Pi Calculus: Formal Models of Concurrent Systems

Jodi Spacek
Joey Eremondi

CPSC 509

November 28, 2016
History: Process Calculus

- family of languages that model processes
- modelling gives us tools to verify software
- used to model concurrent software, especially for non-deterministic behaviours
- syntax is a set of rules that define behaviours of software so we can reason about them
- includes CCS, CSP, Pi Calculus, others
- approaches define operational, denotational, or algebraic semantics

[4]
History: Communicating Sequential Processes

- developed alongside the precursor to Pi Calculus
- alternative language for modelling concurrent programs
devolved by Tony Hoare 1977
- first a programming language, then a theory built around it
- processes interact through message passing
- still being actively researched, and programmed in!
Go programming language channels use concepts from CSP, passing messages between them.

Figure: Channels in Go
History: Calculus of Communicating Systems

- Precursor to Pi Calculus, Robin Milner 1980
- **synchronisation trees**: interleaving model of parallel computation
- each node in the tree is a sequence of actions, subtrees are future behaviours
- **observation equivalence**: equivalence relation on trees
- **terms**: represent equivalence classes of trees
- **qualitative properties**: evaluates deadlock

[2]
Pi Calculus: Motivation

- Lambda Calculus: Good at modelling procedures
- How do we model more than one process?
- How do we prove things about concurrent systems?
The Language
The Pi Calculus

- Pi Calculus [1, 7, 8]
- *Channels* of communication as a fundamental abstraction
- We send along channels, but the values we send are also channels
- Imagine sockets, but all you can send are IP addresses!
- Can model multiple processes in a single term
Syntax

\[ T \in \text{Term} \]
\[ x, y \in \text{Var} \]
\[ \mathcal{P}, \text{set of all processes} \]
\[ P \in \mathcal{P} \]

\[ P ::= P_1 \mid P_2 \quad \text{(Parallel composition)} \]
\[ (\nu x)P \quad \text{(Fresh channel)} \]
\[ \overline{x}(y).P \quad \text{(Send y over channel x)} \]
\[ x(y).P \quad \text{(Receive y over channel x)} \]
\[ !P \quad \text{(Replicate P infinitely)} \]
\[ 0 \quad \text{(Do nothing)} \]
Parallel Composition

- $P \mid Q$
- Processes $P$ and $Q$ run in parallel
- Can step either process, just like if they were concurrent
- If $P \rightarrow P'$, then $(P \mid Q) \rightarrow (P' \mid Q)$
- If $Q \rightarrow Q'$, then $(P \mid Q) \rightarrow (P \mid Q')$
Fresh channels

- \((\nu x)P\)
- Create a new channel, then do \(P\)
- \(P\) can now refer to \(x\)
- \(x\) bound in this expression
- Shadows any previously defined \(x\) variables
Sending

- Send: $\overline{x}(y).P$
- Send channel $y$ over channel $x$, then do $P$
- Blocking: Cannot progress until the message is received
• Receive: $x(y).P$
• Receive a channel over $x$, then do $P$ with the received value in place of $y$
• Blocking: Cannot progress until a message is sent
• Binding: $y$ bound in this expression, but $x$ free
Replication

- !P
- Create an infinite number of processes, each running \( P \)
- \!P \) the same as \( P \mid \!P \)
Do Nothing

- 0
- The empty processes, does nothing
- Used as an end-marker for termination
Message Semantics

- \((\overline{x(y)}.P \mid x(z).Q) \rightarrow (P \mid [y/z]Q)\)
- Can proceed once there’s a send and receive waiting on the same channel in parallel
- Value sent gets substituted into \(Q\)
Congruence Rules

- Want to treat certain expressions as “equivalent”
- Kind of like context-frames for small-step, but non-deterministic
- For example, don’t care about order of composition
- Can apply congruence rules before/after stepping
Congruence Rules

\[
P \mid Q \equiv Q \mid P \quad \text{(Commutativity)}
\]

\[
P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad \text{(Associativity)}
\]

\[
((\nu x)P) \mid Q \equiv (\nu x)(P \mid Q) \text{ if } x \not\in FV(Q) \quad \text{(Scope extrusion)}
\]

\[
!P \equiv P \mid !P \quad \text{(Replication unfolding)}
\]
Reduction Rules

(communication) \[ (\overline{x}(y).P \mid x(z).Q) \rightarrow (P \mid [y/z]Q) \]

(parallel-composition) \[ P \rightarrow Q \]
\[ P \mid R \rightarrow Q \mid R \]

(restriction) \[ P \rightarrow Q \]
\[ (\nu x)P \mid (\nu x)Q \]

(congruence) \[ P \equiv P' \quad Q \equiv Q' \quad P' \rightarrow Q' \]
\[ P \rightarrow Q \]
Examples!
Use the communication reduction multiple times to send messages between channels:

\[ \overline{x} \langle y \rangle. \overline{y} \langle z \rangle . 0 \mid x \langle w \rangle . w(v) \rightarrow \]
Example of Communication 2

Use the communication reduction multiple times to send messages between channels:

\[ x\langle y \rangle . y\langle z \rangle . 0 \mid x(w).w(v) \rightarrow \]
\[ y\langle z \rangle . 0 \mid [y/w]w(v) = \]
\[ y\langle z \rangle . 0 \mid y(z) \]
Example of Communication 3

Done!

\[
\bar{x}\langle y \rangle \cdot \bar{y}\langle z \rangle \cdot 0 \mid x(w) \cdot w(v) \rightarrow \\
\bar{y}\langle z \rangle \cdot 0 \mid [y/w]w(v) = \\
\bar{y}\langle z \rangle \cdot 0 \mid y(z) \rightarrow \\
0 \mid 0
\]
Example of Congruence (1)

Step the first expression with the second?

\[ x(y).P \mid (\overline{x} y).Q \mid \overline{x} z).R \]

We can't reduce because the ordering is wrong!

\[ \overline{a} x).P \mid a(y).Q \rightarrow P \mid Q[x/y] \]

Our communication rule says that Send needs to appear before Receive. Let's try another approach...
Example of Congruence (2)

Match the first to third expression?

\[ x(y).P \mid \overline{x}(y).Q \mid \overline{x}(z).R \]

This has the same problem, This also doesn’t match any reductions.
Example of Congruence (3)

If we apply our congruence rules, we can rearrange the brackets to do the send from the first to the second, or we can rearrange the compositions to do send from the first to the third.

\[
x(y).P \mid (\overline{x}\langle y\rangle.Q \mid \overline{x}\langle z\rangle.R)
\equiv (\overline{x}\langle y\rangle.Q \mid x(y).P \mid \overline{x}\langle z\rangle.R)
\rightarrow (Q \mid P) \mid \overline{x}\langle z\rangle.R
\]
Example of Restriction (1)

A restriction adds scoping, such that the name is bound within the process it’s attached to:

\[(\nu x)(x(z).0)\]

\[| \ x(y).\overline{y}(x).x(y).0)\]

\[| \ z(v).\overline{v}(v).0\]

We can evaluate the first two expressions, substituting \(z\) for \(y\)

\[= (\nu x)(0)\]

\[| \ [z/y]\overline{y}(x).x(y).0)\]

\[| \ z(v).\overline{v}(v).0\]
Example of Restriction (2)

Note that the substitution $[z/y]$ only applies to outer $\overline{y} \langle x \rangle$. and not to inner scoped $x(y)$.

\[
\rightarrow (\nu x)(0 \mid \overline{z} \langle x \rangle . x(y).0) \\
\quad \mid z(v) . \overline{v} \langle v \rangle .0
\]

Now we can step on channel $z$ and first perform the substitution:

\[
= (\nu x)(0 \mid [x/v]x(y).0) \\
\quad \mid [x/v]\overline{v} \langle v \rangle .0
\]
Example of Restriction (3)

Note there is no restriction on $\nu$, so we are able to rename the inner scoping of $\overline{\nu}\langle\nu\rangle$. We have also extended the restriction to the third expression because we renamed the Send to channel $x$.

$$\rightarrow (\nu x)(0 \mid x(y).0 \mid \overline{x}\langle x\rangle.0)$$

Matching the Send and Receives on channel $x$, we step to:

$$\rightarrow (\nu x)(0 \mid 0 \mid 0)$$

And we are done!
Properties of the Pi Calculus
Nondeterminism

- For some term $P$, might be multiple reductions we can apply
- That is, there exist $Q, Q'$ such that $P \rightarrow Q \land P \rightarrow Q'$.
- Lambda calculus traditionally also non-deterministic, many $\beta$-reductions
- Resolving this non-determinism with $\lambda$: lazy vs strict
Confluence: If $A \rightarrow^* B$ and $A \rightarrow^* C$, then there’s some $D$ where $B \rightarrow^* D$ and $C \rightarrow^* D$

Lambda calculus is confluent: if you terminate, you get the same results regardless of the order of reductions

Pi calculus is not confluent!
Example of non-confluence (1)

\[ \overline{x}\langle y\rangle.0 \mid x(z)\overline{z}\langle z\rangle.0 \mid x(w).0 \]

Let’s step the first two expressions; a Send and a Receive on channel x:

\[ = 0 \mid \frac{y}{z}\overline{z}\langle z\rangle.0 \mid x(w).0 \]

This substitution evaluates to

\[ \rightarrow 0 \mid \overline{y}\langle y\rangle.0 \mid x(w).0 \]

Stuck: the Send and the Receive are on different channels! Is there another way?
Example of non-confluence (2)

OK, this time let’s step the first and third expressions on $x$:

$\overline{x}(y).0 \mid x(z).\overline{z}(z).0 \mid x(w).0$

Perform the renaming:

$= 0 \mid x(z).z0 \mid [y/w]0$

We’re left with a totally different result:

$\rightarrow 0 \mid x(z).\overline{z}(z).0 \mid 0$

Which process should take the message from channel $x$?

What kind of condition is this known as in concurrent computing?

Why is it important that we’re able to express this?
Consequences of Non-confluence

- There might be many final “values” for a process
- Allows us to model race conditions
- This is deliberate: can’t prove that programs are deterministic without first modelling non-determinism
Channels Are Enough: Turing Completeness
Sending Tuples

- Can model multiple value sending with single sending [8]
- $\langle y_1, \ldots, y_n \rangle. P = (\nu p)\overline{x}\langle p \rangle. \overline{p}\langle y_1 \rangle \ldots \overline{p}\langle y_n \rangle. P$
- Create a fresh channel, send the channel, then send all the values over that channel
Boolean Values

- For some channel \( b \) [8]:
  - \( TRUE(b) = !b(t,f).t\langle\rangle.0 \)
  - \( FALSE(b) = !b(t,f).f\langle\rangle.0 \)
- Booleans receive two channels (branches)
- True notifies the first branch, False notifies the second
Branching

- If $b$ then $P$ else $Q$:
- $TEST(b) = (\nu t)(\nu f)\overline{b}(t, f).(t().P \mid f().Q)$
- Send message to boolean channel, will receive response on one of true/false branches
- If have $TRUE(b) \mid TEST(b)$, will run $P$
- Similarly, will run $Q$ in False case
Branching Example

- If $FALSE(b)$ then $\overline{x}\langle y \rangle.0$ else $\overline{x}\langle z \rangle.0$
- Substitute in $TEST(b)$, with $FALSE(b)$ running in parallel

$$
!b(t', f').\overline{f'}\langle \rangle.0 \mid (\nu t)(\nu f)\overline{b}\langle t, f \rangle.(t().\overline{x}\langle y \rangle.0 \mid f().\overline{x}\langle z \rangle.0)
$$

$$
\equiv b(t', f').\overline{f'}\langle \rangle.0 \mid !b(t', f').\overline{f'}\langle \rangle.0 \mid (\nu t)(\nu f)\overline{b}\langle t, f \rangle.(t().\overline{x}\langle y \rangle.0 \mid f().\overline{x}\langle z \rangle.0)
$$

$$
\rightarrow \overline{f}\langle \rangle.0 \mid !b(t', f').\overline{f'}\langle \rangle.0 \mid (\nu t)(\nu f)(t().\overline{x}\langle y \rangle.0 \mid f().\overline{x}\langle z \rangle.0)
$$

$$
\rightarrow !b(t', f').\overline{f'}\langle \rangle.0 \mid (\nu t)(\nu f)(t().\overline{x}\langle y \rangle.0 \mid \overline{x}\langle z \rangle.0)
$$

- If had someone listening on $x$, would do $\overline{x}\langle z \rangle.0$
- Will never do $\overline{x}\langle y \rangle.0$ since $t$ never receives a message
- $t$ will never receive a message, since $\nu$ limits its scope
Simulating the Lambda Calculus

- Given Lambda expression $M$, and a channel $p$
- Translation function $\Phi : \Lambda \times \text{VAR} \rightarrow \mathcal{P}$ (from [8])
  - $\Phi(\lambda x. M, p) = p(x, q) \cdot \Phi(M, q)$
  - $\Phi(x, p) = \overline{x}\langle p \rangle . 0$
  - $\Phi(M \ N, p) = (\nu q)(\Phi(M, q) \mid (\nu y)q\langle y, p \rangle . 0 \mid !y(r) . \Phi(N, r)))$
- This is enough to show Turing completeness
- There are translations for strict and lazy calculi [7]
Functions

- \( \Phi(\lambda x. M, p) = p(x, q).\Phi(M, q) \)
- Every function has an argument port \( p \)
- Listen on \( p \) for an argument and a new argument port
- Do the function body when receive (i.e. when called)
Variables

- $\Phi(x, p) = \overline{x} \langle p \rangle.0$
- Sends the argument port along the variable’s channel
Applications

- \( \Phi(M, N, p) = (\nu q)(\Phi(M, q) \parallel ((\nu y)\langle y, p \rangle.0 \parallel !y(r).\Phi(N, r) )) \)
- Creates a fresh channel \( q \) for the function to listen on
- Make the process for the function with \( q \) as its port
- In parallel, send \( q \) a fresh channel \( y \) to send on for its argument, and a port \( p \) to send along the argument channel
- In parallel, makes a process that infinitely listens on \( y \)
Example Translated Program

- \((\lambda x.x \ x) \ (\lambda w.w)\)
- Evaluates to \((\lambda w.w)\)

\[
\Phi((\lambda x.x \ x) \ (\lambda w.w), p) \\
= (\nu q)(\Phi(\lambda x.x \ x, q) \ | \ ((\nu y)\overline{q}\langle y, p\rangle.0 \ | \ !y(r).\Phi(\lambda w.w, r)) \\
= (\nu q)(\Phi(\lambda x.x \ x, q) \ | \ ((\nu y)\overline{q}\langle y, p\rangle.0 \ | \ !y(r).\Phi(\lambda w.w, r)) \\
= (\nu q)(q(x, s).\Phi(x \ x, s)) \ | \ ((\nu y)\overline{q}\langle y, p\rangle.0 \ | \ !y(r).\Phi(\lambda w.w, r)) \\
\to (\nu q)[y/x]\Phi(x \ x, p) \ | \ !y(r).\Phi(\lambda w.w, r)
\]

Translation: First Process

\[
[y/x] \Phi(x, x, p) \\
= [y/x] (\Phi(x, q) | (\nu y') \overline{q'} \langle y', p \rangle . 0 | !y'(r'). \Phi(x, r')) \\
= [y/x] (x \langle q' \rangle . 0 | (\nu y') \overline{q'} \langle y', p \rangle . 0 | !y'(r'). x \langle r' \rangle . 0) \\
= (\bar{y} \langle q' \rangle . 0 | (\nu y') \overline{q'} \langle y', p \rangle . 0 | !y'(r'). \bar{y} \langle r' \rangle . 0)
\]
Translation: Second Process

\[ !y(r).\Phi(\lambda w.w, r) = !y(r).r(w, u).\Phi(w, u) = !y(r).r(w, u).\overline{\langle u \rangle}.0 \]
\[ \equiv y(r).r(w, u).\overline{\langle u \rangle}.0 \mid !y(r).r(w, u).\overline{\langle u \rangle}.0 \]
Translation: Bringing them together

- First:  \((\overline{y}q').0 | ((\nu y')q'y',p).0 | !y'(r').\overline{y}(r').0)\)
- Second:  \(y(r).r(w,u).\overline{w}u).0 | !y(r).r(w,u).\overline{w}u).0\)
- Do the receive on \(y\) gives:

\[
((\nu y')q'\langle y',p\rangle.0 | !y'(r').\overline{y}(r').0)) \Rightarrow \overline{y}'(p).0 \Rightarrow !y(r).r(w,u).\overline{w}u).0
\]

\[
\equiv((\nu y')y'(r').\overline{y}(r').0 | !y'(r').\overline{y}(r').0)) \Rightarrow \overline{y}'(p).0 \Rightarrow !y(r).r(w,u).\overline{w}u).0
\]

\[
\rightarrow((\nu y')\overline{y}(p).0 | !y'(r').\overline{y}(r').0)) \Rightarrow !y(r).r(w,u).\overline{w}u).0
\]
Translation: Unrolling the bang

\[(\nu y')\overline{y}(p).0 \mid !y'(r').\overline{y}(r').0) \mid !y(r).r(w,u).\overline{w}(u).0\]

\[\equiv (\nu y')\overline{y}(p).0 \mid !y'(r').\overline{y}(r').0) \mid y(r).r(w,u).\overline{w}(u).0 \mid !y(r).r(w,u).\overline{w}(u).0\]

\[\rightarrow (\nu y')!y'(r').\overline{y}(r').0) \mid p(w,u).\overline{w}(u).0 \mid !y(r).r(w,u).\overline{w}(u).0\]

- What are we left with?
- Two deadlocked ! processes, which do nothing
- And \(\Phi(\lambda w. w, u)\), our result with port channel \(p\)
Extensions of the Pi Calculus
Language Features

- Channels are enough, but they’re not convenient
- Can extend with traditional language features, branching, iteration, etc.
- Looks more like a normal imperative language

[3]
Session Types: Motivation

- Would like to filter out programs that deadlock or are non-deterministic
- Impose a *behavioral type system*
- Describes a protocol for a program
- Shape of communications, rather than data

[3]
An example type system

\[ T ::= \]
| \( ?x. T \) (Receive on \( x \))
| \( !x. T \) (Send on \( x \))
| \( \alpha \) (Recursion variable)
| \( \mu \alpha. T \) (Recursive type)

- If we’ve augmented our language, abstracts away all but communication
- Can check session types for linearity (make sure sends and receives line up)
- Rules what you’d expect (sends match sends, etc.)
- Only models 2 processes (2-party)
- Many extensions and variants exist
Multiparty Asynchronous Session Types

\[ P ::= s : h \text{ Message queue} \]
\[ m ::= (q, p, v) \mid (q, p, s[p']) \mid (q, p, l) \text{ Message in transit} \]
\[ h ::= h : m \text{ Queue} \]

- aims to model asynchronous behaviour in Distributed Systems by allowing multiple channels to communicate
- can model conversation communications, leader/follower, quorums
- extends binary session types, additional rules to rearrange senders and receivers
- allows messages to be matched in a queue, removing restriction of matching channel names
- messages in a queue can be values, channels (for delegation), or a label
- message \( m \) is concatenated onto queue \( h \)

[3] [5]
Multiparty Asynchronous Session Types - In Action

Figure: Fencing of parent-child relationships in MiGo

- process calculus called *Migo*, a tool in Go to fence off channels
- restrict channels to a finite set of *fenced* types
- this more closely models realistic topology in a system of nodes
- statically ensure liveness and safety guarantees
- eg. sending a message on a closed channel can be verified statically

[6]
Not this kind of pie

Figure: No Pie
References
References


