A Wee Bit of Concurrency

Adrian King
26 January 2015

Your Comfort Zone?

- Many programmers today are comfortable with functional programming as the dominant paradigm.
- It's easier to write code this way because it sweeps *effects* under the rug—those annoying interactions with the real world or other other processes, or nonstandard flows of control.
- Many effects can be understood as phenomena of *concurrency*, where multiple things happen at once, or the order in which things happen is unpredictable.

… You're Leaving It.

- We can get pretty far trying to minimize or contain effects and ignore concurrency. Many programs are pure computations with limited interaction with the outside world.
- But some systems are all about interactions, and you can't get away with ignoring them.
- Functional programming has a tidy, well-understood theoretical foundation: the lambda calculus.
- There is no analog to the lambda calculus for concurrency. Theories of concurrency try to answer different questions and use different notations.

Part 0:
The Fool's Paradise

of Pure Functional Programming

The Paradise Part

- Traditionally, computer programmers write in imperative style—code is a series of commands that affect the world, or change the state of the program.
- Functional programming is the discipline of phrasing as much computation as possible in terms of pure (effect-free) functions.
- Regularities in impure real-world phenomena (I/O, concurrency, mutability) can often be described (mostly) in a pure way.
- Pure computations are usually easier to reason about than straightforward imperative code (once you learn the lingo), and easier to compose with one another.

Example: Mutability in Paradise

- An imperative program often maintains a current (mutable) state, updating the state destructively as it goes along.
- Instead, you can rephrase the update code as a (pure) state transition function between successive *immutable* states.
- This idea is the basis, e.g., for the Haskell State Monad.
- Monads play well with Haskell's other functional features, so this lets you do interesting things with state.

Another Example: Functional Reactive Programming

- Functional reactive programming tries to describe a dynamic system in terms of a static information flow through the system.

- When it works, this can make for much simpler code than manually operating on each datum as it passes through each part of the system.

The Fool Part

- Sometimes, describing a system in pure functional terms results in greater verbosity and less clarity, not the other way around.
- When a program's primary purpose is describing effects, functional programing can be a poor fit.
- What happens to the State Monad when multiple threads want to change the same thing?
- How should functional reactive programming handle networks of changing topology?

The Spaghetti Monad

- Haskell winds up putting hard-to-handle effects into the IO Monad.
- The expectation behind the Haskell IO Monad is that you don't put very much code in it.
- Otherwise, your code looks just like old-fashioned spaghetti code.

- Pure code that describes effects (like the IO Monad) requires an interpreter that actually performs the effects.
- Who writes that interpreter?

Foundations of the Fool's Paradise: the Lambda Calculus

- The lambda calculus is a way of describing computation in terms of functions.
- You can add lots of bells and whistles to the lambda calculus, but at the core are only three constructs:
  - Variables
  - Functions (or abstractions)
  - Applications (or function calls)
- The basic lambda calculus has no side effects, yet can compute anything that is computable.
- Pure functional programming languages can be defined in terms of a translation to some form of the lambda calculus.

Lambda Calculus Examples

$$x\ y$$

Apply the variable *x* to the variable *y*. (Neither variable is bound in an abstraction—both are *free*.)

$$\lambda f.\ \lambda x.\ f\ (f\ x)$$

An abstraction that takes two (curried) arguments—a function *f*, and a value *x*—and applies *f* twice to *x*.

$$\lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y$$

A function of three arguments that applies its last argument to the first two.

Confluence and Determinacy

- The lambda calculus is *confluent*, meaning that if an expression does not diverge, you can replace any subexpression with what it evaluates to, in any order—the final result is always the same.

Leaving Paradise: Concurrency and Indeterminacy

- In most concurrent systems, multiple transitions (leading to different outcomes) are possible from any given state (if you represent the system as a state machine).
- Different executions of concurrent programs may behave differently.

Concurrency vs. Parallelism

- The terms *concurrency* and *parallelism* are sometimes used interchangeably, but many authors prefer to distinguish between the two.
- Parallelism means things happening simultaneously.
- Concurrency means things happening (mostly) independently—maybe simultaneously, maybe not.

Concurrency at the Lowest Level

- There are many techniques to organize concurrent programs: server-client relationships, thread pools, futures, parallelized collections, and so on.
- There are many techniques to coordinate concurrent work: locks, semaphores, atomically updated values, concurrent maps and queues, and so on.
- We won't talk about *any* of those here.
- Instead, we're looking at the ways people have tried to construct the simplest possible languages in which those ideas could be expressed.

Surgeon General's Warning

Concurrent programming is complicated. Debugging concurrent programs can be baffling. Scheduling concurrent activities entails mutable state, at least in the scheduler. Sharing access to mutable state is error-prone. Incorrect management of shared concurrent resources can lead to deadlock, starvation, or livelock. Multiple processes can consume more memory and CPU time than single processes. Do not make explicit use of concurrency when other programming techniques are available. If you have concluded that you *must* write concurrent code, check again whether there is really no alternative. Then check again.

Part 1:

Formalisms

Formal Descriptions

- Some of the descriptions of concurrency we'll look at are formally specified. Some aren't.
- Our favorite sort of formalism specifies a *syntax* for a language that describes something (in this case, the state of a concurrent system), along with *rules* that let you derive new somethings from that something (in this case, later states of the same concurrent system).
- The lambda calculus is such a formalism.

Syntax and Rules

- Syntax looks like highly abbreviated BNF (nonterminal names are generally one letter):

$$
\begin{array}{ll}
e ::= & x \\
 & \lambda x.\ e \\
 & e_1\ e_2
\end{array}
$$

- Subscripts distinguish different occurrences of the same production.

- Rules are written with premises above a line, and a conclusion below:

$$\frac{}{x \Downarrow x}$$

$$\frac{e_1 \Downarrow e_2}{\lambda x.\ e_1 \Downarrow \lambda x.\ e_2}$$

$$\frac{e_1 \Downarrow \lambda x.\ e_3 \qquad e_2 \Downarrow e_4}{e_1\ e_2 \Downarrow [e_4/x]e_3}$$

- You can derive the conclusion whenever the premises hold.

Formalisms Are about Rewriting

- Formal rules define ways in which you can rewrite symbolic expressions.
- That is the only form of execution we consider here.
- For example, in arithmetic, you can rewrite:

  (1 + 2) + (3 + 4)

  as:

  3 + 7

  or:

  (3 + 4) + (1 + 2)

  according to rules phrased in terms only of symbol manipulation.

Rules of the Untyped Lambda Calculus

- You may have recognized the earlier syntax and evaluation rules as those of the untyped lambda calculus.

$e ::=$	x	variable
	$\lambda x.\ e$	function
	$e_1\ e_2$	application

- e is an expression; x is a variable.

$$\frac{}{x \Downarrow x}$$

A variable evaluates to itself.

$$\frac{e_1 \Downarrow e_2}{\lambda x.\ e_1 \Downarrow \lambda x.\ e_2}$$

Optional rule: a function may be optimized.

$$\frac{e_1 \Downarrow \lambda x.\ e_3 \qquad e_2 \Downarrow e_4}{e_1\ e_2 \Downarrow [e_4/x]e_3}$$

Apply a function by substituting the argument in the function body.

Substitution

- The substitution rule used in the evaluation of applications has some subtleties.

- The notation $[e_2/x]e_1$ or $[x \mapsto e_2]e_1$ means the result of substituting e_2 for x wherever x occurs in e_1, after renaming variables as necessary to avoid conflicts.

- Conflict example:

$$[z/x](\lambda z.\ x) \;\Rightarrow\; \lambda z.\ z$$

The function $\lambda z.\ x$ always returns x—substitution shouldn't turn it into the identity function! Rename it $\lambda y.\ x$.

Part 2:

The Pi Calculus

Why Pi?

- I'm introducing the pi calculus first not because it is necessarily the best of the process calculi, but because:
  - It's economical
  - It's relatively recent (Robin Milner et al., 1992), so lacks some of the quirks of earlier systems
  - The notation and reduction rules are not too strange if you're familiar with the lambda calculus
  - It is easy to translate the lambda calculus into it
  - It supports the key notion of *mobility*: a network of communicating processes does not have to have a fixed topology

Pi Calculus Syntax

$P ::=$	$P_1 \mid P_2$	Run processes P_1 and P_2 in parallel
	$a(x).P$	Wait for input value on channel a; bind it to x in P
	$\bar{a}\langle x\rangle.P$	Write x's value to channel a, then run P
	$(\nu a)P$	Create new channel a; a can appear in P
	$!P$	Run infinitely many copies of P
	0	Do nothing

- P ranges over processes.
- a and x both range over channel identifiers, but a emphasizes the identifier's role as a channel, whereas x emphasizes the role as a bound variable.
- Binders: $a(x).P$ binds x; $(\nu a)P$ binds a.
- Informally, trailing $.0$ is sometimes omitted.

ν Is New

- The ν operator binds its variable argument to a newly created channel (which is different from any existing channel).
- The static scope of the variable bound by ν is limited to the immediately following process, but the newly allocated channel persists indefinitely.
- ν is not like anything in the lambda calculus. Unlike values in functional languages, channels are distinguished not by what they contain but by when they are allocated.
- Think pointers, or maybe gensyms.

Test Driving the Pi Calculus

- Read this: $\bar{a}\langle b\rangle.0 \mid a(x).Q$

  as "there are two processes running in parallel: the first sends b on channel a, then stops; the second reads from channel a and binds the result to the variable x, then runs some process Q (which may refer to x)".

- So what does this do?:

  $(\nu a)(\bar{a}\langle x\rangle$
  $\quad \mid a(\_).\bar{e}\langle y\rangle \mid a(\_).\bar{u}\langle z\rangle)$

- How about this one?:

  $(\nu a)(\bar{a}\langle i\rangle \mid a(e).\bar{e}\langle a\rangle.a(e))$
  $\mid i(u).\bar{u}\langle u\rangle$

Reduction Rules for the Pi Calculus

$$\frac{}{\bar{a}\langle b\rangle.P \mid a(c).Q \;\rightarrow\; P \mid [b/c]Q}$$

If one process is writing on a channel another is reading, substitute the value read for the variable to which the value is bound

$$\frac{P \;\rightarrow\; Q}{P \mid R \;\rightarrow\; Q \mid R}$$

Reduce parallel processes independently

$$\frac{P \;\rightarrow\; Q}{(\nu a)P \;\rightarrow\; (\nu a)Q}$$

Adding unused channels changes nothing

$$\frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

Congruent processes reduce to congruent processes

where $A \rightarrow B$ means "A reduces to B", and the structural congruence relation $\equiv$ is defined by:

- $P \equiv Q$ if renaming bound variables in P yields Q
- $P \mid Q \equiv Q \mid P$, $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, and $P \mid 0 \equiv P$
- $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$ and $(\nu a)0 \equiv 0$
- $!P \equiv P \mid !P$
- $(\nu a)(P \mid Q) \equiv ((\nu a)P) \mid Q$ if a is not free in Q

Encoding Values in the Pi Calculus

- As with the raw lambda calculus, encoding values requires ingenuity—we have none of the built-in data types of a typical programming language.
- We represent all values in the pi calculus as channels.
- Pi calculus processes are operations on values.
- This is a little bit like the Church encoding of arithmetic in the lambda calculus, where functions are values, and expressions compute values.

Sending Tuples in the Pi Calculus

- It is more work to get something useful done in the pi calculus than in the lambda calculus. Often, you need to send more than one value to a pi calculus process for something interesting to happen, so you need to send tuples.

- Sending a pair (x,y) to a channel a uses an auxiliary channel e:

$$(\nu e)(\bar{a}\langle e\rangle.\bar{e}\langle x\rangle.\bar{e}\langle y\rangle)$$

- The recipient code listening to a reads the pair from e:

a(e).e(x).e(y). … *do something with x and y* …

Encoding Functions in the Pi Calculus

- To get the effect of a function application in the pi calculus, you must send the callee a place to return control to as well as the function argument.
- That is, pi calculus functions are always written in continuation-passing style, because all transfers of control are explicit.
- Also, you must use ! in a function body to allow it to run arbitrarily many times—otherwise it runs just once.

```
(νfact)(! (fact(args).args(n).args(e).
           if n = 1 then ē⟨1⟩
           else (νu)(
                    (νa)(fāct⟨a⟩.ā⟨n – 1⟩.ā⟨u⟩)
                  | u(n′).ē⟨n * n′⟩))
        … do something with fact … )
```

Lambda in Pi

- To encode a lambda calculus expression into the pi calculus, transform all abstractions and applications into continuation-passing style, as on the previous slide.
- Free variables are just free channels.
- You must also, of course, figure out encodings for useful things like booleans, integers, etc.
- Enjoy!

Part 3:

The Actor Model

Why Actors?

- Actors have been around a while (Carl Hewitt, Peter Bishop, and Richard Steiger, 1973).
- The model is presented in fairly intuitive terms.
- A number of languages (like Erlang) and libraries (like Scala's Akka) have been based on the actor model and are in reasonably widespread use.
- The actor model description offers an interesting contrast with the pi calculus:
  - Actor communication is asynchronous
  - Behavior is specified locally (per actor), instead of reduction rules being applied globally (to a whole system)

Acting 101

- The actor model is not usually thought of as a self-contained low-level language, like the pi calculus, but as an API along with a set of conventions and guarantees implementable in various languages.
- Each actor is associated at any given moment with a behavior, which is a function* that takes a message and:
  - Creates new actors;
  - Sends messages to other actors;
  - And/or specifies the behavior to associate with the actor for processing the next message.
- An actor is thus a sequential* process.

* Unless it isn't, depending on the implementation.

Asynchrony

- A key difference between the actor model and the pi calculus is that actor messages are delivered asynchronously.
- Recall the pi calculus message-sending reduction rule:

$$\frac{}{\bar{a}\langle b\rangle.P \mid a(c).Q \;\rightarrow\; P \mid [b/c]Q}$$

- This specifies an instantaneous transition of the system from a state before the message is sent to one where the message has been received. The message spends no time in transit.
- The actor model, by contrast, does not specify how long a message might spend in transit.

Message Ordering and Local Time

- If you don't know how long a message might spend in transit, then it's hard to guarantee an order in which messages are delivered. Consequently, the actor model provides no such guarantee.
- Actors don't necessarily agree on what time it is on a global clock, or what order things happen in. As in Einsteinian relativity, actors must agree on causality (events caused by the sending of a message cannot precede the message send), but otherwise keep their own time.

Locality and Mobility

- As in the pi calculus, messages may contain places to which messages can be sent. That is, an actor (or at least an actor's mailbox) can be part of a message.
- In fact, the only other actors an actor knows about are the ones it has been told about or that it has itself created.
- So the mobility rules for the actor model are much like those of the pi calculus: a pi calculus process also knows only channels in its environment, channels it reads, or channels it creates with ν.

Actors vs. Channels

- Compared with the actor model, the pi calculus uses a layer of indirection in message sending.
- The actor model sends directly to an actor, while the pi calculus sends to a channel on which zero or more processes may be listening.
- You can use actors to simulate the message-passing conventions of the pi calculus by designating some actors as pi calculus processes and others as pi calculus channels. Process actors that listen on a channel send a message to the corresponding channel actor to say that they are ready to receive on the channel.

Pi Flavor vs. Actor Flavor

- The pi calculus is defined more precisely than the actor model. Different implementations of the actor model embody slight changes in semantics from one another.
- The layer of indirection provided by channels is sometimes more flexible than the actor model's convention of hardwiring a target actor.
- Many concurrent programs can be expressed equally well by either model.
- Either model is satisfactory for encoding the lambda calculus.

Part 4:

Sync vs. Async

To Synchronize or Not to Synchronize

- The distinction between synchronous messaging in the pi calculus and asynchronous messaging in the actor model is profound.

- The pi calculus I presented earlier is also known as the *synchronous* pi calculus.
- There also exists an *asynchronous* pi calculus.
- Comparing the two variants of the pi calculus illustrates the tradeoffs between synchrony and asynchrony.

The Applicability of Asynchrony

- The reduction semantics shown earlier for the synchronous pi calculus are easy to implement in a single thread of execution.
- But single-threadedness hardly matches our idea of what concurrency is all about. We expect concurrency formalisms to work in a distributed system (one without global synchronization or coordination).
- Asynchronous message sending is the kind we expect (and can easily implement) in a distributed system.
- Most work on implementing the pi calculus for practical purposes concentrates on the asynchronous version.

Asynchronous Pi Calculus Mods

- To change the synchronous pi calculus into the asynchronous pi calculus, we change just one piece of syntax and *leave the reduction rules the same*:

~~$\bar{a}\langle x\rangle.P$~~	~~Write x's value to channel a, then run P~~
$\bar{a}\langle x\rangle$	Write x's value to channel a, then stop

- To get an effect similar to the original synchronous syntax, writes must occur *in parallel* with something else—that is, $\bar{a}\langle x\rangle \mid P$ rather than $\bar{a}\langle x\rangle.P$.
- This lets the system deliver messages whenever it wants, so it is simpler to actually build a distributed version of the asynchronous pi calculus than the synchronous version.

Choice

- Some formulations of the pi calculus include a *choice* operator:

  $$P_1 + P_2$$

  which runs either P_1 or P_2.

- In an asynchronous context, it is hard to implement this operator in a well-behaved way.
- Input-guarded choice is easier to implement asynchronously in a well-behaved way:

  $$i_1(x_1).P_1 + i_2(x_2).P_2$$

- The sense in which these operators are well- or ill-behaved is rather technical. Another lecture?

Another Common Pi Calculus Mod

- Many implementations based on the pi calculus make a further restriction on the original pi calculus syntax, replacing unrestricted replication with replication guarded by a read:

~~!P~~	~~Run infinitely many copies of P~~
!a(x).P	Wait for input value on channel a; bind it to x in P; then, in parallel, run P and listen again on a

- The original unrestricted replication would let you spawn infinitely many *writes* in parallel, which is not useful in practice.

Sync > Async

- There is a (rather technical) sense in which the synchronous pi calculus is strictly more powerful than the asynchronous. Intuitively, there is more control over the order in which things happen in the synchronous calculus.
- One obvious consequence for us of moving to the asynchronous calculus is that the encoding of functions used in the synchronous calculus no longer works—it relies on our (now illegal) encoding of tuples:

$$(\nu e)(\bar{a}\langle e\rangle.\bar{e}\langle x\rangle.\bar{e}\langle y\rangle)$$

which in turn relied on the order of message delivery, which is indeterminate in the asynchronous calculus.

Polyadic Pi

- The usual remedy for the inability of the asynchronous calculus to pass tuples around is to redefine channel reads and writes to build in communication of tuples:

~~$a(x).P$~~	~~Wait for input value on channel a; bind it to x in P~~
~~$\bar{a}\langle x\rangle.P$~~	~~Write x's value to channel a, then run P~~
$a(x_1, \ldots , x_n)$	Wait for input values on channel a; bind them to x_i in P
$\bar{a}\langle x_1, \ldots , x_n\rangle.P$	Write x_i's values to channel a, then run P

  although this means you must also change the corresponding reduction rules to match only tuples of the same length.

- An implementation of the pi calculus in an existing programming language can use the underlying language's tuples (and other data structures).

Zip-Locking: Tuples without Polyady

- In fact, the polyadic pi calculus is just syntactic sugar. It is possible to encode tuples in the asynchronous calculus using a technique Martin Odersky named *zip-locking*:
  - Send a pair (a,b) on channel p:

$$(\nu u)(\bar{p}\langle u\rangle \mid u(e).(\bar{e}\langle a\rangle \mid u(i).\bar{\imath}\langle b\rangle))$$

  - Receive (a,b) on channel p, then proceed as P:

$$p(u).(\nu e)(\bar{u}\langle e\rangle \mid e(a).(\nu i)(\bar{u}\langle i\rangle \mid i(b).P))$$

- In the example, u is an auxiliary channel created by the sender, and e and i auxiliary channels created by the receiver.

Actors and Asynchronous Pi

- Although there are various formulations of both the actor model and the asynchronous pi calculus, you can usually come up with a way to translate each to the other.
- This is because both models:
  - share the property of mobility
  - have an asynchronous model of communication

Pict

- The experimental programming language Pict (Benjamin Pierce and David Turner, 1997) is an attempt to turn the pi calculus into a real-world programming language, and to make it manageable by imposing a type system on it.
- The syntax is far more keyboard-friendly than the original asynchronous pi calculus syntax:

$P_1 \mid P_2$	(P1 | P2)
$a(x).P$	a ? x = P
$\bar{a}\langle x \rangle$	a ! x
$(\nu a)P$	new a: *type* P
$!a(x).P$	a ?* x = P

- The tutorial (easy to find online) is well worth a read if you want to learn to think how to compose practical code in the pi calculus.

Part 5:
A Quick Preview

of Joins and Ambients

The Join Calculus: Better Living through Chemistry

- The join calculus (Fournier and Gonthier, 1996) uses the metaphor of chemical reactions to describe how messages can combine to produce results (and new messages).
- The join calculus can be translated to the asynchronous pi calculus and vice versa.
- JoCaml is an OCaml dialect with support for the join calculus. A tiny sample reaction:

  ```
  a() & b() = c() & d()
  ```

  which, when reagents a and b are available in the "chemical soup", consumes them and sends c and d as new reagents (messages) into the soup.

The Ambient Calculus: Where It's At

- Whereas the pi calculus is interested in which processes can talk to which other processes,the ambient calculus (Cardelli and Gordon, 1998) concerns itself with *where* processes execute.
- Ambients may be thought of as data or processes, or places where processes execute.
- Ambients are also administrative domains with which capabilities are associated to control their movement.
- Researchers interested in different capabilities have proposed a number of variations on the ambient calculus.

Part 6:
Concurrency Formalisms

That Didn't Make the Cut

Petri Nets

- Petri nets were devised in 1939 by Carl Adam Petri to model chemical reactions. In his 1962 doctoral dissertation, he advocated their use to model concurrent behavior more generally.
- Petri nets are state transition systems based on graphs, with well-defined rules and a large body of literature.
- The presence or absence of marks at nodes control which state transitions are eligible to fire.
- Despite their long history, it is hard to see how to map Petri net concepts to modern software facilities.

Communicating Sequential Processes

- Introduced by Tony Hoare in 1978, CSP was originally a programming language proposal; mathematically well-defined semantics and a more concise notation came later.
- By modern standards, CSP, although reasonably expressive for an immobile formalism, is not particularly minimal.

$Proc :=$	STOP	
	SKIP	
	$e \rightarrow Proc$	prefixing
	$Proc \;\Box\; Proc$	external choice
	$Proc \sqcap Proc$	nondeterministic choice
	$Proc \;\vert\vert\vert\; Proc$	interleaving
	$Proc \;\vert[\{X\}]\vert\; Proc$	interface parallel
	$Proc \setminus X$	hiding
	$Proc \,;\, Proc$	sequential composition
	if b then $Proc$ else $Proc$	boolean conditional
	$Proc \triangleright Proc$	timeout
	$Proc \vartriangle Proc$	interrupt

CSP syntax, per Wikipedia

Calculus of Communicating Systems

- Robin Milner, 1980.
- No notion of mobility, but more economical than CSP:

$P ::=$	0	nothing
	$a.P$	perform action a, then continue as P
	$A = P$	(possibly recursive) definition of A as P
	$P_1 + P_2$	either P_1 or P_2
	$P_1 \mid P_2$	P_1 and P_2 simultaneously
	$P\,[a_2/a_1]$	substitution of a_2 for a_1 in P
	$P_1 \setminus a$	P_1 without action a

- CCS is the immediate predecessor of the pi calculus.

Algebra of Communicating Processes

- Jan Bergstra and Jan Willem Klop, 1982.
- Uses familiar algebraic symbols $+$ and $\cdot$ to represent alternative and successive actions. $||$ represents parallel composition; $||\!\lfloor$ a sort of serialized merge, where the first action of the left term occurs first.
- There are no general-purpose communication channels, but the $|$ operator is used to separate two processes that are communicating.
- Like many of the earlier process calculi, the notation seems more focused on the mere fact of interprocess communication than what is being communicated.

Part 7:

The Future

To Be Continued…

- Concurrency is a bigger field than I had realized.
- I expect to explore more process calculi and their variants in future lectures.
- In particular, I'd like to explore the join calculus in some detail, and explain how it translates into the pi calculus and vice versa.
- I'd like to learn more about the ambient calculus. (Who wouldn't?)

And Some Future Topics on Concurrency in General

- What does it mean for a concurrent system to be correct?
- What does it mean for two processes to be equivalent?
- What guarantees can be made about delivery of messages in a concurrent system?
- What does “type system” mean in the context of concurrency, and what concurrent type systems have been studied?
- What kinds of connections are there between concurrent programming, functional programming, and logic programming?