Communicating Concurrent Kittens

Brian Beckman

29 Mar 2023

Contents

1 Prologue

I must confess a mild mental disability: when I see mathematics or programs, I see drowning kittens, and I want to save them.

`x | y & z`, five little kittens, `x`, `|`, `y`, `&`, `z` drowning in the Syntactical Sea of Forgotten Precedence. Venus-the-boatwright rises on her half-shell, fists full of bows and sterns. She reaches down and builds little boats, each from a bow and a stern. She puts some little kittens in each boat. Sometimes, she puts another little boat in a boat in place of a kitten.

She kisses the first kitten (or little boat) in each little boat and says "You're the captain; take care of the others!" She blesses the little boats and sinks back into the sea, leaving `(& z (| x y))` sailing safely on the waves. Venus knows you didn't mean `(| x (& y z))`.

Have you been burnt by C code like `x|y&z` or `x/y*z`? Mathematicians and Physicists normally read x/yz as $x/(yz)$,[1, 2, 3] [4] but I know of no programming language that treats `x/y*z` like that. Rather, programming languages treat it like $(xz)/y$.[5]

Perhaps you can dream, like me, of saving the drowning kittens. Wright the boats in drydock[6] or asea,[7] right the boats so they float,[8] and write `(& z (| x y))` and `(* z (/ x y))`! Or did you mean `(| x (& y z))` and `(/ x (* y z))`? Only you can know!

If you don't like bows and sterns and little boats, perhaps you can tolerate me. My affliction is not my fault! It is like misophonia:[9] built-in, incurable, misery-making for others, inexplicable except to fellow sufferers.[10]

1.1 "What are you doing?"

Let's implement some fundamentals of the rho calculus[11] and the pi calculus[12] in a DSL[13] made of little boats, meaning "embedded in Clojure." Asynchronous behavior is easy to model in Clojure, and there is `clojure.spec`[14] for type checking.

[1] Section E.2.e of the Style Guide of the American Physical society states that multiplication has higher precedence than division, thus implies my statement.

[2] `https://cdn.journals.aps.org/files/styleguide-pr.pdf`

[3] I have not found recent corroboration from the American Mathematical Society, but I know I have read it in the past!

[4] `https://math.stackexchange.com/questions/213406/does-x-yz-mean-x-yz-or-x-yz`

[5] even in Mathematica, where `x / y z === x z / y`

[6] compiled ahead-of-time (AOT)

[7] interpreted or compiled just-in-time (JIT)

[8] type-check and optimize

[9] `https://www.webmd.com/mental-health/what-is-misophonia`

[10] Plus, I don't like writing parsers: it's boring.

[11] Meredith, L. G.; Radestock, Mattias (22 December 2005). "A Reflective Higher-Order Calculus". Electronic Notes in Theoretical Computer Science. 141 (5): 49–67. `https://doi.org/10.1016/j.entcs.2005.05.016`.

[12] `https://en.wikipedia.org/wiki/%CE%A0-calculus`

[13] `https://en.wikipedia.org/wiki/Domain-specific_language`

[14] `https://clojure.org/guides/spec`

Plus Clojure *looks like* little boats to save the drowning kittens. It's important for programs to look good. Mathematics and programming *are visual arts*.

1.2 "Why not RhoLang?"

You call up your friend Nancy and invite her to a party at your house. She asks "Will Ted be there?" You think to yourself *what fun!* and innocently say, "Yes, I think so! I invited him!" Nancy says "I'm terribly sorry, but I won't be able to make it to your party. Please have a great time and give my best to everyone!" Nancy hangs up and thinks to herself *except to Ted*.

Ted's last name is Scala.

Let's compromise. Nancy won't ever see Scala again, plus she can't stand to see kittens drowning, i.e., implicit precedence rules. Otherwise she likes RhoLang. Let's create something with the same semantics as RhoLang embedded in Clojure, and call it CrowLang.

Perhaps CrowLang can inter-operate with RhoLang. They're both on the Java Virtual Machine, after all. Or perhaps we'll write new lcompilers[15] [*sic*] for CrowLang and RhoLang. You can write an lcompiler for Scala, Nancy doesn't care. Lcompilers are fast, flexible, modularized, and easy to write. In fact, some day, lcompilers will use rho and pi for internal type-checking and other formalisms!

1.3 "Why not something better?"

Considered and rejected:

Agda — too obscure

Haskell or Mathematica — I think Mathematica is my all-time favorite programming language. Sadly, no one else will use it. Haskell is in my top-five favorites, rejected for the same reason.

Python — well, ok, umm, never mind, no

Common Lisp — not modern, otherwise fantastic!

Racket — designed for DSLs after all, but no one else uses it

Coq — full of rabbit holes, otherwise lovely!

C++ — I would drive us both crazy.

There *is* a path of least resistance for me, considering all things.

1.4 "Clojure doesn't have types!"

Not so. Clojure.spec[16] is at least as strong as types. It's not static, that's true, but `s/conform` *could* be static.

[15] `https://github.com/lcompilers`
[16] `https://clojure.org/guides/spec`

Static or not hardly matters in our case. One can build little boats in drydock before setting sail, or Venus-the-boatwright will build them at sea for us. One can check types, do rewriting, prove theorems. It's good enough.

Also, Clojure is already in our toolchain. We use it for abstract interpretation and test generation[17] for lfortran and lpython.[15]

1.5 "How did you write this?"

This is an executable document. When I produce a PDF from it, all code blocks are executed and results reported. I like this better than Jupyter notebooks for many reasons. This document is an instance of Knuth's literate programming[18] in org-babel.[19]

[17] https://github.com/rebcabin/asr-tester
[18] https://en.wikipedia.org/wiki/Literate_programming
[19] https://orgmode.org/worg/org-contrib/babel/

2 Π Communicating Concurrent Kittens

From the wiki page:[12]

P, Q	$::=$			
	$\mid$ 0	napping kitten	Do nothing; halt.	
	$\mid$ $x(y).P$	listening kitten	Listen on channel x for channel y.	
	$\mid$ $\bar{x}\langle y\rangle.P$	chatting kitten	Say "y" on channel x; don't wait.	(1)
	$\mid$ $P \mid Q$	two kittens	Run P and Q in parallel.	
	$\mid$ $(\nu\, x)\, P$	whispering kitten	fresh channel name x; Use it in P.	
	$\mid$ $!P$	mama cat	Run copies of P forever.	

This pi calculus is-low-level, like λ calculus, only with concurrency added. We'd have to build up numbers (like Church numerals), Booleans, sets, functions, conditionals, loops, everything. We'll do a little better, later. First, let's save some drowning kittens!

2.1 A Cartoon

Here is a tiny calculation cartoon, again from the wiki page, showing a reduction similar to an η-reduction in λ calculus:

$$\begin{array}{lll} (\nu\, x) & (& \bar{x}\langle z\rangle.0 \\ & \mid & x(y).\bar{y}\langle x\rangle.x(y).0 \\ & \mid & z(v).\bar{v}\langle v\rangle.0 \\ &) & \end{array} \tag{2}$$

I see four drowning kittens. Kitten Zero is a whispering kitten $(\nu\, x)\, P$. She whispers "x" to the other kittens, meaning "let's talk on channel x!" One might write:

$$(\nu\, x) \left(\begin{array}{ll} & \bar{x}\langle z\rangle.0 \\ \mid & x(y).\bar{y}\langle x\rangle.x(y).0 \\ \mid & z(v).\bar{v}\langle v\rangle.0 \end{array} \right) \tag{3}$$

Kitten Zero is obviously going to be the captain of a boat containing the remaining kittens, but we've only just started to wright boats.

> *We'll say "kitten" and mean "an actual kitten, or a little boat containing kittens or more little boats." Each little boat contains zero-or-more kittens and zero-or-more more littler boats.*

One of the three remaining kittens is chatting on channel x and the other two are listening, one on x and the other on z. These three are doing their things two at a time, $P \mid (Q \mid R)$ or $(P \mid Q) \mid R$, it doesn't matter how you think about it (*associativity of* par).[20]

[20] A better *par* boat could hold any number of kittens, in any order. We'll get there.

Kitten One, $\bar{x}\langle z\rangle.0$, chats on x, "Hey, let's continue chatting on z!" Then she takes a nap. Only one of the other kittens, Kitten Two, $x(y).\bar{y}\langle x\rangle.x(y).0$, listens on x.[21] She thinks, "Oy! Here I am waiting on x for someone to tell me where (on what channel) to continue, and I just heard 'continue on z,' so I'll switch to z. After I switch, I'll have something to say, but just let me switch, first!"

$$(\nu\, x)\left(\begin{array}{ll} & \bar{x}\langle z\rangle.0 \\ & \| \downarrow \\ | & x(y).\bar{y}\langle x\rangle.x(y).0 \\ & \quad \downarrow \ \downarrow \\ | & x(z).\bar{z}\langle x\rangle.x(y).0 \\ | & z(v).\bar{v}\langle v\rangle.0 \end{array}\right) \tag{4}$$

The out-channel $\bar{x}$ of Kitten One matches the in-channel x of Kitten Two; Kitten One said something and Kitten Two heard it. The *say-prefix*, $\bar{x}\langle z\rangle$, before the dot of Kitten One, $\bar{x}\langle z\rangle.0$, gets gobbled up, and then Kitten One takes a nap, 0. The *hear-prefix*, $x(y)$, before Kitten Two's first dot, the first dot of $x(y).\bar{y}\langle x\rangle.x(y).0$, also gets gobbled up. Plus, y changes to z in the next say-prefix, $\bar{y}\langle x\rangle$, of Kitten Two's first suffix, $\bar{y}\langle x\rangle.x(y).0$:

$$(\nu\, x)\left(\begin{array}{ll} & 0 \\ | & \bar{z}\langle x\rangle.x(y).0 \\ | & z(v).\bar{v}\langle v\rangle.0 \end{array}\right) \tag{5}$$

This is rather like the substitution of actual arguments for formal parameters in a function call in an ordinary programming language.

2.2 Free and Bound

Before this substitution of z for y, the y in Kitten Two's next say-prefix, $\bar{y}\langle x\rangle$, is *free*. That means it must change to z. However, the y in Kitten Two's next hear-suffix, $x(y).0$, isn't free. The y in $x(y)$ doesn't change to z because that y is *local* to the final suffix, 0. If 0 had more to do using y, that y would shadow the earlier y. In this case, the suffix 0 has no more to do; don't worry.

Bound is a synonym for *not free*. A variable y is either free or bound in a prefix, $(\nu\, x)$, $\bar{y}\langle x\rangle$, or $x(y)$. It can't be both.[22] Once y is bound, it's bound in all suffixes to the right up until the next binding of y. Any re-bindings of y in a long suffix pertain to the closest binding to the left. That closest binding must be a whispering kitten or a listening kitten.

[21] If more than one kitten listens on the same channel, one sees a classic race condition. A compiler can detect this directly from the syntax of the program! At run time, only one will hear and the other will starve, at least for a while.

[22] What about the strange case $x(x)$? We'll solve that soon.

2.3 Binding

There are only two ways to bind a name — only two *binding prefix forms*:

whispering — $(\nu\, x)\, P$ binds x in its suffixes $P.Q.\cdots$,
up until the next binding of x.

listening — $x(y).\, Q$ binds y in its suffixes $Q.R.\cdots$,
up until the next binding of y.

Definition 1. *binding, scope:* Each binding of a given name, say y, pertains to the entire suffix of its binding form, up until the next binding of y. That new binding *shadows* the prior binding. This is like the *environment model* or *lexical binding* of an ordinary programming language. A sequence of binding prefixes describes a right-hugging nest of *scopes* in which to look up values of bound variables.

Shadowing, if undesirable, can be removed by α-renaming the new bound occurrence of y, say to y_1, bringing the prior binding of y into scope of y_1. ■

α-Renaming is explained immediately below in Section 2.5.

In our example, looking at Kitten Two's suffix, $\bar{y}\langle x\rangle.\, x(y).\, 0$, one doesn't yet know *to what value* y gets bound. One can only find out later when the hear-prefix $x(y)$ lines up with x in a say-prefix like $\bar{x}\langle z\rangle$ again.

This usage of the word *bound* means *eventually bound to something*. The term *bound* by itself can be ambiguous, because one might also say *bound* when we *do* know *bound to what*.

2.4 Substitution

Here is a general rule for *substitution*, with some terminology to be clarified:

Definition 2. *substitution*: When the channel x of a left-most say-prefix, $\bar{x}\langle z\rangle$, equals the channel x of a left-most hear-prefix, $x(y)$, the prefixes are gobbled up and all free occurrences of y on the right of the hear-prefix suffer substitution of z for y. If there are two or more listeners on x, the results are non-deterministic. ■

2.5 Renaming

What if there were already some bound z's amongst the suffixes of free y's? The kitten listening on y and hearing z would have to patch that up first. It doesn't matter what temporary name she gives to a channel, so long as the same channel has the same bound name everywhere in the suffixes. One might rename preexisting z's something like z_1 so long as z_1 doesn't itself collide with preexisting names. That's *alpha renaming*. It might harmlessly un-shadow some names.

One doesn't have that problem here, but we might later. Kittens always remember their sailorly duty to clean up messes in their boats.

Definition 3. *renaming*: Prior to substitution of z for a free variable y in the suffixes of a hear-prefix, any bound occurrences of z to the right of the hear-prefix must be renamed consistently lest they collide with the incoming z that replaces y. ■

2.6 Animated Cartoons

I can't animate cartoons in a paper, but I visualize calculations as symbols moving around in an animated cartoon (please forgive another of my mental afflictions: synaesthesia). It saves me mistakes. I animate calculations with pen and paper.

2.7 Finishing Up

In Equation 5, Kitten Two, now $\bar{z}\langle x\rangle.x(z).0$, says on z "Switch to x, will you?" to whomever is listening. Then she waits and listens on x for y. Kitten Three, $z(v).\bar{v}\langle v\rangle.0$, is listening on z for a channel. She temporarily calls that channel v, but now she knows that v is really x:

$$(\nu\, x)\begin{pmatrix} & 0 \\ | & x(y).0 \\ | & \bar{x}\langle x\rangle.0 \end{pmatrix} \tag{6}$$

See how the z chat-listen pair got gobbled up and how x got substituted for both free v's in Kitten Three's suffix? If not, do an animation on paper. Kitten Three didn't have to patch up any bound x's, but she remembers to check. Both occurrences of x in $\bar{x}\langle x\rangle$ are free, just as both v's were free before substitution.

Kitten Three says "x" on x and takes a nap without waiting. Kitten Two hears on x that her temporary, bound channel name y really should be x again. She changes her y to x, notices she doesn't have any patching up or anything else to do, and takes a nap. If you don't see it in your mind's eye, animate it on paper.

$$(\nu\, x)\begin{pmatrix} & 0 \\ | & 0 \\ | & 0 \end{pmatrix} \tag{7}$$

In your animation, you'll see that Kitten Three becomes $x(x)$ after matching up and before renaming and substitution. This temporary condition appears to state that x is both bound and free in the same prefix, and that can't be!

The resolution is that the two x's are different x's! the first x, outside the parentheses, is a real, free name of a real channel — in fact, the channel furnished by and bound in the whispering Kitten Zero $(\nu\, x)$. That x is subject to *matching up* with a say-prefix on $\bar{x}$. The second x, inside the parentheses, is a bound stand-in for the real x said by $\bar{x}\langle x\rangle$. That real x gets substituted for stand-in x in the suffix, which happens to be 0, don't worry.

All the kittens are napping safely in the whisperer's boat.

2.8 All Names are Channels

Every variable, x, y, z, v, stands in for a communication channel. Sometimes one knows what channel a variable stands for, say a bound variable in a whisper or a free variable before matching or after substitution. Other times, a variable stands

for a channel we'll find out about later, say a bound variable in a hear-prefix matching. That's all one has so far: channels, known or unknown.

Here are the stages in a reduction:

1. **Matching** — a free channel x in a hear-prefix $x(y)$ equals a free channel $\bar{x}$ in a say-prefix $\bar{x}\langle z\rangle$. Exactly one of the matching hear-prefixes is chosen, nondeterministically. It is noted that z will replace x.
2. **Renaming** — All bound z's in the suffix of $x(y)$ are consistently renamed to prevent collisions with the incoming x.
3. **Substitution** — All free z's in the suffix of $x(y)$ are replaced with z.
4. **Gobbling** — $x(y)$ and $\bar{x}\langle z\rangle$ are removed, exposing the first prefix of their suffixes.

Sidestep the "funny" problem of $x(x)$; never construct it. Just gobble its predecessor hear-prefix.

2.9 Bail the Boats!

For now, we've got all kittens safely napping in the big "whisper" boat. But they're not *dry*. They had to bail out a *lot* of water — syntactic noise — to keep from drowning whilst Venus-the-boatwright was working. Venus will fix that with some little boats *inside* other boats, including the biggest "whisper" boat.

Venus first bails out most of the water, leaving little skeletal, boats-in-progress — ordinary mathematical function notation:

$$(\nu\, x)\left(\begin{array}{l} \quad \mathtt{say}(x, z, 0) \\ |\ \ \mathtt{hear}(x, y, \mathtt{say}(y, x, \mathtt{hear}(x, y, 0))) \\ |\ \ \mathtt{hear}(z, \nu, \mathtt{say}(\nu, \nu, 0)) \end{array}\right) \tag{8}$$

There is still too much water, and some kittens still aren't inside boats! Venus! Finish the boats:

$$(\nu\, x)\left(\begin{array}{l} \quad (\mathtt{say}\ x\ z\ 0) \\ |\ \ (\mathtt{hear}\ x\ y\ (\mathtt{say}\ y\ x\ (\mathtt{hear}\ x\ y\ 0))) \\ |\ \ (\mathtt{hear}\ z\ \nu\ (\mathtt{say}\ \nu\ \nu\ 0)) \end{array}\right) \tag{9}$$

Venus! You're not done! Everything must be a kitten or a boat!

```
(channel x
  (par (say x z 0)
       (par (hear x y
              (say y x
                (hear x y 0)))
            (hear z v
             (say v v 0)))))
```

Hooray, all the kittens are safe and dry! But they can't nap, yet. Venus! Rearrange the boats so kittens can chat and then nap!

```
(channel x
  (par (par (say  x z 0)  ;; Oooh!, x's line up!
            (hear x y
                  (say y x
                       (hear x y 0))))
       (hear z v
             (say v v 0))))
```

This is great because there is a rule that says whenever a `say` and a `hear` line up their channels, rename, substitute and gobble up one `say` and its matching `hear`:

```
(channel x
  (par (par 0
            (say z x
                 (hear x y 0)))
       (hear z v
             (say v v 0))))
```

Darn it! Venus! Rearrange the `par` boats again, (it's always OK to do that):

```
(channel x
  (par 0
       (par (say  z x (hear x y 0))
            (hear z v (say  v v 0)))))
```

Substitute and gobble:

```
(channel x
  (par 0
       (par (hear x y 0)
            (say  x x 0))))
```

One more time:

```
(channel x
  (par 0 (par 0 0)))
```

Inside a `par` boat, it doesn't matter whether you write `hear` before `say` or `say` before `hear` — `par` is the captain and doesn't care; `par` is commutative. Also, because any number of napping kittens in `par` boats is equivalent to a all the kittens napping, write

```
(channel x 0)
```

Finally, because there is nothing to do with channel x, The whispering kitten can nap, too.

```
0
```

Thanks, Venus!

2.10 Kitten Boat Calculus

This is what Venus-the-boatwright had in mind whilst she built:

K, L	::=			
	|	`(nap)`	napping kitten	Do nothing; halt.
	|	`(hear` x y K`)`	listening kitten	Listen on channel x for channel y.
	|	`(say` x y K`)`	chatting kitten	Say "y" on channel x; don't wait.
	|	`(par` K L`)`	two kittens	Run K and L in parallel.
	|	`(channel` x K`)`	whispering kitten	fresh channel name x; use it in K.
	|	`(repeat` K`)`	mama cat	run copies of K forever.

(10)

3 Channels and Names

The kittens are named Kitten One, Kitten Two, and Kitten Three. These aren't names in kitten-speak, not names for channels like x and y. These are names in boat-speak, just so one doesn't write out the full boats over and over again.

Let's run some real code! For technical reasons, there is some punctuation — dots and quote marks here and there — for kittens written out in Clojure.

3.1 Kit-1

```
(def kit-1
  (say. 'x 'z (nap.)))
```

Notice that when `kit-1` eventually takes a nap, she's not saying or hearing anything. *The free names of `(nap)`, the names subject to substitution, are the empty set*:

```
(free-names (nap.))
```

```
#{}
```

In fact, the names that `kit-1` will eventually know about while napping, the *bound names, subject to renaming*, are also the empty set:

```
(bound-names (nap.))
```

```
#{}
```

Before she naps, Kitten One says z on x, so both those names are free for `kit-1`, meaning she just barks them out. They don't stand for anything else in potential suffixes of `kit-1`:

```
(free-names kit-1)
```

```
#{x z}
```

`Kit-1` doesn't wait for any names before nap-time, so her *bound names* are the empty set:

```
(bound-names kit-1)
```

```
#{}
```

3.2 Kit-2

Kitten Two listens on x for bound y, then says, on whatever y becomes, "x".

```
(def kit-2
  (hear. 'x 'y
   (say. 'y 'x
    (hear. 'x 'y (nap.)))))
```

We know that in her immediately-next say-prefix, `(say y x)`, y is a free variable and subject to substitution. It eventually becomes z, but *she* doesn't know so yet. She only knows that she will *eventually* know that y stands for z; y is eventually bound, thus bound.

```
(bound-names kit-2)
```

```
#{y}
```

Kitten Two's final activity is to listen on x for whatever-y-becomes. In that final activity, in isolation, she doesn't know whether she will ever know x, so the free variables — subject to substitution — of that final activity had better include x.

```
(do (def kit-2-final
      (hear. 'x 'y (nap.)))
    (free-names kit-2-final))
```

```
#{x}
```

By nap-time, she'll know what y stands for, but she won't use it while napping; y is eventually bound thus bound in her final activity:

```
(bound-names kit-2-final)
```

```
#{y}
```

In her next-to-last activity, which includes her last activity, she will know what y is, so it is bound:

```
(bound-names
 (say. 'y 'x
  kit-2-final))
```

```
#{y}
```

Kit-2 never uses x. She just passes x along, so it's free:

```
(free-names kit-2)
```

```
#{x}
```

3.3 Kit-3

Kitten Three listens on *z* for *v* — a temporary name — then says "*v*" on *v*: after substitution of something for *v*:

```
(def kit-3
  (hear. 'z 'v
   (say. 'v 'v (nap.))))
```

Her bound names include *v*, at least until it becomes free before substitution:

```
(bound-names kit-3)
```

```
#{v}
```

Her free names — subject to substitution — include *z*:

```
(free-names kit-3)
```

```
#{z}
```

Can you write down the free and bound names in her last activity, `(say 'v 'v)`? Here are spoilers:

```
(let [kit-3-last (say. 'v 'v (nap.))]
  (println (free-names kit-3-last))
  (println (bound-names kit-3-last)))
```

```
#{v}
#{}
```

3.4 Kitten Zero — the Whisper Boat

The bound names of Kitten Zero, captain of the Whisper Boat, include all the bound names of the other kittens, so had better be x for her own, y from Kitten Two, and *v* from kitten Three:

```
(do (def whisper-boat
      (channel. 'x
                (par. kit-1
                      (par. kit-2 kit-3))))
    (bound-names whisper-boat))
```

```
#{x y v}
```

Can you write out her free names? Here is a spoiler:

```
(free-names whisper-boat)
```

```
#{z}
```

The free names — subject to substitution — include only z from both Kitten One, who barks them out in `(say. 'x 'z)`, and Kitten Three, who listens on z for a substitution:

4 Change Log

2023-30-Mar :: weeding out the "we's"

2023-29-Mar :: Many small corrections.

2023-28-Mar :: Done building boats.

2023-26-Mar :: Current version.

2023-22-Mar :: Start.