Content Addressed Descriptors and Interfaces with Spritely Goblins

Christine Lemmer-Webber

April 25, 2025

> Language is a continuous reverse engineering effort between the parties involved.
>
> – Jonathan A. Rees, individual correspondence

1 Programming as a conversation

Communication always occurs in some degree of shared context between the parties involved. Human language notoriously drifts dramatically; meaning of words change regularly. Much work is done to try to protect communication from ambiguity. In computer science, this is typically done through typing or contracts (to the extent that there is any difference between them). Types and contracts restrict the scope of incoming information to what is expected within the receiving context.

Static typing takes a pre-emptive approach to preventing misbehavior through incorrect inputs, dynamic typing and run-time contracts perform late prevention of misbehavior, and linting tries to politely inform the programmer that misbehavior might be present.

This is all well and good when two parties already know ahead of time what kind of things they intend to talk about. However, in development of a program, one is often times "figuring out" what communication might be possible. Lisp enthusiasts frequently praise lisp environment's propensity towards "REPL driven development"; the programmer crafts the program by programming up until a point where it's unclear what the next piece is, experimenting live at an interactive REPL, reading documentation against datastructures and procedures, and eventually moving this code into a more permanent place of a source code file (but even here there is little distinction between the source code file and the interactive REPL). Thus, programming becomes a conversation.

Lisp users may have some of this best kind of tooling, but they are hardly alone.

Static programmers with sufficient tooling also enjoy conversational programming. Many IDE environments can explain what methods are available on an object, and even compiler errors are a kind of conversation.

What is strange about conventional programming practices is that developers tend to perform this kind of inquiring-conversation while writing programs, but once programs are written, execution and runtime tends to be very ossified (until the program is once again changed by a programmer, possibly due to new requirements or after some debugging session). However, even within these ossified systems, pattern matching and dispatch based on the response to the question of "what kind of object is this?" is the basis of all program flow. Every conditional involves a question about the state of some value, from merely "is this true or false?" to "is this number prime?" or "is this an integer or a string?"

Types, contracts, interfaces, documentation, schemas, and vocabulary are all smudged across these domains. While preventing errors is important, we should not ignore the role of communication in terms of behavior flow, nor in terms of discovery of potential interaction.

2 Pigeons and vocabulary

Do you and I mean the same thing? Lojban enthusiasts clearly solved language, creating something completely syntactically unambiguous! Except, oh no, turns out syntactic unambiguity does not mean meaning unambiguity. What's a bear? From an evolutionary standpoint, something moved from a pre-bear state to a bear-state, but evolution didn't put a pin in it… this was rough, statistical, approximate. Not to mention, what's a "dead bear"? Is it still a bear? When does it decompose into bear goo? And when does it decompose beyond that? Vocabulary thus seems to be a tug-of-war between fuzziness and crispness.

This is also a problem in our computer programs. In an exercise-tracking program, "run" might mean running some program while also referring to the human activity of "running".

The linked data community tries to improve upon this situation by having vocabulary tied to URIs, most popularly HTTP based URIs. This seems like an improvement… this should open up the number of pigeonholes we have on hand to stuff our meaning-pigeons into! Except:

- HTTP based websites go down all the time or change hands
- HTTP(S) conventionally relies on DNS and SSL Certificate Authorities, which are majorly centralized

- Shared namespaces are an opening for fraught governance and bikeshedding
- Vocabulary drift still happens… a term's meaning today might shift tomorrow as the world's context shifts around it

Some of this war is unwinnable, particularly around vocabulary drift. Identity is the sticky mudball of associations that forms around identifiers, and it's always going to be kind of messy. However, we can make the situation better. But first let's talk about "descriptors".

3 Dungeons and Descriptors

Let us imagine playing a dungeon crawler type game. We might encounter treasure which we might collect, monsters which we might fight, potions which we might collect and drink, and doors and chests which we might open (but for different reasons!). We also might want to attack monsters and drink potions, but if we attack potions and drink monsters, we might not get the results we expect, if we get any result at all (other than wasted effort and confusion).

We might start by asking something, what kind of thing are you? For instance, what's a goblin? Well… maybe it's a goblin?

```
(type-of goblin)          ; => 'goblin
```

Except maybe all Goblins are monsters, and maybe you can attack them. Okay.

```
(types-of goblin)         ; => (set 'gameobj 'goblin 'monster
                          ;         'attackable 'befriendable
                              ↪  'lookable)
```

Hm. Maybe a thing is more defined by what it does. What kind of methods does our goblin-object provide?

```
(methods goblin)     ; => (set 'attack 'look 'befriend)
```

We might use these to prevent a programmer from accidentally trying to invoke a goblin with the wrong method ahead of time (the object itself may do similar

things to protect itself). Or we might use them to decide what kinds of things we, as programmers, might be able to do with this object.

There's clearly a large overlap between the (nominative) `types-of` procedure and the `methods` procedure. Except the `types-of` also provides some additional information than just whether or not our invocation is correct: it gives us a sense of what this thing might be representationally. For instance, we might display items and monsters very differently.

In a sense, this all relates to conceptual ideas of what we think the behavior of this object is… rather than its actual behavior. Indeed, it may be more appropriate in a mutually suspicious environment, such as in a mutually suspicious peer-to-peer network to gain the idea of calling these `alleged-types-of` and `alleged-methods`… in statically compiled programs, the compiler is able to act as a central planner, ensuring that objects are what they say they are. In a mutually distributed system, objects can claim to be something they are not. For instance, that game object might call itself a treasure chest… but it might actually be a mimic! Chomp!

Here we have provided these descriptors via an unordered set. Providing them in an ordered set might be better in some cases, because certain behaviors, or suggested displays, may override others. For that matter we might think we'd want to automatically infer some descriptors from others… aren't all monsters gameobjs in this game anyway? And aren't all gameobjs lookable? But, combining both of these can lead to complicated challenges… here be dragons. Best to just focus on unordered sets for this writeup.

We still have a problem though… we still can't tell the difference between "run" (make your character run away!) and "run" (run this custom NPC script!) in this game either. What to do?

4 Introducing "Content Addressed Descriptors"

What if whether or not you and I mean the same thing can be determined based on the description of what we mean?

```
(require "cad.rkt")
```

Now let's make a content-addressed descriptor. Let's take a term proposed as an ActivityStreams extension:

```
(define sensitive-desc "\
The sensitive property (a boolean) on an object indicates \
that some users may wish to apply discretion about viewing its \
content, whether due to nudity, violence, or any other likely \
aspects that viewers may be sensitive to. This is comparable to \
what is popularly called \"NSFW\" (Not Safe For Work) or \
\"trigger warning\" in some systems. Implementations may choose \
to hide content flagged with this property by default, exposed at user
    ↪ discretion.")
```

The CAD (where CAD stands for "Content Addressed Descriptor") can be derived with the `cadify` procedure:

```
> (cadify 'sensitive sensitive-desc)
'sensitive-BeL38HC3S5jhzbMEnVrx4CSM2LHy3EnKCQdUK2k7PCnc
```

Depending on how strongly one wants to guard against running out of pigeonholes, one could shorten the hash component to just the first eight characters:

```
> (cadify 'sensitive sensitive-desc #:short? #t)
'sensitive-BeL38HC3
```

Either would be possible with different tradeoffs, though for consistency, a single choice should be made in a system.

The algorithm for deriving this identifier is quite simple:

- Take two arguments, a shortname[1] (a string) and a description (may be any Preserves document)
- Canonicalize description to Syrup
- Hash canonicalized description with sha256, then hash with sha256 again (producing sha256d), and finally encode with base58
- If shortening, reduce hashed-and-encoded description to first eight characters

[1] The shortname is what is called in a petnames system a "self-proposed name".

- Append together the shortname, a dash, then the hashed-and-encoded description[2]
- That's it!

Hm, notice that the description can be any Preserves document... so, structured data rather than "just strings" are possible.

We could imagine this could be useful for various things. For instance, we could use a dictionary which describes several fields. Perhaps we are playing a game where we are describing creatures, and we want to know their `name`, a descriptive `desc` (here using SXML inspired markup), and a set of habitats:

```
(define ogre-desc
  (cadify "ogre"
          `#hash((name . "Ogre")
                 (desc . ((p "A large, bumbling beast..")
                          (p (b "WARNING:")
                             " Ogres get angry very quickly!")))
                 (habitats . ,(set "cave" "forest" "swamp"
                     ↪  "dungeon")))))
```

As we can see, this hashes just fine:

```
(cadify 'ogre ogre-desc)
'ogre-3LT14kW6WBzfC1JcZArLu92kV7nzdQNDDjbHMPUbCLMz
```

Strictly speaking, content addressing is always a kind of optimization, as the equivalent non-hashed content could always be used in place. This is true, but the compression gains of using content-addressing are significant.[3] Since we result in the same hash every time, it is fairly trivial to also set up a repository from which the expanded content addressed definitions can be retrieved every time.[4]

[2] We have glossed over what "append together" means; presumably we mean string concatenation, but in this example we have coerced the string into a symbol. In some contexts, a string-only approach would be appropriate, perhaps in others a URI. In others still we might want a "struct" or other more formalized record representation. It would be important to be consistent about how to represent these, so in time it would be worthwhile to "converge" on a representation. In the meanwhile, in describing these properties generally, we are being loose about things.

[3] Imagine explaining what a new word means based on several known words, and instead of using the new word in a conversation, repeating the explanation every time in its place!

[4] The code for this is trivial to write... see the Fetching descriptors section of this document for

Like words in ordinary language, content addressed terms can also be referenced within the definitions of other content addressed terms.

5 Distributed objects and content addressed descriptors

We'd like to apply our content-addressed descriptors to distributed objects, but, well... first we need a distributed object system. Spritely Goblins is a distributed object (or, "classic actor model") system following object capability security patterns.

5.1 Wait! We need a tutorial of Spritely Goblins!

This particular article isn't concerned with teaching you how to use Spritely Goblins, but okay I suppose some amount of introduction is in order, so let's be brief. First, Goblins is written in Racket, which is a kind of Scheme, which is a kind of Lisp, hence all the parenthetical syntax. See Racket's documentation. For that matter, also see Goblins' documentation.

```
(require goblins)
```

But okay, here's a small, just barely mini-tutorial of Goblins.[5] We can define a

an example implementation.

[5]We are cheating a bit here; `spawn`, `$`, `<-`, and `on` all only work inside of a goblins "turn" which sets up the appropriate "goblins context". Most Goblins programming is done assuming such a Goblins context is already running, but bootstrapping a program requires setting one up. Since we are showing off examples running within a REPL in an exploratory style, unfortunately this requires setting up one for every interaction. (Perhaps a Racket `#lang` can reduce some of this pain in the future.) See the Goblins tutorial for more on how to do this, but here is a simple way to do it:

```
(require goblins/actor-lib/bootstrap)
(define-vat-run vat-run (make-vat))
```

Then you can interpret code like:

```
> (define alice (spawn ^greeter "Alice"))
> ($ alice "Bob")
"Hello Bob, my name is Alice!"
```

as:

constructor, let's call it `^greeter`:

```
;; Constructor
(define (^greeter bcom my-name)
  ;; Handler
  (lambda (your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)))
```

We can then spawn such a greeter:[5]

```
> (define alice (spawn ^greeter "Alice"))
```

`spawn` implicitly passes a `bcom` to the constructor (used to update behavior of objects).[6] `"Alice"` is thus what is passed to `my-name`.

The inner lambda is the message handler. We can invoke an actor in a sequential call-return style with `$`:

```
> ($ alice "Bob")
"Hello Bob, my name is Alice!"
```

Asynchronous programming is also possible and uses the `<-` operator, which returns a promise. We can then set up a callback to this promise with `on`:

```
> (define greet-me-vow (<- alice "Ben"))
> (on greet-me-vow
      (lambda (we-heard)
        (displayln (format "Heard: ~a" we-heard))))
;; Prints out the following:
;;   Heard: Hello Ben, my name is Alice!
```

```
> (define alice (vat-run (spawn ^greeter "Alice")))
> (vat-run ($ alice "Bob"))
"Hello Bob, my name is Alice!"
```

[6] There's a lot more to say about this; the entirety of state changes can be managed with `bcom`

Note that Goblins implements CapTP, which is a distributed networked protocol. Code that is written using asynchronous message sending with <- works trivially over the network with CapTP. Since this portion of network code is abstracted for us, we need not spend significant time thinking about how to invent new protocols for each kind of actor we make; the behavior of that actor is already possible to be made available on the network over CapTP without any serious changes to Goblins code which already makes use of asynchronous message passing with <-.

There is plenty more that can be said about Goblins, but this is the entirety of what we need to continue with our article. So continue we shall!

5.2 A brief aside on methods in Goblins

"Method dispatch" is not first-class in Goblins. Instead, method dispatch is a possible convention, where the first argument to an invocation is considered the selected method name (for objects which use the method-dispatch pattern). The "methods" macro is a common tool to make writing these easy:[7]

```
(require goblins/actor-lib/methods)

(define (^creature bcom name adjective noise)
  (methods
   [(get-name) name]
   [(interaction)
    (format "You see the ~a ~a, which says \"~a\"!!!"
            adjective name noise)]))
```

```
> (define wolf (spawn ^creature "wolf" "sneaky" "AWOOO"))
> ($ wolf 'get-name)
"wolf"
> ($ wolf 'interaction)
"You see the sneaky wolf, which says \"AWOOO\"!!!"
```

5.3 Adding a descriptive method

Now that we have a way to have multiple methods on an object, this seems like it might be an excellent way to add content-addressed vocabulary to an object.

(pronounced "become"), which is a functional way of handling state changes for actors.

[7] All the methods macro does is create a procedure that dispatches based on the first argument.

Really, that's it!

```
(define goblin-cad
  (cadify 'goblin
          "A small mythical creature with pointed ears."))

;; ...
(define (^goblin _bcom ...)
   (methods
     [(foo-method) ...]
     [(_describe) (set goblin-cad gameobj-cad monster-cad
                       attackable-cad befriendable-cad
                            ↪  lookable-cad)]))
```

Now we can call a Goblin with the `'_describe` method, and it should return a set of descriptions.

Of course, we want consistency: when we are looking to describe an object, we should expect a uniform way to find out that object's descriptors. The solution in the E programming language is to provide miranda methods, which are commonly named object methods, and which if the user does not provide such a method, a default method is supplied for them (for instance, a reasonable default here would be to return an empty set).

5.4 Miranda Methods don't work when methods are inessential

Unfortunately, the above approach won't work for us. As we said, methods are not first-class in Goblins, and therefore we cannot expect all objects to support them. There may be many actors, such as instantiations of our `^greeter` earlier, which do not accept a method argument.

But without methods as an essential construct, how can we invoke something that is not the standard behavior of the object? How could we request a set of descriptors from `^greeter`, for example?

5.5 Warding off method behavior

Goblins has a solution for behavior that is not the default behavior. This behavior is present, but invisible: it is warded off from the user by a "warden", and only the relevant "incanter" can cast the right spell, exposing the hidden functionality.[8]

[8] This idea of "wards" and "incanters" has a history, immediately borrowed from but modified from an idea called energetic secrets in the wonderful Joule language, the direct precursor to E. The idea of energetic secrets is that methods are not broadly available; one needs a capability to access them. Like with wards and incanters, Joule used sealers and unsealers to implement what is called rights amplification. (Joule's energetic secrets itself was inspired by the T programming language's

First, we need to make a ward and incanter pair:[9]

```
(require goblins/actor-lib/ward)

(define-values (describe-warden describe-incanter)
  (spawn-warding-pair))
```

Now that we have this pair, we can make a modified version of our `^greeter`:

```
(define (^greeter bcom my-name)
  ;; Main behavior
  (define (main-beh your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name))
  ;; Descriptors behavior
  (define (describe-beh)
    (set greeter-cad))

  ;; Ward off the descriptors behavior so that only the descriptors
  ;; incanter can access it.  Fall back to main behavior if not
  ;; invoked by the incanter.
  (ward describe-warden describe-beh
        #:extends main-beh))
```

Now, we can use the `describe-incanter`:

```
> ($ describe-incanter my-greeter)
(set 'greeter-7QA4GgZ7RsxDTuruyyAnihBw1euENDnu2rKrbuhpz4dj)
```

This same pattern also be used with `<-`, so works perfectly fine over the network.[10]

However, the new `^greeter` definition is not very comfortable to write. If we had to describe many actor definitions this way, it would feel quite tedious. Thankfully, it is not much work to make a usable utility.

---

approach to lexical-reference based dispatch, a stark contrast to the implicit-name-quoting approach of method dispatch used generally.)

[9] We are once again glossing over the fact that invoking spawn can only be done in an actor context.

[10] If the warden and your actor are not in the same vat, be sure to pass in the `#:async?` keyword to the `ward` procedure.

```
(define (describe descriptions main-beh)
  (define (describe-beh) descriptions)
  (ward describe-warden describe-beh
        #:extends main-beh))
```

Now `^greeter` is much easier to read:

```
(define (^greeter bcom my-name)
  (describe
   (set greeter-cad)
   (lambda (your-name)
     (format "Hello ~a, my name is ~a!" your-name my-name))))
```

With Goblins being a lisp, we could add macros to make this even more convenient to read and write. Such an activity is left as an exercise for the reader.

5.6 Fetching descriptors

At the beginning of this document, we talked about the experience of coding at a REPL and being able to have a "conversation". We can now query for a set of descriptors describing an object, but we don't seem to have any way to retrieve what those descriptors mean… good enough to know if you and I mean the same thing in advance, but not good enough for me to discover new things that you mean.

Thankfully, since our terms are content-addressed, this is easy enough to "solve".

Here is an example of a procedure which can spawn both a registry of terms as well as an object which can retrieve them:

```
(define (spawn-cad-registry-pair [cad-registry #hasheq()])
  (define registry (spawn ^hash cad-registry))
  (define (^register-cad bcom)
    (lambda (name description)
      (define cad (cadify name description))
      ($ registry 'set cad (record* 'cad name description))
      cad))
  (define (^fetch-cad bcom)
    (lambda (name)
      ($ registry 'ref name)))
  (values (spawn ^register-cad)
          (spawn ^fetch-cad)))
```

With this, we have two separate capabilities; one allows users to register terms they may find interesting, and another allows fetching them. Through this, it should be possible to retrieve the definitions corresponding to the content-addressed term identifier. The receiving party can even verify that the hash matches the proclaimed definition, ensuring accuracy.

6 From Descriptors to Interfaces

So far we've looked at terms that are descriptors of objects. These descriptors are more for human consumption than for computer consumption.

This is useful; two programmers using the same term are likely to know they have a shared understanding of an object, and reading descriptions can help with understanding and recovery, but can we add descriptors in such a way that make them feel more meaningfully part of the program?

6.1 Describing methods

The first way we'll look at useful program-relevant aspects of our content addressed terms is to actually use them as our method names. Since, as we've said, method names are a convention rather than a built-in. Since the convention is a symbol as a first argument, and our content addressed terms can be rendered as symbols, this mean they can also be our object's first argument. We can also add this to our object's set of descriptors, and in this way what our object is can do is also what our object is described as.

6.2 A taste of an IDL

It would be even better if our methods, or object as a whole, could be described in terms of how it can be invoked. Having such descriptions could help annotate behavior in our IDE, can give nicely rendered invocation explanations translated to our programming language syntax of choice, do (a degree of) type checking, and so on.

An initial source of inspiration could be Racket's contract system. Here is a simple contract for a procedure which takes two arguments, a string and a positive integer and returns another string (the final argument to `->` is the type of the return value):

```
(define parrot-string/c
  (-> string? positive-integer? string?)))
```

(By convention, contracts end with `/c` in Racket.)

This can be used to guard a procedure against misuse with `define/contract`:

```
(define/contract (parrot-string str numtimes)
  parrot-string/c
  (string-append
   "BAWK! "
   (for/fold ([result-str "BAWK!!!"])
             ([i numtimes])
     (string-append str "! " result-str))))
```

It's easy enough to actually turn this into a Goblins actor, since `parrot-string` can simply be the current behavior:

```
(define/contract (^parrot _bcom)
  parrot-string/c
  parrot-string)
```

Now let's say we spawned an instance of this parrot and invoked it incorrectly:

```
> (define parrot (spawn ^parrot))
> ($ parrot 'wrong "arguments")
; parrot-string: contract violation
;   expected: string?
;   given: 'foo
;   in: the 1st argument of
;       (-> string? positive-integer? string?)
;   contract from: (function parrot-string)
```

So now our actor's behavior is protected, assuring that it can only be invoked with the correct type of inputs. However, we are interested in this information being made available to clients, including remote clients, of an object.

Consider again our contract:

```
(define parrot-string/c
  (-> string? positive-integer? string?))
```

We need some way to represent this in a way that can be conveyed to a remote reader. What kind of syntax should we use to describe a remote contract?

Well, for the moment let's go with the simplest idea possible. We will use a syntax that is modeled directly off of Racket's contract system… in fact, we can simply quote the previous expression:

```
(define parrot-string/iface
  '(-> string? positive-integer? string?))
```

Of course, this is not the same as an evaluated contract. But we could use this as a kind of DSL. It would be even trivial to simply "eval" this in a safe context, but we do not even need to resort to that. Making a transformer for this DSL is trivial:

```
(define (quoted->contract iface)
  (define (convert-one iface)
    (match iface
      [(list '-> args ... return-type)
       (define converted-args
         (map convert-one args))
       (dynamic->* #:mandatory-domain-contracts converted-args
                   #:range-contracts (list (convert-one
                       ↪  return-type)))]
      [(list 'and/c ifaces ...)
       (apply and/c (map convert-one ifaces))]
      [(list 'or/c ifaces ...)
       (apply or/c (map convert-one ifaces))]
      ['string? string?]
      ['integer? integer?]
      ['positive-integer? positive-integer?]
      ['float? (and/c number? inexact?)]
      ['symbol? symbol?]))
  (convert-one iface))
```

This simple procedure is sufficient to use as a basis for developing a simple DSL that takes out quoted contract-style DSL and restores them to the same contracts they represent:

```
> (define/contract (takes-string-and-positive-int str int)
    (quoted->contract '(-> string? positive-integer? string?))
    "yup it worked")
> (takes-string-and-positive-int "foo" 33)
"yup it worked"
> (takes-string-and-positive-int 'foo "33")
; takes-string-and-positive-int: contract violation
;   expected: string?
;   given: 'foo
;   in: the 1st argument of
;       (-> string? positive-integer? string?)
;   contract from:
;       (function takes-string-and-positive-int)
```

It is also possible to build a simple pretty printer for this syntax:

```
(define (explain-iface iface [indent-level 0] [bullet? #f])
  (define first-line? #t)
  (define (displayln-indent str)
    (display "; ")
    ;; indent
    (for ([i (* indent-level 2)])
      (display " "))
    ;; maybe display a bullet
    (when bullet?
      (if first-line?
          (begin (set! first-line? #f)
                 (display "- "))
          (display "  ")))
    ;; display the rest
    (displayln str))
  (define next-indent-level
    (if bullet?
        (+ indent-level 2)
        (+ indent-level 1)))
  (match iface
    [(list '-> args ... return-type)
     (displayln-indent (format "A procedure which takes ~a arguments:"
                               (length args)))
     (for ([arg args])
       (explain-iface arg next-indent-level #t))
     (displayln-indent "and returns:")
     (explain-iface return-type next-indent-level)]
    [(list 'and/c ifaces ...)
     (displayln-indent "Something which meets *all* of:")
     (for ([iface ifaces])
       (explain-iface iface next-indent-level #t))]
    [(list 'or/c ifaces ...)
     (displayln-indent "Something which meets *any* of:")
     (for ([iface ifaces])
       (explain-iface iface next-indent-level #t))]
    ['string? (displayln-indent "A string")]
    ['integer? (displayln-indent "An integer")]
    ['positive-integer? (displayln-indent "A positive integer")]
    ['float? (displayln-indent "A floating point number")]
    ['symbol? (displayln-indent "A symbol")]
    ['boolean? (displayln-indent "A boolean")]))
```

As we can see, this works just fine:

```
> (explain-iface '(-> string? positive-integer? string?))
; A procedure which takes 2 arguments:
;   - A string
;   - A positive integer
; ... and returns:
;   A string
```

It is even able to handle nested structures!

```
> (explain-iface '(-> (-> (or/c symbol? string?) string?)
    ↪  positive-integer? string?))
; A procedure which takes 2 arguments:
;   - A procedure which takes 1 arguments:
;       - Something which meets *any* of:
;           - A symbol
;           - A string
;     ... and returns:
;       A string
;   - A positive integer
; ... and returns:
;   A string
```

Of course, what we need to do next is actually hook up such descriptions to our object so that they are queryable. We will use a warden/incanter pair again to do this:

```
(define-values (describe-warden describe)
  (spawn-warding-pair))
```

Now we can define a simple procedure to share the interface with us and attach it, warded, to the object's behavior at construction time.

```
(define (^parrot _bcom)
  (ward describe-warden
        (lambda _ parrot-string/iface)
        #:extends parrot-string
        #:async? #t))
```

Given both a reference to both the `describe` incanter and a parrot, we can ask for its interface from anywhere on the network:

```
> (a-run
   (on (<- describe parrot)
       explain-iface))
; A procedure which takes 2 arguments:
;   - A string
;   - A positive integer
; ... and returns:
;   A string
```

Since the described s-expression is a Syrup-compatible expression, this also means we could compose it into a larger description and even turn this entire structure into a content addressed descriptor:

```
> (cadify 'parrot-string parrot-string/iface)
'parrot-string-H4N94Ubct7Ng96Q9Rijm89Do1gTkqareckGYQ479xUkQ
> (cadify 'parrot-actor
          (record* 'actor
                   "Like a parrot, repeats everything back to you"
                   (cadify 'parrot-string parrot-string/iface)))
'parrot-actor-4NfCmGsgYhidKpGH7kFfD7v3cJGCgBw2UqfX9VEHTrgc
```

Here we showed a very small (and incomplete) DSL which is purely illustrative.[11] A more robust DSL should follow similar principles but would be more carefully considered for the needs of its users. On this note, Preserves Schema is probably

[11] Notably we have also made querying for what interface an actor follows was implemented in such a way that the invoker was an explicit capability. It would be possible for us to instead make querying for descriptors universal and first-class. We could make a global pair of the warden/incanter for descriptions and simply apply them everywhere within a process. To make this work over CapTP, we could take advantage of the "bootstrap" actor associated with a connection, and attach a description-showing incanter to it by default. This allows for behavior akin to the self-descriptive miranda-methods without needing to take a method-first approach. However, consider that the ward/incanter approach already indicates that there may be times where some actors have "hidden" functionality. Thus, it might be that some users might have information about methods that are applicable to them and not others… for example, some users might be administrators of a social network and have access to extra methods to silence some users which others do not have access to. (An excellent example of this where Object-Capability Security in Virtual Environments when demonstrating "rights amplification" where teachers have access to some methods on represented participants that the students do not.) As such, it would be appropriate for different users to also get different descriptions relevant to what they are expected to be able to do.

the right design, and unshockingly compatible with Syrup (since it is a Preserves encoding).

7 Limitations in a mutually suspicious network

What we might observe is that the contract we showed in the IDL section relied on runtime enforcement. The difference between runtime contracts and static type checking is merely whether it can be done ahead of time. This is mostly an optimization, but it can also be used to anticipate and protect against bugs when expected behavior is violated.

The general idea of anticipating and warning against unexpected behavior may still hold. However, in a mutually suspicious network, the most optimistic one may get about optimizing away contracts is at the boundary of mutual suspicion.

This is because optimizations are generally done from the standpoint of a kind of pseudo-centralized planner. A static type checker for a Haskell program can eliminate checks between procedures by knowing the complete annotations of types between invoked procedures.

This is not the case when we enter the arena of mutual suspicion on the network. As an example, consider if Alice on environment A invokes Bob on environment B, who fulfills a promise (as its return value) returning what, according to Bob's alleged contract, will be a non-negative integer. Alice passes this value to Andrew (who, like Alice, is on environment A), who expects in turn a non-negative integer, and would behave very dangerously (for instance, perhaps corrupting a financial database) if a negative integer would happen to make its way through. In a fully local statically checked system with a language environment that is able to perform full trusted analysis, the language environment could detect that Bob should return a non-negative integer to Alice's continuation and that Alice would then pass this integer to Andrew, and thus remove the runtime check between Alice and Andrew.

But what if Bob were to misbehave? In a mutually suspicious system such as Goblins, we cannot trust the contracts of other entities so much that we assume them to be true.

However, optimizations can still be done. Alice and Andrew's language environment could recognize that a boundary contract enforcement must still be performed on the value from Bob's fulfilled promise. However, since Alice and Andrew's behaviors can be fully analyzed and are indeed managed by the same language environment, the environment can safely remove the runtime check between Alice and Andrew if Bob's behavior is still checked at runtime (whether by

validation, or validation through parsing).[12]

If environment A is able to "trust" that environment B will perform as it claims it will, these restrictions may be removed, at the peril of this trust being false. Distributed communication with a trusted auditor could be one way to do this; the auditor provides assurance that the code works as expected, and A (or the relevant entities on A that would otherwise demand a contract check) can perform optimized routines with those assurances.

So it is our observation here that the general case of mutual suspicion implies that proclaimed interfaces are really "alleged" interfaces. Still, we should not take this to feel bad about our designs… that we can safely operate in mutually suspicious environments at all is a wonderful kind of success! We must simply understand the emergent limitations of the problem domain.

8 Implementation

```
#lang racket/base

(require racket/match
         racket/contract
         crypto
         crypto/libcrypto
         base58
         syrup
         goblins
         goblins/actor-lib/common)

(provide cadify
         split-cad-symbol
         symbol->cad)

(crypto-factories (list libcrypto-factory))

;; Avoid length extension attacks
(define (sha256d input)
  (digest 'sha256 (digest 'sha256 input)))

(define/contract (cadify shortname description #:short? [short? #f])
  (->* ((or/c string? symbol?) any/c) (#:short? boolean?) symbol?)
  (define base58-hash
```

[12] Note that the requirements on boundary checks between Typed Racket and Untyped Racket are very similar and operate in much the mechanism described here: enforcement occurs primarily at the border.

```
    (base58-encode (sha256d (syrup-encode description)) #:check? #f))
  (string->symbol
   (format "~a-~a"
           shortname
           (if short?
               (substring base58-hash 0 8)
               base58-hash))))

(module+ test
  (require rackunit)

  (crypto-factories (list libcrypto-factory))

  (define look-desc "\
Look at a fantasary game object, returning a string \
designating current description.  Takes no arguments.")

  (define look-cad (cadify 'look look-desc))

  (define look-scad (cadify 'look look-desc #:short? #t))

  (define fantasary-gameobj-desc
    "A game object in the Fantasary game system."))

(define (spawn-cad-registry-pair [cad-registry #hasheq()])
  (define registry (spawn ^hash cad-registry))
  (define (^register-cad bcom)
    (lambda (name description)
      (define cad (cadify name description))
      ($ registry 'set cad (record* 'cad name description))
      cad))
  (define (^fetch-cad bcom)
    (lambda (name)
      ($ registry 'ref name)))
  (values (spawn ^register-cad)
          (spawn ^fetch-cad)))


(module+ test
  (require goblins/actor-lib/bootstrap)
  (define-vat-run vat-run (make-vat))
  (define-values (register-cad fetch-cad)
    (vat-run (spawn-cad-registry-pair)))
  (test-equal?
   "Register cad claims to register the cad"
   (vat-run ($ register-cad 'foo-bar "A foo and a bar, by far, by
       ↪  far"))
   'foo-bar-2aNSpqx96JFBzdSWUQXJZu6haew1tF9hLRicqHyCmyiX)
```

```
  (test-equal?
   "fetch-cad gives us the cad's definition"
   (vat-run ($ fetch-cad
       ↪  'foo-bar-2aNSpqx96JFBzdSWUQXJZu6haew1tF9hLRicqHyCmyiX))
   (record* 'cad 'foo-bar "A foo and a bar, by far, by far")))


;;;; Not really used yet.  Maybe later.

(define (cad hash description)
  (record* 'cad hash description))

(define (cad? obj)
  (match obj
    [(record 'cad (list string? bytes?))
     #t]
    [_ #f]))

(define (cad-hash cad)
  (match cad
    [(record 'cad (list (? string? _shortname) (? bytes? hash)))
     hash]))

(define (cad-shortname cad)
  (match cad
    [(record 'cad (list (? string? shortname) (? bytes? _hash)))
     shortname]))

(define (cad->symbol cad)
  (match cad
    [(record 'cad (list (? string? shortname) (? bytes? hash)))
     (define base58-hash (base58-encode hash #:check? #f))
     (string->symbol
      (format "~a-~a"
              shortname
              (substring base58-hash 0 8)))]))

(define (split-cad-symbol cadify-symbol)
  (define str (symbol->string cadify-symbol))
  (define last-dash #f)
  (define str-len (string-length str))
  (for ([i str-len])
    (when (eq? (string-ref str i) #\-)
      (set! last-dash i)))
  (if last-dash
      (values (substring str 0 last-dash)
              (substring str (add1 last-dash) str-len))
      (error "No dash in string")))
```

```
(define (symbol->cad cad-sym)
  (define-values (shortname encoded-hash)
    (split-cad-symbol cad-sym))
  (define hash (base58-decode encoded-hash #:check? #f))
  (cad shortname hash))
```

9 Examples

9.1 CAD examples

Pretty simple:

```
cd public
racket cad-examples.rkt
```

It's pretty simple, it just shows how various kinds of descriptors can be content-addressed in a consistent way.

```
#lang racket

(require "cad.rkt"
         syrup)

(define (explain-cad shortname description)
  (displayln (format "** Shortname: ~a**" shortname))
  (displayln "** Description: **")
  (pretty-print description)
  (displayln "** CAD: **")
  (displayln (format "  ~a " (cadify shortname description)))
  (newline))

(displayln "==== Simple strings ====")
(define sensitive-desc "\
The sensitive property (a boolean) on an object indicates \
that some users may wish to apply discretion about viewing its \
content, whether due to nudity, violence, or any other likely \
aspects that viewers may be sensitive to. This is comparable to \
what is popularly called \"NSFW\" (Not Safe For Work) or \
\"trigger warning\" in some systems. Implementations may choose \
```

```
to hide content flagged with this property by default, exposed at user
    ↪ discretion.")
(explain-cad 'sensitive sensitive-desc)

(displayln "==== Composite structures ====")
(define ogre-desc
  `#hash((name . "Ogre")
         (desc . ((p "A large, bumbling beast..")
                  (p (b "WARNING:")
                     " Ogres get angry very quickly!")))
         (habitats . ,(set "cave" "forest" "swamp" "dungeon"))))
(explain-cad 'ogre ogre-desc)

(displayln "==== Interface descriptions ====")
(define parrot-string/iface
  '(-> string? positive-integer? string?))
(explain-cad 'parrot-string parrot-string/iface)

(displayln "==== Composite actor descriptions, with interfaces ====")
(define parrot-actor-desc
  (record* 'actor
           "Like a parrot, repeats everything back to you"
           (cadify 'parrot-string parrot-string/iface)))
(explain-cad 'parrot-actor parrot-actor-desc)
```

9.2 Rich interface examples

You'll need to launch the Tor daemon for Goblins. It's mildly annoying to set up at the time of writing, but the instructions are on the Goblins CapTP documentation page.

Okay, done? Then open two terminals. In the first one:

```
cd public
racket rich-interface-server.rkt
```

That'll print out two "sturdyrefs", one for the parrot, one for the describe incanter, that you want to paste to `rich-interface-client.rkt`.[13] But actually, the last line it spits out is just the whole command you want to copy-paste to run the entirety of that command for your convenience.

[13] By the way, Goblins with Tor Onion Services is peer to peer so client to server so this is kind of a misnomer. But it's the eaiest way to say "here's the thing that will be connecting, and here's the

Anyway, once you do so it'll do a few things:

- Fetch and print the structured representation of the parrot's remote interface
- Print that out in a human-meaningful form
- Invoke the parrot and print out its result

```
#lang racket

(provide quoted->contract
         explain-iface)

(require syrup
         goblins/actor-lib/ward)

(define (quoted->contract interface)
  (define (convert-one iface)
    (match iface
      [(list '-> args ... return-type)
       (define converted-args
         (map convert-one args))
       (dynamic->* #:mandatory-domain-contracts converted-args
                   #:range-contracts (list (convert-one
                        ↪  return-type)))]
      [(list 'and/c ifaces ...)
       (apply and/c (map convert-one ifaces))]
      [(list 'or/c ifaces ...)
       (apply or/c (map convert-one ifaces))]
      ['string? string?]
      ['integer? integer?]
      ['positive-integer? positive-integer?]
      ['float? (and/c number? inexact?)]
      ['symbol? symbol?]
      ['boolean? boolean?]))
  (convert-one interface))

(define (explain-iface iface [indent-level 0] [bullet? #f])
  (define first-line? #t)
  (define (displayln-indent str)
    (display "; ")
    ;; indent
    (for ([i (* indent-level 2)])
      (display " "))
```

thing it will connect to" in familiar language I know of.

```
    ;; maybe display a bullet
    (when bullet?
      (if first-line?
          (begin (set! first-line? #f)
                 (display "- "))
          (display "  ")))
    ;; display the rest
    (displayln str))
  (define next-indent-level
    (if bullet?
        (+ indent-level 2)
        (+ indent-level 1)))
  (match iface
    [(list '-> args ... return-type)
     (displayln-indent (format "A procedure which takes ~a arguments:"
                               (length args)))
     (for ([arg args])
       (explain-iface arg next-indent-level #t))
     (displayln-indent "... and returns:")
     (explain-iface return-type next-indent-level)]
    [(list 'and/c ifaces ...)
     (displayln-indent "Something which meets *all* of:")
     (for ([iface ifaces])
       (explain-iface iface next-indent-level #t))]
    [(list 'or/c ifaces ...)
     (displayln-indent "Something which meets *any* of:")
     (for ([iface ifaces])
       (explain-iface iface next-indent-level #t))]
    ['string? (displayln-indent "A string")]
    ['integer? (displayln-indent "An integer")]
    ['positive-integer? (displayln-indent "A positive integer")]
    ['float? (displayln-indent "A floating point number")]
    ['symbol? (displayln-indent "A symbol")]
    ['boolean? (displayln-indent "A boolean")]))
```

```
#lang racket

(require goblins
         goblins/actor-lib/bootstrap
         goblins/actor-lib/ward
         syrup
         "rich-interface-tools.rkt"
         "cad.rkt")

;; Description warden/incanter pair
```

```
;; ================================
(define-vat-run describe-vat-run (make-vat))
(define-values (describe-warden describe)
  (describe-vat-run
   (spawn-warding-pair #:async? #t)))

;; Repeater behavior and contracts
;; ================================

(define parrot-string/c
  (-> string? positive-integer? string?))

(define parrot-string/iface
  '(-> string? positive-integer? string?))

(define parrot-string-cad
  (cadify 'parrot-string parrot-string/iface))

(define/contract (parrot-string str numtimes)
  parrot-string/c
  (string-append
   "BAWK! "
   (for/fold ([result-str "BAWK!!!"])
             ([i numtimes])
     (string-append str "! " result-str))))

(define (^parrot _bcom)
  (ward describe-warden
        (lambda _ parrot-string/iface)
        #:extends parrot-string
        #:async? #t))

(module+ main
  (require goblins/ocapn
           goblins/ocapn/netlayer/onion)

  (define-vat-run a-run (make-vat))
  (define-vat-run machine-run (make-vat))

  (define mycapn
    (machine-run (spawn-mycapn (setup-onion-netlayer))))

  (define parrot (a-run (spawn ^parrot)))
  (define parrot-sref
    (machine-run ($ mycapn 'register parrot 'onion)))
  (define describe-sref
    (machine-run ($ mycapn 'register describe 'onion)))
```

```
  (displayln "*** Connect to parrot at:")
  (displayln (ocapn-sturdyref->string parrot-sref))
  (displayln "*** Connect to describe incanter at:")
  (displayln (ocapn-sturdyref->string describe-sref))
  (displayln "*** Or, simply run:")
  (displayln (format "racket rich-interface-client.rkt ~a ~a"
                     (ocapn-sturdyref->string parrot-sref)
                     (ocapn-sturdyref->string describe-sref)))

  ;; keep things alive hack
  (sync (make-semaphore)))

(module+ test
  (define-vat-run a-run (make-vat))
  (define parrot (a-run (spawn ^parrot)))

  (a-run
   (on (<- describe parrot)
       explain-iface))

  )
```

```
#lang racket

(require goblins
         goblins/actor-lib/let-on
         goblins/actor-lib/await
         goblins/actor-lib/bootstrap
         goblins/ocapn
         goblins/ocapn/netlayer/onion
         syrup
         "rich-interface-tools.rkt")

(define (main parrot-sref describe-sref)
  (define-vat-run a-run (make-vat))
  (define-vat-run machine-run (make-vat))

  (define mycapn
    (machine-run (spawn-mycapn (setup-onion-netlayer))))
  (define done (make-semaphore))
  (define (uhoh _e)
    (displayln "Uhoh, something went wrong, huh?")
    (semaphore-post done))
  (a-run
   (with-await await
```

```
      ;; using await like this avoids the crossed hellos problem
      ;; while also not solving the crossed hellos problem #thisisfine
      (define describe (await (<- mycapn 'enliven describe-sref)))
      (define parrot (await (<- mycapn 'enliven parrot-sref)))
      (await
       (on (<- describe parrot)
           (lambda (iface)
             (displayln ";; Parrot's interface (encoded):")
             (display ";") (write iface)
             (newline) (newline)
             (displayln ";; Parrot's interface (explained):")
             (explain-iface iface))
           #:catch uhoh
           #:promise? #t))
      (newline)
      (await
       (on (<- parrot "Repeat three times" 3)
           (lambda (parrot-says)
             (displayln ";; Parrot says:")
             (display "; ")
             (display parrot-says)
             (newline))
           #:catch uhoh
           #:promise? #t))
      (semaphore-post done)))
  ;; hack to wait to close this file until we hear back
  (let () (sync done) (void)))

(module+ main
  (command-line
   #:args (parrot-address describe-address)
   (main (string->ocapn-sturdyref parrot-address)
         (string->ocapn-sturdyref describe-address))))
```

10 Thanks, licensing, etc

This document, and the underlying work on which it builds, were generously funded by NLnet and NGI Zero. This grant supported many aspects which enabled this document to appear to propose such "simple" approaches: the CapTP networked communication layer implementation was largely funded by this grant. The initial assumptions were that the protocol itself would need to support interfaces as a first-class operation, but the development of the wards-and-incanters system ended up being the ultimate solution to this system. Having both CapTP and the ward/incanter structure allowed for the ideas explored here to be layered on top of

the underlying networked protocol in a more beautiful and simple way than was anticipated from the outset, also allowing for evolution of the underlying ideas within systems developed in practice. Thank you also to supporters of the FOSS & Crafts Studios patreon account, who have provided ongoing support to Spritely's development.

Sandro Hawke is the one responsible for pointing out that the paragraph of text in a specification describing expected behavior is the real determination of whether or not two parties mean the same thing as opposed to some arbitrary URI... so why not use the text instead? I thought this observation was great, which inspired the approach to content addressed definitions/vocabulary in general, though it turns out Sandro meant something else. The observation, nonetheless, laid the foundation for much of this writing.

The Object Capability Security community made enormous contributions, directly and indirectly to this document. The original design for CapTP came from the E programming language, and most of Goblins' design is directly descendent from those combining E and Scheme. There are too many people in the object capability community to thank directly, but Mark S. Miller's help in guiding Chris through fundamental concepts, and Kevin Reid's feedback on the idea of "interfaces" based on what an object claims it can do stand out. Dean Tribble came up with the idea of energetic secrets, on which the ward/incanter design is based. Although not exhaustive, comments from Carl Hewitt, Matt Rice, Dale Schumacher, Alan Karp, Ian Denhardt, Bill Frantz, Corbin Simpson, and Baldur Jóhannsson provided substantial impact on the thinking of this paper or on Goblins' networking layer itself.

Tony Garnock-Jones came up for the design for Preserves (of which Syrup is an encoding) and Preserves Schema and provided much thinking into what it means to encode data. Serge Wroclawski provided feedback on the idea of content addressed descriptors and their application to interfaces, went through and provided feedback on the Goblins tutorial, and wrote an example bot for goblin-chat. Jonathan Frederickson, Isak Andersson (bitpuffin) and Jessica Tallon also read and provided useful and early feedback on Goblins' tutorial and designs. Jonathan Rees contributed many hours of thinking on the nature of shared meaning in communication systems. Additionally, Jonathan Rees provided a clear link between object capability security and Scheme (and scheme-like languages generally), which was the author's initial introduction to the ideas and feasibility of object capability security. Stephen Webber helped guide me down many more computer science rabbit holes than I might have dared to tread otherwise. Morgan Lemmer-Webber proofread this paper, suggested structural edits, and generally provided much emotional support.