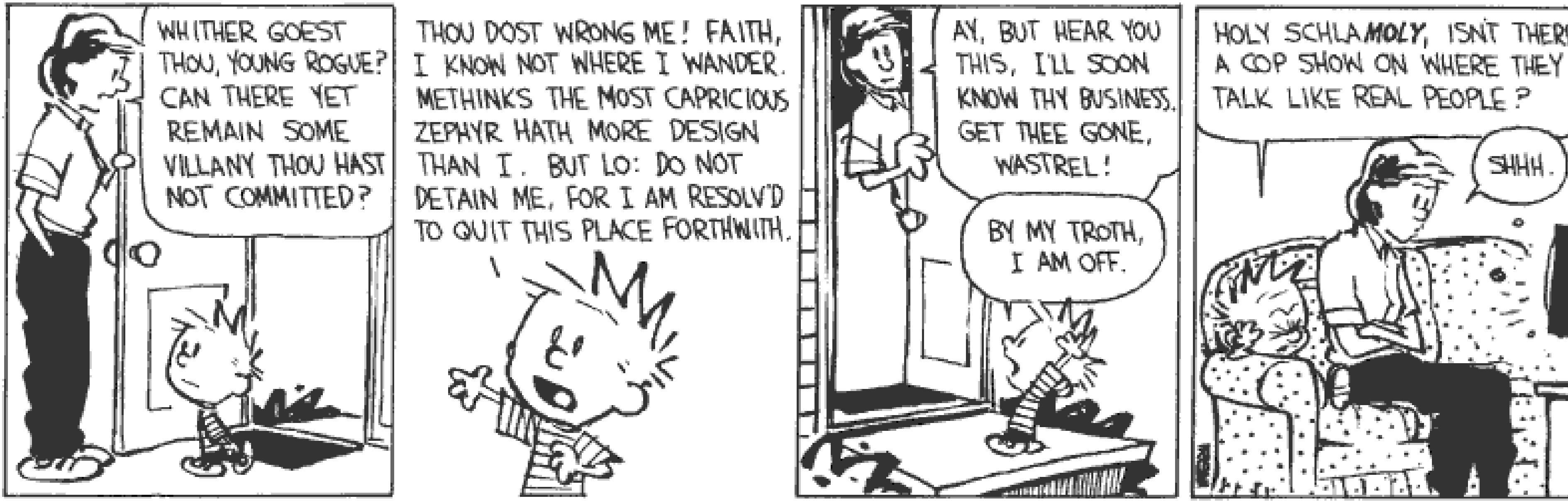
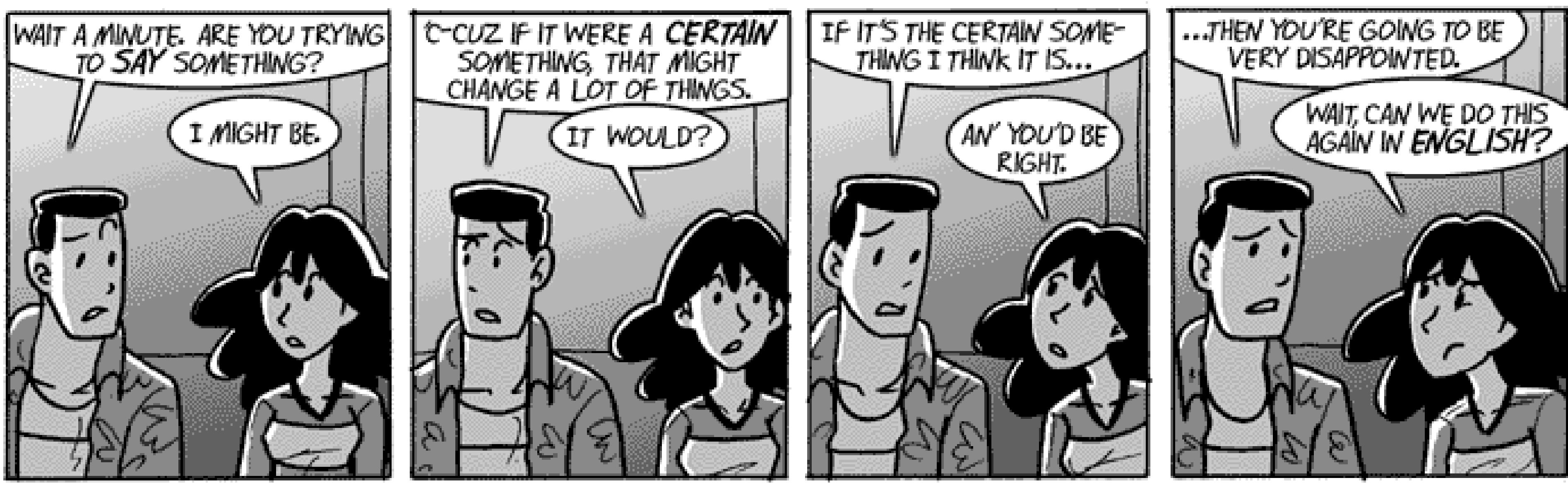


Communication and Concurrency



Preliminary Definition

- A calculus is a *method or system of calculation*
- The early Greeks used pebbles arranged in patterns to learn arithmetic and geometry
- The Latin word for pebble is “calculus” (diminutive of calx/calcis)
- Popular flavors:
  - differential, integral, propositional, predicate, lambda, pi, join, of communicating systems

Cunning Plan

- Types of Concurrency
- Modeling Concurrency
- Pi Calculus
- Channels and Scopes
- Semantics
- Security
- Real Languages

Relevance – PLDI 2015

Mechanized Verification of Fine-Grained Concurrent Programs

Ilya Sergey

In this paper, we focus on *program logics* as a generic approach to specify a program and formally prove its correctness *wrt.* the given specification. In such logics, program specifications (or specs) are represented by Hoare triples $\{P\}\ c\ \{Q\}$, where c is a program being described, P is a precondition that constrains a state in which the program is safe to run, and Q is a postcondition,

Anindya Banerjee

Asynchronous Programming, Analysis and Testing with State Machines

Pant

$$\frac{\neg\ell(v)}{(\ell, h, S, \mathsf{if}\ (v)\ ss_t\ \mathsf{else}\ ss_f;\ ss) \to_s (\ell, h, S, ss_f;\ ss)}$$

$$\frac{\ell(v)}{\begin{array}{l}(\ell, h, S, \mathsf{while}\ (v)\ ss_b;\ ss)\\ \quad \to_s (\ell, h, S, ss_b;\ \mathsf{while}\ (v)\ ss_b;\ ss)\end{array}}\ (\text{WHI}$$

$$\frac{\neg\ell(v)}{(\ell, h, S, \mathsf{while}\ (v)\ ss_b;\ ss) \to_s (\ell, h, S, ss)}\ (\text{WHIL}$$

Figure 3. Operational semantics

$$\frac{\begin{array}{c}M(i) = (m, q, E, \ell, S, \mathsf{send}_{dst}\ evt(v);\ ss)\\ M_s = M[i \mapsto (m, q, E, \ell, S, ss)]\\ M_s(dst) = (m', q', E', \ell', S', ss')\\ M' = M_s[dst \mapsto (m', q', E' : evt(\ell(v)), \ell', S', ss')]\end{array}}{(h, M) \to_t (h, M')}\ (\text{SEND})$$

$$\frac{\begin{array}{c}M(i) = (m, q, E, \ell, S, \varepsilon) \qquad T_m(q, E) = (q', val, E')\\ M' = M[i \mapsto (m, q', E', \ell, S, v_m.q'(val))]\end{array}}{(h, M) \to_t (h, M')}\ (\text{RECEIVE})$$

Figure 4. Transition rules

Relevance – PLDI 2015

Efficient Synthesis of Network Updates

starting at src eventually reach dst. Temporal logics are an expressive and well-studied language for specifying such trace-based properties. Hence, we use Linear Temporal Logic (LTL) to describe traces in our network model. Let AP be atomic propositions that test the value of a switch, port, or packet field: $f_i = n$. We call elements of the set 2^{AP} *traffic classes*. Intuitively, each traffic class T identifies a set of packets that agree on the values of particular header fields. An LTL formula φ in negation normal form (NNF) is either $true$, $false$, atomic proposition p in AP, negated proposition $\neg p$, disjunction $\varphi_1 \vee \varphi_2$, conjunction $\varphi_2 \wedge \varphi_2$, next $X\varphi$, until $\varphi_1 U \varphi_2$, or release $\varphi_1 R \varphi_2$, where φ_1 and φ_2 are LTL formulas in NNF. The operators F and G can be defined using other

4.3 Formal Properties

The following two theorems show that our algorithm is sound for careful updates, and complete if we limit our search to *simple* update sequences (see Appendix B for proofs).

Theorem 1 (Soundness). *Given initial network N_i, final configuration N_f, and LTL formula φ, if* ORDERUPDATE *returns a command sequence $cmds$, then $N_i \xrightarrow{cmds} N'$ s.t. $N' \simeq N_f$, and $cmds$ is correct with respect to φ and N_i.*

Theorem 2 (Completeness). *Given initial network N_i, final configuration N_f, and specification φ, if there exists a simple, careful sequence $cmds$ with $N_i \xrightarrow{cmds} N'$ s.t. $N' \simeq N_f$, then* ORDERUPDATE *returns one such sequence.*

Verdi: A Framework for Implementing and Formally Verifying Distributed Systems

```
| Agent n =>
  match msg with
    | GrantMsg =>        (* lock acquired    *)
      s := true;;        (* update state     *)
      output Grant       (* notify listeners *)
    | _ => nop           (* never happens    *)
```

Figure 3. A simple lock service application implemented in Verdi, under the assumption of a reliable network. Verdi extracts these definitions into OCaml and links the resulting code with a runtime to send and receive messages over the network.

g Woos Pavel Panchekha
ichael D. Ernst Thomas Anderson

[35] D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521781779.

Take-Home Message

- The pi calculus is a formal system for modeling concurrency in which "communication channels" take center stage.
- Key concerns include non-determinism and security. The pi calculus models synchronous communication. Can someone eavesdrop on my channel?

Possible Concurrency

- No Concurrency
- Threads and Shared Variables
  - A language mechanism for specifying interleaving computations; often run on a single processor
- Parallel (SIMD)
  - A single program with simultaneous operations on multiple data (high-perf physics, science, ...)
- Distributed processes
  - Code running at multiple sites (e.g., internet agents, DHT, Byzantine fault tolerance, Internet routing)
- Different research communities $\Rightarrow$ different notions

(There Must Be) Fifty Ways to Describe Concurrency

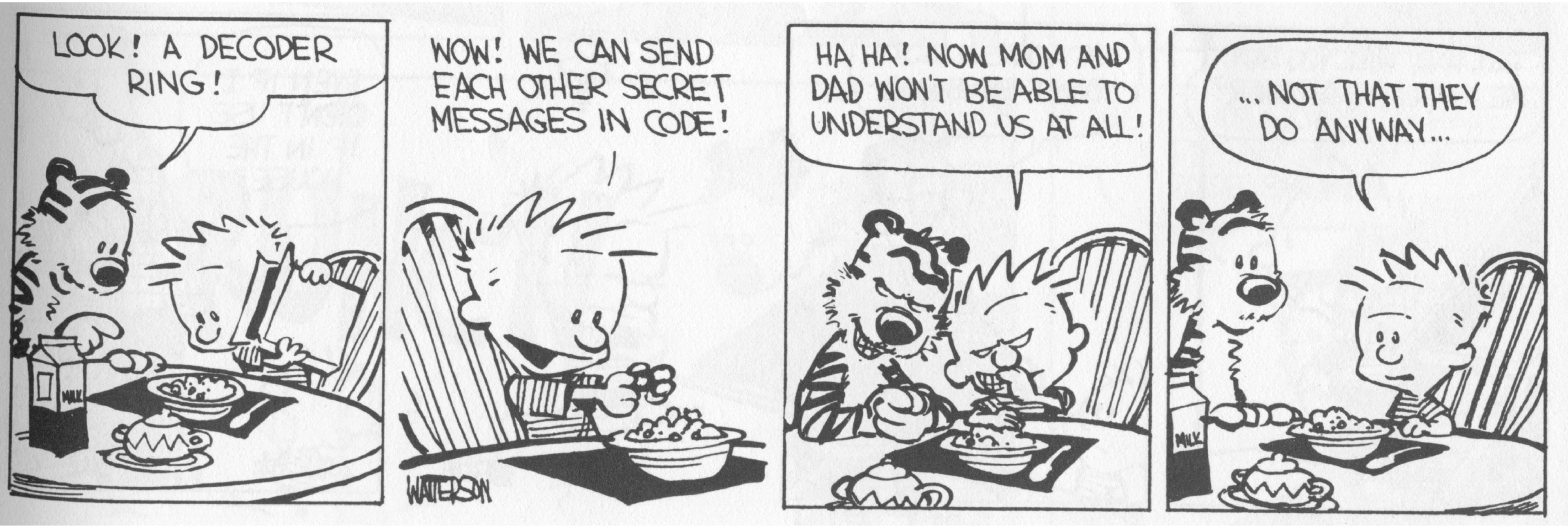
- No Concurrency
  - Sequential processes are modeled by the λ-calculus. Natural way to observe an algorithm: examine its output for various inputs $\Rightarrow$ functions
- Threads and Shared Variables
  - Small-step opsem with contextual semantics (e.g., callcc), or special type systems (e.g., [FF00])
- Parallel (SIMD)
  - Not in this class (e.g., Titanium, etc.)
- Distributed processes
  - ???

Modeling Concurrency

- Concurrent systems are naturally non-deterministic
  - Interleaving of atomic actions from different processes
  - New concurrent scheduling possibly yields new result
- Concurrent processes can be observed in many ways
  - When are two concurrent systems equivalent?
  - Intra-process behavior vs. inter-process behavior
- Concurrency can be described in many ways
  - **Process creation**: fork/wait, cobegin/coend, data parallelism
  - **Process communication**: shared memory, message passing
  - **Process synchronization**: monitors, semaphores, transactions

Message Passing

- These “many ways” lead to a variety of process calculi
- We will focus on message passing!



Communication and Messages

- Communication is a fundamental concept
  - But not for everything (e.g., not much about parallel or scientific computing in this lecture)
- Communication through message passing
  - synchronous or asynchronous
  - static or dynamic communication topology
  - first-order or high-order data
- Historically: Weak treatment of communication
  - I/O often not considered part of the language
- Even “modern” languages have primitive I/O
  - First-class messages are rare
  - Higher-level remote procedure call is rare

Calculi and Languages

- Many calculi and languages use message-passing
  - Communicating Sequential Processes (CSP) (Hoare, 1978)
  - Occam (Jones)
  - Calculus of Communicating Systems (CCS) (Milner, 1980)
  - The Pi Calculus (Milner, 1989 and others)
  - Pict (Pierce and Turner)
  - Concurrent ML (Reppy)
  - Java RMI
- Messaging is built in some higher-level primitives
  - Remote procedure call
  - Remote method invocation

The Pi Calculus

- The pi calculus is a process algebra
  - Each process runs a different program
  - Processes run concurrently
  - But they can communicate
- Communication happens on channels
  - channels are first-class objects
    - channel names can be sent on channels
  - can have access restrictions for channels
- In λ-calculus everything is a function
- In Pi calculus everything is a process

Pi Calculus Grammar

- Processes communicate on channels
  - **c<M>** *send message M on channel c*
  - **c(x)** *receives message value x from channel c*
- Sequencing
  - **c<M>.p** *sends message M on c, then does p*
  - **c(x).p** *receives x on c, then does p with x (x is bound in p)*
- Concurrency
  - **p | q** *is the parallel composition of p and q*
- Replication
  - **! p** *creates an infinite number of replicas of p*

Examples

- For example we might define

  Speaker = air<M> *// send msg M over air*
  Phone = air(x).wire<x> *// copy air to wire*
  ATT = wire(x).fiber<x> *// copy wire to fiber*
  System = Speaker | Phone | ATT

- Communication between processes is modeled by reduction:

  Speaker | Phone → wire<M> *// send msg M to wire*
  wire<M> | ATT → fiber<M> *// send msg M to fiber*

- Composing these reductions we get

  Speaker | Phone | ATT → fiber<M> *// send msg M to fiber*

Channel Visibility

- Anybody can monitor an unrestricted channel!
- Modeling such snooping:

  WireTap = wire(x).wire<x>.NSA<x>

  – Copies the messages from the wire to NSA
  – Possible since the name “wire” is globally visible
- Now the composition:

  WireTap | wire<M> | ATT $\rightarrow$

  wire<M>.NSA<M> | ATT $\rightarrow$

  NSA<M> | fiber<M> *// OOPS !*

Restriction

- The restriction operator $(\nu c)\ p$ makes a fresh channel c within process p
  - ν is the Greek letter "nu"
  - The name c is local (bound) in p
  - c is not known outside of p
- Restricted channels *cannot be monitored*

  wire(x) … | (ν wire)(wire<M> | ATT) $\rightarrow$

  wire(x) … | fiber<M>
- The scope of the name wire is restricted
- There is no conflict with the global wire

Restriction and Scope

- Restriction
  - is a binding construct (like λ, $\forall$, $\exists$, ...)
  - is lexically scoped
  - allocates a new object (a new channel)
  - somewhat like Unix pipe(2) system call

$(\nu c)p$ is like let c = new Channel() in p

- c can be sent outside its initial scope
  - But only if p decides so (intentional leak)

First-Class Channels

- Channel c can leave its scope of declaration
  - via a message d<c> from within p
  - d is some other channel known to p
  - Intentional with "friend" processes (e.g., send my IM handle=c to a buddy via email=d)
- Allowing channels to be sent as messages means communication topology is dynamic
  - If channels are not sent as messages (or stored in the heap) then the communication topology is static
  - This differentiates Pi-calculus from CCS

Example of First-Class Channels

Consider:

MobilePhone = air(x).cell<x>

ATT1 = wire<cell>

ATT2 = wire(y).y(x).fiber<x>

y will be bound to cell!

in

(ν cell)(MobilePhone | ATT1) | ATT2

- ATT1 passes cell out of the static scope of the restriction ν cell

Q: Books (734 / 842)

- Name either the Martian protagonist or the Martian word for "to drink" in Robert Heinlein's 1961 sci-fi novel **Stranger in a Strange Land**. The novel won the Hugo award and the word has entered the OED.

Q: General (485 / 842)

- In the works **Treatise on the Human Being** and **Discourse on the Method** (1637) Descartes considers a theory in which the soul is like a little person that sits inside the brain to observe and direct. Name the little person or the gland most closely associated with this theory. Optionally, translate “*je pense*, *donc je suis*”, which first appears in DoTM.

Scope Extrusion

- A channel is just a name
  - First-class names must be usable in any scope
- The pi calculus restrictions distribute:

  $((\nu\ c)\ p)\ |\ q\ =\ (\nu\ c)(p\ |\ q)$ *if c not free in q*

- Renaming is needed in general:

  $$((\nu\ c)\ p)\ |\ q = ((\nu\ d)\ [d/c]\ p)\ |\ q$$
  $$= (\nu\ d)([d/c]\ p|\ q)$$

  where "d" is fresh (does not appear in p or q)

- This scope extrusion distinguishes the pi calculus from other process calculi

Syntax of the Pi Calculus

There are many versions of the Pi calculus

A basic version:

p,q ::=	*(p and q are processes)*
nil	*nil process (sometimes written 0)*
x<y>.p	*sending data y on channel x*
x(y).p	*receiving data y from channel x*
p | q	*parallel composition*
!p	*replication*
(ν x)p	*restriction (new channel x used in p)*

- Note that only variables can be channels and messages

Operational Semantics

- One basic rule of computation: data transfer

$$\frac{}{x\langle y\rangle.p \mid x(z).q \;\rightarrow\; p \mid [y/z]q}$$

  - Synchronous communication: 1 sender, 1 receiver
  - Both the sender and the receiver proceed afterwards
- Rules for local (non-communicating) progress:

$$\frac{p \rightarrow p'}{p \mid q \rightarrow p' \mid q} \qquad \frac{p \rightarrow p'}{(\nu x)p \rightarrow (\nu x)p'}$$

$$\frac{p \equiv p' \quad p' \rightarrow q' \quad q' \equiv q}{p \rightarrow q}$$

Structural Congruence

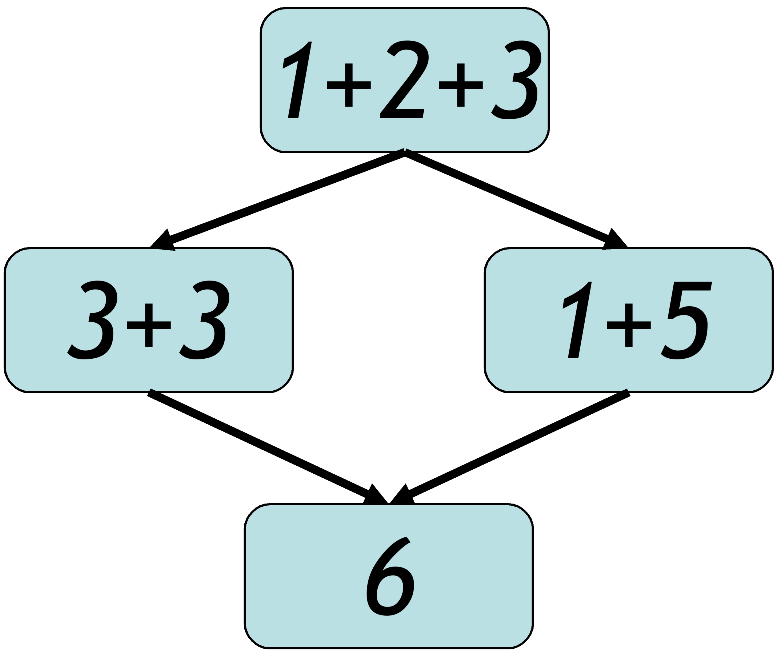
$$\frac{}{p \equiv p} \qquad \frac{q \equiv p}{p \equiv q} \qquad \frac{p \equiv q \quad q \equiv r}{p \equiv r}$$

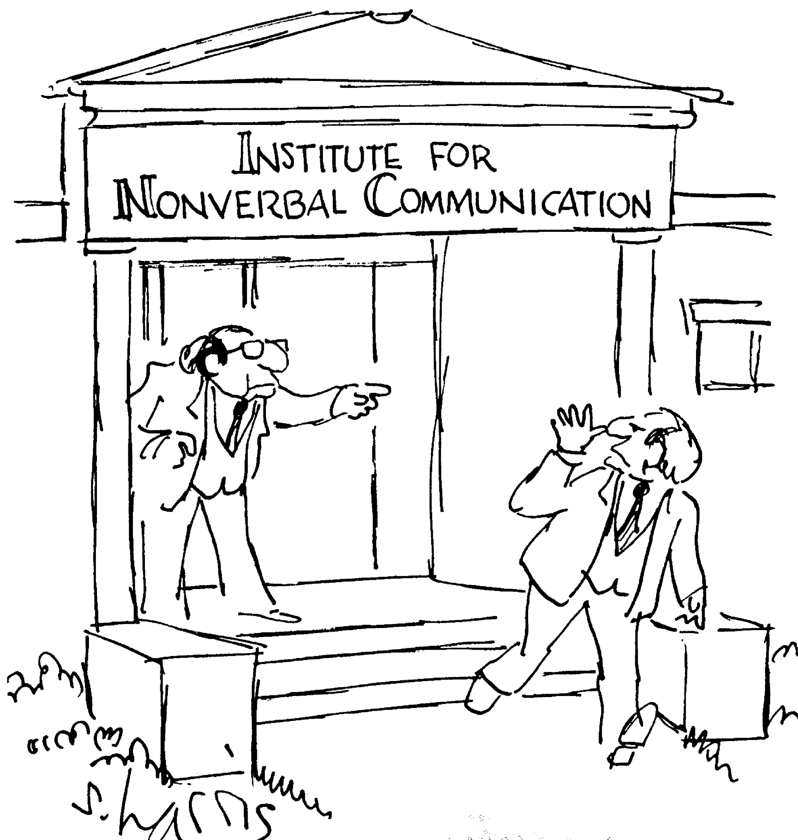
$$\frac{p \equiv p'}{p \mid q \equiv p' \mid q} \qquad \frac{p \equiv p'}{(\nu x)p \equiv (\nu x)p'}$$

$$\begin{array}{rcll}
!p & \equiv & p \mid !p & \\
p \mid \texttt{nil} & \equiv & p & \\
p \mid q & \equiv & q \mid p & \\
(\nu x)(\nu y)p & \equiv & (\nu y)(\nu x)p & \\
(\nu x)\texttt{nil} & \equiv & \texttt{nil} & \\
(\nu x)(p \mid q) & \equiv & (\nu x)p \mid q & \quad x \text{ not free in } q
\end{array}$$

Semantics and Evaluation

- IMP opsem has the "diamond property"
- Does the Pi Calculus? Why or why not?





Theory of Pi Calculus

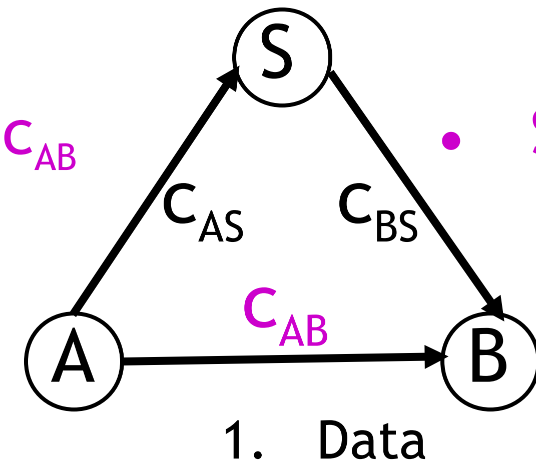
- The Pi calculus does ***not*** have the Church-Rosser property
  - Recall: WireTap | wire<M> | ATT $\to^*$ NSA<M> | fiber<M>
  - Also: WireTap | wire<M> | ATT $\to^*$ WireTap | fiber<M>
  - This captures the *non-deterministic nature* of concurrency
- For Pi-calculus there are
  - Type systems
  - Equivalences and logics
  - Expressiveness results, through encodings of numbers, lists, procedures, objects

Pi Calculus Applications

- A number of languages are based on Pi
  - e.g., Pict (Pierce and Turner)
- Specification and verification
  - mobile phone protocols, security protocols
- Pi channels have nice built-in properties, such as:
  - integrity
  - confidentiality (with ν)
  - exactly-once semantics
  - mobility (channels as first-class values)
- These properties are useful in high-level descriptions of security protocols
- More detailed descriptions are possible in the spi calculus (= pi calculus + cryptography)

A Typical Security Protocol

- Establishment and use of a secret channel:



- New channel c_{AB}
- Same new channel c_{AB}

- A and B are two clients
- S is an authentication server
- c_{AS} and c_{BS} are existing private channels with server
- c_{AB} is a new channel for the clients

That Security Protocol in Pi

- That protocol is described as follows:

  $A(M) = (\nu\, c_{AB})\; c_{AS}\langle c_{AB}\rangle .\; c_{AB}\langle M\rangle$

  $S = !\;(c_{AS}(x).\; c_{BS}\langle x\rangle \mid c_{BS}(x).\; c_{AS}\langle x\rangle)$

  $B = c_{BS}(x).\; x(y).\; \mathrm{Work}(y)$

  $\mathrm{System}(M) = (\nu\, c_{AS})(\nu\, c_{BS})\; A(M) \mid S \mid B$

  - Where Work(y) represents what B does with the message M (bound to y) that it receives
  - The $\mid c_{BS}(x).\; c_{AS}\langle x\rangle$ makes the server symmetric

Some Security Properties

- An <u>authenticity</u> property
  - For all N, if B receives N then A sent N to B
- A <u>secrecy</u> property
  - An outsider cannot tell System(M) apart from System(N), unless B reveals some part of A's message
- Both of these properties can be formalized and proved in the Pi calculus
- The secrecy property can be treated via a simple type system

Mainstream Languages

- Communication channels are not found in popular languages
  - sockets in C are reminiscent of channels
  - STREAMS (never used) are even closer
  - ML has exactly what we've described (surprise)

- More popular is *remote procedure call* or (for OO languages) *remote method invocation*

Concurrent ML

- Concurrent ML (CML) extends of ML with:
  - threads
  - typed channels
  - pre-emptive scheduling
  - garbage collection for threads and channels
  - synchronous communication
  - events as first-class values
- OCaml has it (Event, Thread), etc.
  - "**First-class synchronous communication**. This module implements synchronous inter-thread communications over channels. *As in John Reppy's Concurrent ML system*, the communication events are first-class values: they can be built and combined independently before being offered for communication."

Threads and Channels in CML

val spawn : (unit → unit) → thread *(* create a new thread *)*
val channel : unit → ‘a chan *(* create a new typed channel *)*
val accept : ‘a chan → ‘a *(* message passing operations *)*
val send : (‘a chan * ‘a) → unit

So one can write, for example:

```
fun serverLoop () =  let request = accept recCh in
                     send (replyCh, workOn request);
                     serverLoop ()
```

Basic Events in Concurrent ML

val sync : ‘a event → ‘a *(* force synchronization on an event, block until this communication succeeds *)*

val transmit : (‘a chan * ‘a) → unit event *(* nonblocking; promises to do the send at some point *)*

val receive : ‘a chan → ‘a event *(* sets up the rendezvous, but you don’t actually get the value until you sync *)*

val choose : ‘a event list → ‘a event *(* succeeds when one of the events in the list succeeds *)*

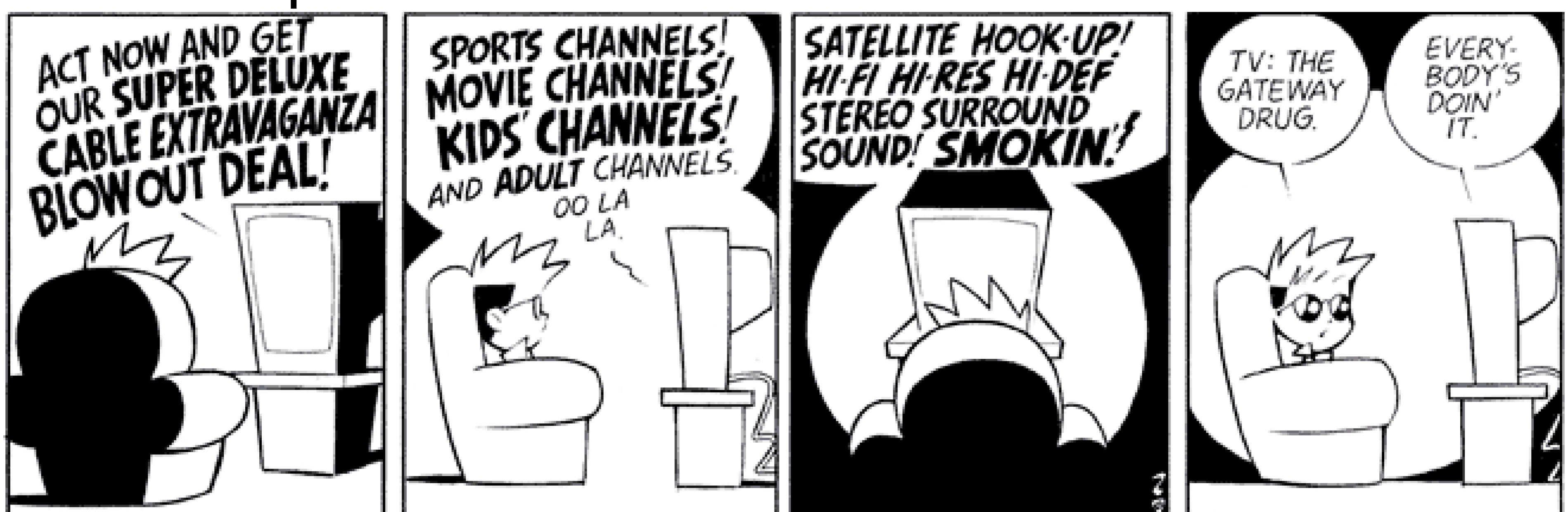
val wrap : (‘a event * (‘a → ‘b)) → ‘b event *(* do an action after synchronization on an event *)*

So you can write, as in Unix syscall select(2):

select (mylist : ‘a event list) : ‘a = sync (choose mylist)

Java Remote Method Invocation

- Java RMI is a Java extension with
  - Java method invocation syntax
  - similar semantics
  - static checks
  - distributed garbage collection
  - exceptions for failures



RMI notes

- Compare RMI with pure message passing
  - RMI is weaker, but OK for many purposes
- RMI not a perfect fit into Java:
  - non-remote objects are passed by copy in RMI
  - clients use remote interfaces, not remote classes
  - clients must handle RemoteException
  - using same syntax for MI and RMI leads to hidden performance costs
- But it is not an unreasonable design!

Homework

- Project
  - Need help? Stop by my office or send email.