. Generally, for a language to be an *object capability programming language*, it should uphold the following properties: 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.
- **Application level safety:** All programs are untrusted by default. Loading a program doesn't mean it can do dangerous things. Individual applications should be sandboxed to begin with no interesting authority, and users should have the ability to launch new sandboxed applications. Access to the filesystem, network, system clocks, etc should also be capabilities passed in at this layer.[33]
- **Operating system level safety:** The operating system itself should be programmed with *object capability security* in mind. It should have a secure and auditable kernel. Access to external devices should be contained and managed on a capability level.
- **Hardware level safety:** The hardware itself should not be a path to violating the integrity of the system, as free of side-channel attacks as possible, tamper-resistant, auditable and controllable by the end user, and understandable with well published specifications.
- **Supply chain level safety:** Users should be able to be sure that hardware produced matches the hardware security specifications laid out, that the production facilities are auditable, and that backdoors are not inserted.
- **Cryptographic level safety:** There should be fundamental cryptographic operations which have understandable abstractions.
- **Physics and mathematics level safety:** It should be certain that the physics and the mathematics of the universe actually function in the manner described so that all of these abstractions are possible.[34]

33 Sandboxing alone is insufficient. Running in an enclosed environment where all available capabilities are defined at launch time is insufficient; this will result in too narrowly available a range of capabilities, and users will drive a sledgehammer through the walls by handing too-large of a bundle of capabilities by default. Instead, operating systems must provide the ability to "pass in" capabilities as a system is run, not only at initialization time.

34 If we are living in a simulation, we ask that those running simulation politely not tamper with the abstraction

This is a tall order (especially that last one). Listing these out can make the process of building a fully secure system feel like an impossible task. Thankfully, things are better than they appear: while layers lower on the stack are able to subvert the integrity of layers higher on the stack, at any layer of operation users benefit from protection. For example, if a user is running a web browser in what is considered to be a generally insecure operating system, if the execution of untrusted code is contrained from accessing the user's file system, the user is still protected from some levels of vulnerability.

Spritely, aiming to provide a *trusted computing base* which users can rely on, is interested in secure implementations of every one of these layers. However, for the purpose of upholding Goblins' abstractions most especially, the most obvious layer of importance is on the *library level safety* and *language level safety* layers. To this end, the choice of Guile for this task is not a coincidence: while more work needs to be done, Guile has the right fundamental operations of sandboxed evaluation which are needed to build a secure environment.[35] The demonstration of such an *object capability programming language* with Goblins running on top of it will be the focus of a future Spritely Institute paper.

6. Portable encrypted storage

> Every seller of cloud storage services will tell you that their service is “secure”. But what they mean by that is something fundamentally different from what we mean. What they mean by “secure” is that after you’ve given them the power to read and modify your data, they try really hard not to let this power be abused. This turns out to be difficult! Bugs, misconfigurations, or operator error can accidentally expose your data to another customer or to the public, or can corrupt your data. Criminals routinely gain illicit access to corporate servers. Even more insidious is the fact that the employees themselves sometimes violate customer privacy out of carelessness, avarice, or mere curiosity. The most conscientious of these service providers spend considerable effort and expense trying to mitigate these risks.
>
> What we mean by “security” is something different. *The service provider never has the ability to read or modify your data in the first place: never.*
>
> — The Tahoe-LAFS manual on "provider-independent security"

How does one keep information alive even when computers drop from the network? Is there a way to keep information alive and not beholden to the liveness of a particular hosting provider without sacrificing the privacy and security of users? Can robust and private data storage be achieved in a way that upholds the same level of capability security properties demonstrated in this paper so far?

Security as relationships between objects provided an example of implementing a blog purely in terms of behavior. It handwaved past several details, mostly notably how to construct OCaPN *URIs* so that live connections to blogposts can be bootstrapped from out-of-band, how to persist the running object graph to long-term storage via safe serialization, how to encode a more sophisticated markup language (eg HTML or Markdown) to allow for rich document formatting, or any example of embedding (potentially large) static media within said documents.

Nonetheless, these blogposts resemble contemporary blogs served over HTTP in the following way:

barriers we have come to rely on unless we are to be given access to the parent environment in which our simulation runs.

35 It should be seen as a good sign that the previously linked sandboxed evaluations in Guile page references A Security Kernel Based on the Lambda Calculus, which has been mentioned several times throughout this paper.

access to these documents requires a live reference to a particular entity on a particular machine and is retrieved via a live interaction over a live connection. While this was useful for demonstrating that a capability system with interesting interactions can be constructed out of a *behavior-oriented* system rather than a *data-oriented* system, the blogposts themselves are fundamentally *data-oriented* and could be stored as useful portable documents.

Unfortunately, this means that an interesting document is subject to the bandwidth (and to a smaller degree, processing) availability and uptime of a single machine on the network. Hosting costs for producing a useful resource can grow, and usually fall on the shoulders of that particular resource. Should this machine no longer be available on the network, pointers to documents hosted by it can disappear. This is the general state of the web today, and is a major drive towards centralization and general bitrot of useful and historical information.

The solution to this problem is to support *portable encrypted storage*, which must fulfill the following properties:

1. Documents must be **content addressed** and **location agnostic**. In other words, the name of the particular resource is based on information stemming from the content itself rather than a particular network location. Generally this name is the hash of the corresponding document in the case of *immutable* documents and a public key (or hash thereof) in the case of *mutable* documents.
2. Both **immutable** and **mutable** documents must be supported, with the latter generally being built upon the former.
3. Documents must be **encrypted** such that the documents can be stored in locations that are oblivious to their actual contents. Only those possessing read capabilities should be able to access the documents' contents.
4. Documents should be **chunked** so that they are not vulnerable to *size-of-file attacks*.
5. Reading (and, in the case of mutable documents, writing) documents must be accessed through abstract **capabilities**.
6. Files must be *network agnostic*, meaning that they are not only *location agnostic* but agnostic even to a specific network structure. *peer-to-peer*, *client-to-server*, and *sneakernet* networks all should be supported with the same object *URIs* between them.

Many systems have been written which supply some of these properties.

IPFS is the most popular but does not provide the privacy and encryption requirements listed above, although it can be used as a foundation on which those layers are based. Spritely has written its own toy examples that satisfy all of the above requirements with Magenc and Crystal, as well as an example applied to a social network with Golem. Freenet and Tahoe LAFS were the first systems coming close to fulfilling most (but not all) of the above requirements, and laid the foundations for understanding what these requirements are and how to fulfill them. Currently Encoding for Robust Immutable Storage (ERIS) and Distributed Mutable Containers (DMC) appear to be the most promising directions for fulfilling these requirements.

This paper is primarily designed to discuss *behavior-oriented* systems rather than *data-oriented* systems; Spritely Goblins does not itself implement a solution for *portable encrypted storage* as described above, but can be a good backend for a transport by which they may be distributed, and can compose nicely with the *distributed object programming* features that Goblins does provide. However, given that the purpose of this paper is to describe essential infrastructure, it was important to demonstrate why in the long run *portable encrypted storage* will be provided. Live distributed object programming without *portable encrypted storage* is capable in the short term of building full social network systems, but secure long-lived document storage is important to the preservation of the cultural artifacts humans build together and to provide scalability friendly towards *peer-to-peer* networks without undue pressure towards centralization. Fuller expansion of this topic will be the subject of future papers.

7. Conclusions

Despite early ambitions of internet architecture, networked technologies of the last two decades have primarily been built by, and around the needs of, large and centralized institutions. Spritely's vision of re-architecting individual and community experiences on the internet requires a different approach where radically decentralized and participatory secure networked applications are the default result of programming.

Spritely Goblins meets these goals by building on established distributed programming lessons from the object capability community. Goblins further integrates these designs with theoretical approaches from the Lisp/Scheme and functional programming world, building a system that hybridizes actors and the lambda calculus. Many complicated considerations, otherwise relegated to the fringes of an explosion of domain specific languages and protocols, unify under a single model. While implemented on Scheme (for being a strong and natural fit), these ideas are written as a library general enough to be ported to most language environments with first class functions and lexical scoping.

The end result delivers great power to the user. Security analysis moves towards the intuitions of ordinary programming paradigms of reference passing. The vat model of computation synthesizes both synchronous programming against highly localized objects and asynchronous programming against objects which can live anywhere. Turn-based transactionality means that failures do not cause corruption of state in most circumstances. Time travel plus distributed debugging allows the user to more easily pin down problems and analyze them from the point of view of the system at the time where the errors occurred. An integrated safe serialization mechanism allows for objects to describe how they should be persisted using no more authority than that which they have been already granted and, upon being restored, also allows for the possibility of upgrade. And most importantly, Goblins' integration with OCapN (the Object Capability Network) and its implementation of CapTP (the Capability Transport Protocol) provides a unified distributed programming protocol with powerful features such as distributed debugging and efficient promise pipelining.

With all these features combined, Goblins provides a foundation where not only is building a future as robust as Spritely's vision requires possible, it is also comfortable and comprehensible.

8. Appendix: On the choice of Scheme

This paper (and Goblins itself) was written in the Guile implementation of Scheme, itself a *dialect* of *Lisp*. (A Racket version also exists, but is not the subject of this paper. The two versions are very similar.) This choice was made for many reasons, most notably of which was the flexibility, fast iteration time, and extensibility of the underlying language.

The usual *surface syntax* for *Scheme* and other languages like it in the *Lisp* family is "parenthetical", like so:

```
(define (greet name)
  (string-append "Hello " name "!"))
```

In a *parenthetical* representation of *symbolic expressions* (also known as *s-experessions* or *sexps*), the parentheses show where the beginning and end of each "expression" are very clearly. The parenthetical syntax is also highly minimal, but is robust enough that any (really!) programming language can be represented using this kind of Lisp "parenthetical symbolic expression" syntax.

In general, Scheme/Lisp programmers' editors do the work of managing parentheses for them, and most code is read by indentation rather than by the parenthetical grouping. In other words, Lisp programmers usually don't spend much time thinking about the parentheses at all. However, since most programming languages *don't* use syntax like this, experienced programmers sometimes find parenthetical Lisp style syntax intimidating. (In general, students totally new to programming have an easier time learning traditional Lisp syntax than seasoned programmers unfamiliar with Lisp do.)[36]

To keep *experienced programmers* from feeling intimidated, this paper uses Wisp, which looks like so:

```
define (greet name)
  string-append "Hello " name "!"
```

Compare to the previous `greet` example:

```
(define (greet name)
  (string-append "Hello " name "!"))
```

The structure of the language is the same in each of these, only the *surface syntax* has changed. Wisp derives its expression structure from indentation, but the end result is still symbolic expressions, just not expressed parenthetically. Wisp can be converted to parenthetical s-expressions, and vice versa.

9. Appendix: Lisp and Wisp

What follows is a short explanation of how Wisp relates to Lisp. The left-hand syntax is written in Wisp, whereas the right-hand code is written in standard parenthetical Scheme:

```
define (add-drawing p f)               |   (define (add-drawing p f)
  define drawer                        |     (define drawer
    make-pict-drawer p                 |       (make-pict-drawer p))
  new canvas%                          |     (new canvas%
    parent f                           |       (parent f)
    style '(border)                    |       (style '(border))
    paint-callback                     |       (paint-callback
      lambda (self dc)                 |         (lambda (self dc)
        drawer dc 0 0                  |           (drawer dc 0 0)))))
```

These are just different *surface syntax* representations of the same program. The code can mostly be read by indentation, with deeper nested indentation levels representing nested sub-expressions. Sections of code wrapped in parentheses retain their parenthetical representation as-is.

There are only a couple of tricky details to know. First, lines starting with a dot continue a previous expression:

```
render-to-file                         |   (render-to-file
```

36 The first author has found that in running workshops introducing programming, students learning programming for the first time don't find Lisp syntax intimidating once they start programming, but experienced programmers do because Lisp's syntax looks alien at first sight if you know most other languages. The author has even found that in teaching both Scheme (through Racket) and Python in parallel, many students with no programming background whatsoever (the workshops were aimed at students with a humanities background) expressed a strong preference for parenthetical Lisp syntax because of its clarity and found it easier to write and debug given appropriate editor support (Racket makes this easy with its newcomer-friendly IDE, DrRacket). For more about this phenomenon, see the talk Lisp, but Beautiful; Lisp for Everyone.

```
  . "cool-cat.png"                             |      "cool-cat.png"
  make-cat-drawing                             |      (make-cat-drawing
    . #:happy? #t                              |        #:happy? #t
    . #:size 100                               |        #:size 100))
```

Second, a colon can be used to nest a sub-expression on the same line:

```
define (get-and-save-username db)          |    (define (get-and-save-username db)
  define name : input "Name:"              |      (define name (input "Name:"))
  db-store db "username" name              |      (db-store db "username" name))
```

That's all you need to know about Wisp.

10. Appendix: Setting up Guile, Wisp, and Goblins

This appendix follows the broader document in using Guile and Guile Goblins as the example implementation. Although the process of setting up a Racket environment is different, it should be relatively similar.

The easiest way to get started with Goblins and Wisp is to use GNU Guix. Either on the Guix system or using it as a third-party package manager, the following command will get you into a Guile *REPL* using Wisp with Goblins available:

```
> guix shell --pure guile guile-goblins guile-wisp -- guile --language=wisp
```

If you don't want to use Guix, you'll need to obtain Guile, Guile Wisp, and the Guile version of Goblins separately.

10.1. Obtaining Guile

If you're using a Unix-like operating system, you probably have a package manager with Guile available in its repositories - `homebrew`, `apt`, `dnf`, etc. The recommended way to obtain Guile is through such a package manager. Failing that, you may also download Guile source and then compile it according to the instructions in its manual. Note that while Goblins may work with Guile 2, it only supports Guile 3. You'll probably need a few unmentioned dependencies to compile it:

- a C compiler toolchain, such as GCC
- GNU Autotools

With those, inside the directory for the Guile source directory, you'll run the following commands:

```
> ./configure
> make
> make install
```

After that's done you can move on to…

10.2. Obtaining Wisp

Once again, the best route to obtaining Wisp is to use your system package manager where it likely has a name such as `guile-wisp`. However, you can also download its source code and compile it. This presupposes an extant Guile installation. Inside its source directory, you should be able to run:

```
> ./configure
> make
> make install
```

And finally you get to…

10.3. Obtaining Goblins

At the moment, Goblins is only packaged for Guix and Homebrew. The only alternative is to build it manually. Following the instructions in the Guile Goblins source repository, you'll need to obtain a few dependencies:

- Guile 3.0 (handled above)
- guile-gcrypt
- guile-fibers
- Tor (runtime; optional, for networking)
- git (implicitly)

Then it should be as straightforward as running the following commands:

```
> git clone https://gitlab.com/spritely/guile-goblins
> cd guile-goblins
> ./configure
> make
> make install
```

With all of that completed, you get a *REPL* to run the examples in this document as follows:

```
> guile --language=wisp
```

11. Appendix: Using vats in files

In order to start programming with Goblins, you will need to boot up a vat somewhere. In general, Goblins uses Fibers to implement vats.

Running `spawn-vat` returns vat object. Code can be run within the context of a vat using the `,enter-vat` command at the REPL:

```
REPL> define my-vat : spawn-vat
REPL> . my-vat
;; => #<vat YtIWhAE1>
REPL> ,enter-vat my-vat
REPL [1]>
```

You can then leave the vat sub-REPL with `,q`:

```
REPL [1]> ,q
```

Inside of a file - or at the REPL if you so choose - you can instead use the `call-with-vat` procedure:

```
call-with-vat my-vat
  lambda ()
    define alice
      spawn ^greeter "Alice"
    $ alice "Bob""
```

For convenience, the `with-vat` macro can be used instead of `call-with-vat`. Here's the above example rewritten to use `with-vat`:

```
with-vat my-vat
  define alice
    spawn ^greeter "Alice"
  $ alice "Bob"
```

You can use the `vat-halt!` procedure to stop the vat, and `vat-running?` to check its status:

```
REPL> vat-halt! my-vat
REPL> vat-running? my-vat
;; => #f
```

In the future, Guile vats will automatically halt when no more references exist and thus the vat can do no more work. The Racket version of vats already work this way. In the meantime, you'll need to call `halt-vat!` manually.

12. Appendix: Utilities for rendering blog examples

```
;; Blogpost rendering utilities
;; ============================
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-post-content
    $ post 'get-content

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

13. Appendix: Implementing sealers and unsealers

There are two ways to construct sealers and unsealers; one is the "coat check" pattern,[37] the other is the language-protected dynamic type construction pattern. The latter has less complications surrounding garbage collection and leads to some real "a-ha" moments, so that will be shown here.

37 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.

This uses dynamically constructed type-records (as taken from the SRFI-9 scheme extension). In Guile, import the following:

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

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 called `<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
REPL> pos? our-pos
;; => #t
REPL> pos? 'something-else
;; => #f
```

To define these utilities so that they can be used from other modules, start a new file called `simple-sealers.w` and define the following:

```
define-module : simple-sealers
  . #:use-module : srfi srfi-9
  . #:use-module : srfi srfi-9 gnu
  . #:export (make-sealer-triplet)
```

Following from this, 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?
```

An invocation of `make-sealer-triplet` defines 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 environment 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.

Note that upholding the above requires cooperation from the language runtime to not expose tools for deconstructing arbitrary record structures. "Unfortunately", Guile does provide tools readily with `record-accessor`, `record-constructor`, and so on. However, "unfortunately" is in scare quotes because the situation is not so dire (or no more dire than the default situation in Guile, which already provides even more dangerous operations such as accessing the filesystem, the network, and so on). Application safety, library safety, and beyond describes how *language level safety* can be achieved. By not providing these record deconstructing tools to a constrained execution environment, the properties of sealers and unsealers as defined above can be upheld.

14. Appendix: Glossary

14.1. Goblins and capability terminology

- **Abstract Syntax Tree:** The abstracted programming language structure which the programming language operates at. See also *surface syntax*.
- **Actor:** A computational entity that operates only via asynchronous message passing. See also *actor model*, defined below. An *object* which only communicates via asynchronous message passing is usually considered an *actor*.
- **Actor model:** A programming paradigm where computation occurs between fully asynchronous message passing between computational entities named *actors*. An *actor* operating under the *classic actor model* processes one incoming message at a time defined by its current behavior, and in response may create new actors (obtaining their addresses in the process), send messages to other actors (including introducing them to actors this actor knows about in the process), or specify a change of behavior in regard to its next message. (To distinguish this core, original, and general subset of possible variants, the term *classic actor model* is sometimes used.)
- **Actormap:** A transactional heap mapping *object* references to a set of *behaviors*.
- **Access Control List (ACL):** In contrast to *object capability security*, an *Access Control List* system relies on identity checks against approved operations. *ACL* systems tend to exhibit *ambient authority* and *confused deputy* vulnerabilities. See the paper ACLs Don't for an explanation of the many problems inherent to access control lists.
- **Ambient authority:** A source of vulnerabilities in many programs, particularly those operating under an *ACL* model of execution; ambient authority refers to authority that is implicitly available. Programs with *ambient authority* designs tend to be vulnerable to *confused deputy* attacks and usually fail to adhere to the *principle of least authority*, increasing the attack surface of a program dramatically. Since an *object capability* environment involves explicit use of references one holds and has access to, ambient authority risks are significantly smaller.
- **Behavior:** In Goblins, the *behavior* of the object is a procedure defining how it will currently react in response to an incoming message.
- **Behavior-oriented:** In contrast to a *data-oriented* system, a *behavior-oriented* system is primarily defined in terms of the *behaviors* of its participants and their relationships (which may both change over time). The mapping of *references* to *behavior* in Goblins is handled at a low level through the *actormap* (though this is a detail mostly hidden from users of Goblins). *Behavior-oriented* and *data-oriented* systems are duals, but the primary paradigm taken dramatically shapes the structure of the underlying architecture.

- **Capability:** See *object capability*.
- **CapTP:** Originally implemented in *E*, and now (as one layer of *OCapN*) implemented in *Goblins*, *CapTP* provides abstractions for distributed object programming which allow for programming against any object on the network to have the same ease and semantics as against locally hosted objects. Also provides some neat features such as *distributed garbage collection* and *promise pipelining*.
- **Causeway:** A distributed debugger implemented in *E* and a source of inspiration to *Goblins'* distributed debugger.
- **Classic actor model:** See *actor model*.
- **Client-to-server:** A network architecture where certain participants on the network have elevated, structurally central status, named *servers*, and *clients* connect to these as lighter, structurally less significant (and generally less addressable) status. Typically eventually results in a centralized topology. Contrast with *peer-to-peer* architectures.
- **Confused deputy:** A *confused deputy* is a kind of vulnerability which arises when one entity wishes to exploit the authority another entity has but which the former entity does not. Since the general *object capability* paradigm results in "if you can't have it, you can't use it", capability systems are (generally) free from such attacks. (Careless introduction of *identity* or *rights amplification* into an object capability system can re-introduce the possibility of such vulnerabilities, a topic of a future paper.) Originally described in The Confused Deputy (or why capabilities might have been invented) by Norm Hardy.
- **Constructor:** Within *Goblins*, a *constructor* is the procedure which, upon being invoked via *spawn*, returns the initial *behavior* of the newly constructed *object*. *spawn* passes the constructor both a *bcom* capability (for changing *behavior*) and all the remaining arguments passed to *spawn*, allowing for initial behavior to be tuned to the purpose of this particular object and with other *capabilities* as references which allow the object to correctly operate.
- **Data-oriented:** In contrast to *behavior-oriented*, *data-oriented* systems involve heavy analysis of data describing the system. *Data-oriented* systems tend to involve significant amounts of judgements upon data and narrative, and thus tend to encourage *ACL* type designs (and thus also their problems), but this is not universally the case. Many CRUD web applications reading and writing from an SQL database with separate logic for interpreting or modifying that data are often *data-oriented*, and so are many systems which focus on passed messages as descriptive information of updates rather than actions to execute.
- **Dialect:** A language variant, particularly a variant of *Lisp*.
- **Distributed object programming:** A programming style where asynchronous programming may occur against a network of interconnected object relationships, reducing the conceptual overhead of building secure, highly *peer-to-peer* networked programs.
- **Distributed garbage collection:** The cooperation of multiple machines to free resources which are no longer needed. Implemented by *CapTP*. More specifically, *cyclic distributed garbage collection* if cycles crossing *machine* boundaries are collected, and *acyclic distributed garbage collection* if not.
- **E:** A major influence on the design of *Goblins*, direct successor to *Joule*, innovator of many *object capability* security patterns, first implementer of the *vat model of computation*, and the source of the first iteration of *CapTP* (and *VatTP*, as part of *Pluribus*).
- **Eval/Apply:** The heart of most programming languages: *eval* gathers up the values of arguments to an expression and *apply* performs the execution of the expression's behavior

against the evaluated arguments. Each calls the other until achieving a "fixed point" of computation (the result of the total program evaluation). Popular topic of conversation amongst *Scheme* programmers.

- **Functional programming:** Programming without side effects; freedom from time.
- **Goblins:** The very distributed object system described by this paper, and the heart of *Spritely*'s programming environment.
- **Guile:** A particular *dialect* of *Scheme*, on which *Goblins* has been implemented, and the implementation of focus in this paper.
- **Homoiconic, Homoiconicity:** The property, most notably in *Lisp*, of *surface syntax* being the same as the *abstract syntax tree*, under a datastructure which can be manipulated and used by the programmer. This permits easy language extensions in the language and makes the language much more general.
- **Identity:** An abstract signifier for some individual, resource, or concept. More mysterious a topic than it appears at surface level, and comparison of identity equality and equivalence particularly complicated. When *identity* is checked as the primary form of access control, becomes an *Access Control List*.
- **Joule:** A fully asynchronous programming language, and a direct predecessor to *E*.
- **Lambda:** Procedure abstraction, associated heavily with the Lambda Calculus, and generally considered the heart of *Scheme*. Composes *Goblins objects* both as the *constructor* and as the *behavior* of the *object*. Generally considered "The Ultimate".
- **Lisp:** A programming language family known for being highly extensible, easy to implement, and with many *dialects*. (The particular *dialect* of *Lisp* used in this paper is *Scheme*.) *Lisp* has a highly flexible abstract syntax which makes it easy to "write Lisp in Lisp", even "writing code that writes code", making language extensions or variants trivial compared to most other languages. Its *surface syntax* is typically parenthetical but is not necessarily so; see *Wisp* for the indentation-oriented surface syntax used in this paper.
- **Machine:** Within CapTP, a computer or process available on the network which contains *objects* which may be communicated with.
- **Monad:** Something Goblins tres hard to not expose you to. Arguably, an implicit one exists in *Goblins*. The meaning of this entry is left as an exercise for the reader.
- **Near/Far:** *Near* objects are co-located in the same vat, otherwise they are *far*.
- **Netlayer:** Within *OCapN*, an individual *netlayer* implements the abstract *netlayer* interface, which is a way to implement a secure channel of communication between two *machines*. Different transport layers can be used as a *netlayer*, ranging from *peer-to-peer* networks to more contemporary client-server architectures. Originally called *VatTP* in *E*'s implementation of *Pluribus*.
- **Network:** An interconnected system of *machines*. See *OCapN*.
- **OCapN (the Object Capability Network):** The combined layered abstractions of *CapTP*, *Netlayers*, and *OCapN* specific *URIs*. Combined, these allow for the implementation of a fully *peer-to-peer distributed object* programming environment with most networked protocol concerns abstracted away from the developer.
- **Object:** A term with a lot of variant meaning, but which in the case of *Goblins* means a reference to an abstract resource whose *behavior* is fully encapsulated by the runtime or network. (*Goblins* does not mean anything about class hierarchies by the word *object*,

should you be suffering from a Java PTSD induced aversion to the term).

- **Object capability (ocap):** An *object capability* based architecture (sometimes known simply as a *capability* architecture, though this term has prominent naming conflicts) is one where one's authority is based on references which one can invoke to perform computation and cause effects. Without a reference, one can't perform an action, leading to the slogan "if you don't have it, you can't use it." Used as an abstraction of security and favorable to the *principle of least authority*, though maintaining that pattern requires discipline.
- **Object capability programming language:** A programming language upholding *object capability security* properties. Generally has the following properties: 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.
- **Object graph:** The set of relationships between *objects*. In an *object capability programming language*, this is typically the set of other object references within the *behavior* of an object's scope.
- **Peer-to-peer:** A network architecture where a participant in the network has the same abstracted priority across the network routing fabric as any other participant on the network. Contrast with *client-to-server* architectures.
- **Pluribus:** The equivalent of *OCapN* in *E*. Made for a good pun: *E, Pluribus, Unum*.
- **Principle of least authority:** Design systems such that entities hold no more authority than they need in order to reduce the attack surface of an application and its subcomponents. Generally easy to pull off in *object capability* architectures, and hard to pull off in *access control list* architectures.
- **Promise:** A special type of *object* abstraction representing a computation yet to be completed, either fulfilled or broken.
- **Promise pipelining:** From a programming perspective, the ability to send messages to the objects promises will eventually designate before they are fulfilled. From a network perspective, provides an optimization allowing delivery of messages to the host *machine* queuing eventual delivery of messages once dependent promises are fulfilled, eliminating unnecessary round trips. In other words, simplifies dependency-based asynchronous plan construction. Propagates errors.
- **Quasi-functional:** *Goblins'* tricky "looks imperative from the perspective of invoking another actor and functional from the perspective of an object updating its own behavior" twist on kinda-sorta *functional programming*. Allows for powerful *transactional* programming with time-traveling features without having to expose *monad* plumbing directly to the user.
- **Racket:** Another *Scheme* which *Spritely Goblins* is also implemented on, but which is not the focus of this paper.
- **REPL:** Read Eval Print Loop, an interactive programming language shell.
- **Rights amplification:** To (mis-)quote Alan Karp, "combine two things to get access to another thing". Frequently used to provide group-like features in ocap systems. Frequently implemented using *sealers and unsealers*. Used carelessly, can accidentally re-introduce *confused deputy* vulnerabilities, but the patterns shown in this paper are free of such problems. Analysis of this phenomena is hopefully the subject of a future paper.
- **Safe serialization:** Allowing objects to describe how they should be serialized, while still

following the *object capability* motto of "if you don't have it, you can't use it". Implemented by *Goblins*, but originally in Safe Serialization Under Mutual Suspicion, which was inspired by *Uneval/Unapply*.

- **Sealers and unsealers:** The equivalent of public-key cryptography, but implemented in programming language abstractions instead. Frequently used to implement *rights amplification*.
- **Scheme:** A *lambda* heavy *dialect* of *Lisp*. The examples in this paper use a particular *Scheme*, *Guile*. Has some interesting history regarding the exploration of the *actor model*, but probably too long to cover in an already overly-verbose glossary appendix.
- **Surface syntax:** The representation of the programming language that programmers (usually humans) operate at. In *Lisp* derived languages, the *surface syntax* and *abstract syntax tree* are generally not very far apart, which is partly what makes *Lisp* languages so extensible.
- **Swingset:** Another interesting contemporary object capability programming language environment, this one layered on Javascript and produced by Agoric.
- **Sneakernet:** A network architecture where messages are delivered physically from participant to participant (perhaps even on foot, be that foot in wearing a sneaker or not).
- **Spritely:** An umbrella project to advance networked communities and decentralized networked programming abstractions.
- **Spritely Goblins:** See *Goblins*.
- **The Spritely Institute:** The nonprofit which is the fiscal steward and primary developer of *Spritely Goblins* amongst other things (and which produced this paper).
- **Syntactic sugar:** Syntax abstractions which make programming more convenient and (ideally) pleasant to read and write.
- **Transaction, transactionality:** A set of operations which are replied in a conceptually atomic manner: either all occur or none occur. Within *Goblins*, a *turn* is a *transaction* representing a delta of *behavior* changes to the *actormap* (including the introduction of new *near objects*), as well as a queue of messages to be sent. In the event of an error, the changes will not be committed and the messages will not be sent.
- **Turn:** A top-level event handled by a *Vat*, generally a message sent to a particular *object*. One unique feature of *Goblins* is that turns happen within *transactions*.
- **Unum/Presence:** The *unum* is an abstracted, conceptually and programmatically unified object, implemented by individual object *presences*.
- **Uneval/Unapply:** The abstract concept behind *safe serialization* and the inverse of *eval/apply*. Produces a program representing a graph of objects (using only the capabilities the *objects'* behavior had in scope) which can be later re-instantiated using a complimentary kind of *eval/apply*. Originally a remark from Jonathan A. Rees to Mark S. Miller leading to the Safe Serialization Under Mutual Suspicion paper. See also Rees's blogpost: Pickling, uneval, unapply.
- **URI (Universal Resource Identifier):** A type of digital identifier indicating a networked resource. *OCapN* defines several of these to designate *machines* and *distributed objects*.
- **Vat, Vat model:** An event loop which contains a set of objects, designed to be able to communicate with objects in other event loops. Objects within the vat are considered *near* to each other may perform both synchronous and asynchronous programming against each

other, whereas objects *far* from each other may only provide asynchronous programming against each other.

- **W7:** The subset of *Scheme* implemented (on top of Scheme48) for Jonathan A. Rees's PhD dissertation, A Security Kernel Based on the Lambda Calculus. Highly influential to Spritely Goblins in demonstrating clearly that a pure lexically scoped language (such as a strict subset of scheme) with no mutable toplevel scope or other sources of ambient authority is already a viable *object capability programming language*.
- **Wisp:** An indentation-sensitive surface-level *Lisp* syntax, and the one used in this paper. *Wisp* determines its expression boundaries based on whitespace. Compatible with most *Lisp* implementations. Defined under the SRFI-119 specification.

14.2. Core goblins operations

- `spawn`: The `spawn` operator in *Goblins*.
- `$`: The synchronous call-return operator in *Goblins*.
- `<-`: The asynchronous message passing operator in *Goblins*. Returns a *promise*.
- `on`: Set up a callback to be handled with the resolution of a *promise* (possibly returning its own promise related to said resolution).
- `bcom`: Pronounced "become", in *Goblins* `bcom` is the conventional name given to a *capability* relevant to a particular object which permits, and is used to, indicate the next *behavior* of the particular object. Passed by `spawn` (through Goblins' abstract kernel) to the object's *constructor*. Technically implemented as a *sealer*, allowing for a *functional* substrate for updating *behavior*.

14.3. Portable encrypted storage specific terminology

- **Portable encrypted storage:** A document storage system where files are not tied to any particular machine location (via *content addressed storage*) and are encrypted in such a way that hosting content does not provide the ability to read or modify the underlying contents of hosted files.
- **Content addressed (storage):** A document storage system where documents are named and verifiably retrieved by their content rather than by a particular network location.
- **Immutable/mutable:** Immutable objects and files do not change or update, mutable objects and files do.
- **Size-of-file attack:** Statistically determining likeliness that a file contains particular content based on its file size.
- **Chunked:** Split into consistently sized pieces to be latter reassembled, so as to avoid *size-of-file attacks* or for storage and retrieval optimizations.
- **Location agnostic:** Not tied to a particular location on the network.
- **Network agnostic:** Not tied to a particular network configuration or transport.

15. 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, Stephen Webber, Robin Templeton, Leilani Gilpin, Kate Sills, and Ludovic Courtès. (**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 Chip Morningstar who agreed to let Spritely use his diagrams of the unum pattern. Thank you to Arne Babenhauserheide, who developed the Wisp syntax for Lisp used in this paper.

16. Appendix: ChangeLog

16.1. `[2023-09-26 Tue]`

Switched Wisp pre-processing of keywords hack to use normal Wisp approach (dots at the start of line).

16.2. `[2022-07-01 Fri]`

- Incorporated feedback from Jakob L. Kreuze:
  - Add a detailed footnote explaining a bit more about why promise pipelining makes things cleaner and the risks of re-entrancy attack when coroutines provide *splitchronous* operations which appear *synchronous*
  - Various grammar nits
  - Finish explanation of Alisha and Bob bob having their own Carol representations (was a confusing intro to that section without)
  - Rework homoiconic/homoiconicity footnote, glossary entry

16.3. `[2022-06-30 Thu]`

- Incorporated feedback from Leilani Gilpin:
  - Added definitions in glossary for *peer-to-peer*, *client-to-server*, *sneakernet*, and italicize usage of those
  - Italicize usage of *transaction* and friends
  - Clarify: it's not a requirement to read the other (as of this moment: to-be-written) Spritely whitepapers to read this one
- Incorporated feedback from Robin Templeton:
  - Clarify that Unix isn't the origin of ACLs, it popularized them
  - Have a consistent comment syntax in first examples (and make it clear which ones are the `SHELL>` vs `REPL>`)
  - Capitalized Lisp/Smalltalk/Scheme
  - Explain more clearly that it's not that Goblins itself is the essential toolkit to build

something like Spritely, it's that Goblins *provides* the essential features to be a worthy toolkit

- Various garammar nitpicking
- Give the motivation for Portable encrypted storage sooner in its section
- Don't say "social network" in introduction, too captured a term
- Consistency around "**New:** comments
- Clearer introduction of what we mean by Unum in its first introduction
- Call "Lisp types" -> "Lisp dialects"
- Clarify which machine is local vs remote in `fork-motors` example

16.4. [2022-06-28 Tue]

- Many duplicate words removed

16.5. [2022-06-27 Mon]

- Add PDF export
- Explain which parts of the syntax are keywords
- Fix missing equals sign on the `fork-motors` section (a whole bunch of reviewers caught this, thanks to everyone else for being so careful ;))

16.6. [2022-06-26 Sun]

- Incorporated many grammar fixes suggested by Alan Karp
- Guix manifest updated: we're now at the point where anyone with access to the repo can easily build and verify all documents with the following:

```
SHELL> guix shell          # sets up dependencies
GUIX-SHELL> make           # builds papers, extracts all code
GUIX-SHELL> make check     # runs unit tests on examples
```

16.7. [2022-06-24 Fri]

- Add tests for blog examples
- Simplify proxy code / explanation by using `$` instead of `<-`
- Switch Matilda / the Teachers' invocations to use `<-` instead of `$` to show off these do work fully async
- Add explanation of `define-values`, `let-values`

16.8. [2022-06-23 Thu]

- Add tests for sealers/unsealers
- Fix some examples in sealers/unsealers section

16.9. `[2022-06-22 Wed]`

- Export `^cell` in `spritely-core.w` for tests
- Various bugfixes to interactive examples found while writing tests
- More information in predicate / conditional section
- Add unit tests for Taste of Goblins section
- Add makefile rule to run unit tests

16.10. `[2022-06-21 Tue]`

- Both Wisp and Scheme files are now automatically extracted when the user runs `make`
- Fix `format`, was using Racket's version
- Provide and use `method-cell.scm` for importing methods

16.11. `[2022-06-20 Mon]`

- Incorporating suggestions from Jessica Tallon
  - Fixed some renames of `eval-expr` (old name for metacirculator evaluator) to `evaluate` (thanks for the catch, Jessica Tallon!)
  - Rename some comments before lambda / procedure and procedure invocation / application examples
  - Make it clear that the methods macro does get complicated to figure out what's happening to the ellipses… it's not just you, dear reader!
- Rename `for-list` macro to `for`, keep it simpler

16.12. `[2022-06-18 Sat]`

- Add a bit more about the Y Combinator (no, not the company) to a footnote in Scheme in Scheme.
- Many tyops caught by spelcheckr
- Couple of small grammar suggestions from Baldur

16.13. `[2022-06-17 Fri]`

- Refactor introduction to language stuff, add On language and syntax choice with a mini "how to convert wisp to parenthetical syntax in your head" explainer
- Security as relationships between objects written in full!
  - Guest post with review written!
  - Lessons learned written!

16.14. `[2022-06-16 Thu]`

- Adding to Appendix: A small-ish scheme and wisp primer

    - Add explanations of `letrec` and *named lets* to Iteration and recursion
    - Show symbols earlier when showing "some more types".
    - Finish metacircular footnote.
    - Explain that `'foo` is just shorthand for `(quote foo)`, etc
    - Add On the extensibility of Scheme (and Lisps in general)
    - Preview that we'll show how to write our own `when` in the first footnote which mentions
    - Fully explain how the evaluator works in Scheme in Scheme
    - Use `format` sooner

16.15. [2022-06-15 Wed]

- Adding to Appendix: A small-ish scheme and wisp primer
    - Added Closures
    - Add a footnote to Conditionals and predicates explaining that both `cond` and `if` can be written in terms of each other. Also distinguish between `<THEN-BODY>` and `<ELSE-BODY>` in the syntactic explanation of `cond`.
    - Eliminate `newline` from examples… one less procedure to explain!
    - Explain variable arguments, `define*`, `values`
    - Added Mutation, assignment, and other kinds of side effects
    - Added Scheme in Scheme and hoo boy, it's awesome.
    - Add alist and quasiquote examples to Lists and "cons"

16.16. [2022-06-14 Tue]

- Most of Appendix: A small-ish scheme and wisp primer written.
- Correct footnote… we *do* explore rights amplification in this paper :)

16.17. [2022-06-11 Sat]

- Add Makefile, README, instructions for building HTML and extracting output

16.18. [2022-06-10 Fri]

- Finished incomplete sandboxing footnote
- Include explanations of how to build module files explicitly
- Rename section: Application safety, library safety, and beyond (formerly "Application and library safety (and beyond)")
- Some updates to Appendix: Implementing sealers and unsealers
    - Show example of `pos?` predicate in use
    - Explain necessity of language runtime participating

    - Move coat check pattern footnote to this section (which is where it was supposed to be once the appendix was added, whoops)
- Reorder some of the appendices
- Started writing Appendix: A small scheme and wisp primer

16.19. [2022-06-09 Thu]

- Added Application safety, library safety, and beyond
- Added glossary definition for `on`
- Add Portable encrypted storage section and relevant glossary terms
- Made changelog and glossary subsections into actual reified, linkable-by-fragment subsections
- Added Conclusions

16.20. [2022-06-08 Wed]

- Added Spritely Goblins as a society of networked objects
- Remove `^revoker` from revocation pair example since it isn't used (the cell is though)
- Added When schemes go awry: failure propagation through pipelines
- Fleshed out the Appendix: Glossary in no small amount of detail.

16.21. [2022-06-07 Tue]

- Added OCapN section
- `.w` wisp files now extracted from ../spritely-core.org (source of this document) via org-babel. You can view them at:
    - ../taste-of-goblins.w
    - ../goblins-blog.w
    - ../simple-sealer.w
- Cleaned up several code examples.
- Switched to new Wisp syntax adjustment (after discussion with Wisp upstream): lines starting with keywords no longer require dot to continue previous line. Change likely to be incorporated in future wisps.

16.22. [2022-04-02 Sat]

- 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` constructor, 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

17. License