JOIN CALCULUS

Roberto Bruni
Dipartimento di Informatica
Università di Pisa

Contents

- Introduction
- Join calculus + Examples
- Join and π

Join calculus vs. π calculus

- Join is essentially π with restrictions on communication patterns
  - Join combines restriction, reception and replication in a single receptor definition
  - not available separately
  - asynchronous calculus, continuation passing style
  - asynchrony forces one to create and send continuations
- Nevertheless, join and (asynchronous) π have the same expressive power
  - demonstrated by fully abstract encodings in each direction
  - (up to weak barbed congruence)

Motivation (back to 1995)

- Join calculus has been devised to bridge the gap between mathematical abstractions and distributed programming languages
  - process calculus presentation
  - basis for a practical programming language design

Join Calculus vs Petri Nets

- Join-calculus can be seen also as the natural higher order extension of Petri nets
  - places as ports / channels
  - tokens carry values
  - names of places are also admissible values
  - firing can generate fresh pieces of nets
    - new places
    - new transitions
Chemical Abstract Machine

- States are called solutions $s$
  - Multisets of molecules $m_1, \ldots, m_n$
  - Data and rules (reflexive CHAM)
- Evolution (chemical rules)
  - Heating / cooling $\Rightarrow$ (reversible)
  - Structural equivalence

Example: Cell Abstraction

\[ \text{get}(k) | \text{s}(v) \Rightarrow \text{k}(v) | \text{s}(v) \]

- A cell $s$ contains the value $v$
- To get the value:
  - Send a message on port get
  - The parameter $k$ is the return address, where the value $v$ will be sent to
Example: Cell Abstraction

- A cell $s$ contains the value $v$
- To set the value:
  - send a message on port set
  - the parameter $m$ is the new value for $s$
  - $k$ is the return address (for confirmation)

Example: Cell Abstraction

- The initial value in $s$ is $n$
- But get, set and $s$ are locally bound by `def`
  - get and set must be extruded, otherwise no one can use them
  - instead, $s$ is kept private (encapsulation)

Example: Cell Abstraction

- `def get(k) | s(v) :: k(v) | s(v)`
- `set(m,k) | s(v) :: k(m) | s(m)`
- `in s(n) | <get,set>`

- get, set are extruded on public channel $c$
- But $c$ should be known only by the owner of the cell...

Example: Cell Abstraction

- `def create(n,c) ::
  - `def get(k) | s(v) :: k(v) | s(v)`
  - `set(m,k) | s(v) :: k(m) | s(m)`
  - `in s(n) | <get,set>`

- A message to create triggers the outermost `def`:
  - Three fresh names for $s$, get and set are allocated
    - the initial value of $s$ is the first parameter $n$
    - get and set are sent back to the second argument $c$
    - instead $s$ will never be extruded
  - Invariant
    - in every configuration there is exactly one message on $s$

Join Calculus in One Slide

- Syntax
  - `P | Q` :: $0 | x(y) | def D in P | P|Q`
  - `P.D` :: $J\cdot P | D\cdot E$
  - `J\cdot x` :: x(y) | J\cdot K`

- Operational semantics (CHAM Style)
  - `P|Q` :: $P.Q$
  - `D.E` :: $D.E$
  - `D (range $e_{wire}$, event $e_{wire}$)`
    - heating and cooling
  - `def D in P.J\cdot P` :: $J\cdot P, J\cdot E$

JOIN: An Example

- A process $P$
  - $P = z(x,z) :: def x(y) :: z(y,x) in x(y)$

- $P$ as a solution
  - $z(x,z), w(y) :: z(y,w), w(y)$

- A reaction
  - $z(x,z), w(y) :: z(y,w), w(y)$
    - $z(x,z), w(x) :: z(y,w), z(v,w)$
**Homework**

- Guess the meaning of:
  - `def x(u) ▷ y(u) in P`
  - `def y(u) ▷ x(u) in def x(u) ▷ y(u) in P`
  - `def s() ▷ P ∧ s() ▷ Q in s()`
  - `def (c() ▷ P)c() in Q[c()]`

---

**Example: Mailbox (sketched)**

```latex
rb@gmail.it/from,subj,msg) | save() ▷ store/from,subj,msg) | save()
ch/fwd/email) | save() ▷ fwd/email
rb@gmail.it/from,subj,msg) | fwd/email)
ch/vacation/info) | save() ▷ vacation/info
rb@gmail.it/from,subj,msg) | vacation/info)
    inbox(get,next) ▷ store/from,subj,msg)
    def elget(k) ▷ elem(f,s,m,g,n) ▷ k(f,s,m) ▷ elem(f,s,m,t,n)
    ∧ elnext(k) ▷ elem(f,s,m,g,n) ▷ k(g,n)
    in inbox(elget,elnext) ▷ elem(from,subj,msg),get,next)
```

---

**Continuation Passing Style I**

- The form of definitions resembles very much
  - `let f(x)=E in E'` (typical of functional programming)
  - e.g. same scoping discipline
  - Asynchrony forces us to create and send continuations in join
  - e.g. encoding untyped λ-calculus
  - M sends the value of N on ν
  - a value is a process serving requests
  - a request must supply two names
    - x (channel for requests for the value of the argument)
    - w (to eventually return a value)

---

**Continuation Passing Style II**

- **Call-by-name**
  - `⟦x⟧_v = v(x)`
  - `⟦λx.M⟧_v = def k(x,w) ▷ [M]_w in ν(k)`
  - `⟦MN⟧_v = def M(y) ▷ ν_k [N]_k in ν(k)`
  - `⟦q(c) ▷ c(y,v) in [M]_q in [N]_p`
  - **Parallel call-by-value**
    - `⟦x⟧_v = ν(x)`
    - `⟦λx.M⟧_v = def k(x,w) ▷ [M]_w in ν(k)`
    - `⟦MN⟧_v = def q(c) ▷ c(y,v) in [M]_q [N]_p`

---

**Call-by-Name**

- Strategy: leftmost order, no reduction under λ
- Reductions are entirely sequential
- The image of the translation is exactly the deterministic subset of Join (no parallel composition, no conjunction)
  - `⟦x⟧_v = v(x)`
  - `⟦λx.M⟧_v = def k(x,w) ▷ [M]_w in ν(k)`
  - `⟦MN⟧_v = def M(y) ▷ [N]_y in ν(k)`
  - `⟦q(c) ▷ c(y,v) in [M]_q`
Call-by-Name: Example

- \[(\lambda x. M) N_1 = \text{def } y(p) : [N]_p\]
  \(\text{in def } q(c) \triangleright c(y,v) \text{ in } [\lambda x. M]_q\)
- \[(\lambda x. M) N_1 = \text{def } y(p) : [N]_p\]
  \(\text{in def } q(c) \triangleright c(y,v) \text{ in } [\lambda x. M]_q\)
- \[(\lambda x. M) N_1 \rightarrow \text{def } y(p) : [N]_p\]
  \(\text{in def } q(c) \triangleright c(y,v) \text{ in } [\lambda x. M]_q\)
- \[(\lambda x. M) N_1 \rightarrow \text{def } k(x,w) \triangleright [M]_w \text{ in } k(y,v)\]
- \[(\lambda x. M) N_1 \rightarrow \text{def } q(c) \triangleright c(y,v) \text{ in } [\lambda x. M]_q\]
  \(\text{in def } k(x,w) \triangleright [M]_w \text{ in } k(y,v)\)

Parallel Call-by-Value

- Strategy: again no reduction under \(\lambda v\), but in \((TU), T \land U\) can be evaluated in parallel
- Confluent, but non deterministic
- \([x]_v = v(x)\)
- \((\lambda x. M) = \text{def } k(x,w) \triangleright [M]_w \text{ in } v(k)\)
- \([MN]_v = \text{def } q(c)p(y) \triangleright c(y,v) \text{ in } [M]_q[N]_p\)

Call-by-Value: Example

- \[(\lambda x. M) N_1 = \text{def } q(c)p(y) \triangleright c(y,v) \text{ in } [\lambda x. M]_q \parallel [N]_p\]
- \[(\lambda x. M) N_1 = \text{def } q(c)p(y) \triangleright c(y,v) \text{ in } [N]_p \parallel \text{def } k(x,w) \triangleright [M]_w \text{ in } q(k)\]
- \[(\lambda x. M) N_1 \rightarrow \text{def } q(c)p(y) \triangleright c(y,v) \text{ in } p(z) \parallel \text{def } k(x,w) \triangleright [M]_w \text{ in } q(k)\]
- \[(\lambda x. M) N_1 \rightarrow \text{def } q(c)p(y) \triangleright c(y,v) \text{ in } k(z,v) \parallel \text{def } k(x,w) \triangleright [M]_w \text{ in } [M(z)]_k\]
- \[(\lambda x. M) N_1 \rightarrow \text{def } q(c)p(y) \triangleright c(y,v) \text{ in } \text{def } k(x,w) \triangleright [M]_w \text{ in } [M(z)]_k\]

Contents

- Introduction
- Join calculus + Examples
- Join and \(\pi\)

Core Join Calculus

- Syntax
  - a unique syntactic category
    \(P, Q ::= x(u) \mid \text{def } x(u)(v) \triangleright Q \mid P \parallel P; Q\)
- Operational semantics
  - CHAM Style
  - (but also LTS is defined)
  - The core join calculus has the same expressive power as the full join-calculus
    via a fully-abstract encoding.
Full abstraction

- Two process calculi with equivalences ≈₁ and ≈₂
- The first is more expressive than the second if we can find a fully abstract encoding [.]₂→₁
  - i.e. an encoding such that
  - P ≈₂ Q iff [P]₂→₁ ≈₁ [Q]₂→₁
- The two calculi have the same expressive power if each one is more expressive than the other
  - (If one is a sub-calculus of the other, then one implication is obvious)

What is Observable?

- Communication
  - on internal names (no)
  - on free names (yes)
- Internal steps
  - countable: strong semantics (no)
  - immaterial: weak semantics (yes)
- Equivalence
  - reflexive, symmetric and transitive (yes)
  - closed under contexts: congruence (yes)

Basic Observations

- Processes interact with the outside
  - by extruding names on free ports
  - by waiting for answers (via enclosed definitions)
- Processes are distinguished on the basis of their ability to emit messages on their free ports
  - weak asynchronous output barb ĳₗₙₖ
  - Pįₗₙₖ iff
    - x is a free name in P
    - and Ǝu such that P →ₓ Qₓ(u)

Remarks on Barbs

- Two processes P and Q such that
  - Ǝu with Pįₗₙₖ but ¬(Qįₗₙₖ)
  - cannot be reasonably identified!
- Barbs are just elementary experiments
  - barbs do not count reductions (ok)
  - barbs do not observe branching (uhm)
  - barbs do not observe message reception (uhm)

Closure Under Reductions

- Reductions are mute transitions
  - i.e. only trivial labels are present
    - P →P′ can be read as P →ε →P′
- In ordinary (strong) bisimulation
  - if P≡Q and P→P′, then ƎQ′≡P′ s.t. Q→Q′
    - (and vice versa)
  - In weak bisimulation
    - if P≡Q and P →* P′, then ƎQ′≡P′ s.t. Q→* Q′
      - (and vice versa)

Closure Under Contexts

- If P≡Q we expect that P and Q can be used interchangeably in any larger process
  - but P = a(b) and Q = a(c) look equivalent when taken in isolation
    - no reduction, a unique barb ĳₗₙₖ
  - however, they are not equivalent in the context
    - def a(x) : xₒ in [₁]
  - as in fact
    - def a(x) : xₒ in P → bₒ (i.e. def a(x) : xₒ in P ĳₗₙₖ)
    - def a(x) : xₒ in Q → cₒ (i.e. def a(x) : xₒ in Q ĳₗₙ₆)
The Observational Congruence

- We take the largest equivalence relation \( \equiv \) that is a refinement of output barbs
  - if \( P = Q \) then \( (\forall x. P[x]_x = Q[x]_x) \)
  - is closed under weak reduction
    - if \( P = Q \) and \( P \rhd^* P' \), then \( \exists Q' = P' \) s.t. \( Q \rhd^* Q' \)
  - is a congruence w.r.t. definitions and parallel
    - if \( P = Q \) then \( (\forall \Delta. \text{def } D \text{ in } P = \text{def } D \text{ in } Q) \)
    - if \( P = Q \) then \( (\forall \Delta. P | R = Q | R) \)

---

Core Join vs Full Join

- Expressiveness-preserving simplification of syntax
  - recursive binding
    - shift binding variables from definition to reception
      - \( \text{def } j = Q \text{ in } P \implies \text{def } j(\text{def } i = Q_0 \text{ in } P_0) \text{ in } P(i \theta, j \theta) \)
    - where \( \theta \) is the vector of variables in \( f q(i \theta, j \theta) \)
  - complex definitions
    - \( n \)-way join patterns and multiple clauses connected by \( \wedge \) as sequences joining two atoms at most
    - polyadic messages
      - name tuples are communicated by using auxiliary private names

---

Asynchronous \( \pi \)

- Syntax
  - \( P, Q ::= x(u) | x(u).P | vu.P | !x(u).P | P|Q \)

- Abstract semantics
  - asynchronous barbed congruence
  - ex. \( x(u).x(u) = 0 \)
  - ex. equator \( EQ(x,y) = !x(u).y(u) | !y(v).x(v) \)
  - \( P(x/y) = Q(x/y) \) implies \( EQ(x,y) P = EQ(x,y) Q \)

---

Naïve Encoding: Join in \( \pi \)

- \( [x(u)]_{x\in} = x(v) \)
- \( [P|Q]_{x\in} = [P]_{x\in} | [Q]_{x\in} \)
- \( [\text{def } x(u)]_{x\in} = \forall x.y.(x(u),y(v).[Q]_{x\in} | [P]_{x\in}) \)

- In the translation we lose
  - the symmetry between \( x \) and \( y \)
  - the atomicity of their joint reduction
  - it does not matter, because \( x \) and \( y \) are restricted

- Not closed under \( \pi \) contexts
  - if \( x \) or \( y \) are extruded, then new receptors could appear

---

Problems with Full Abstraction of Join in \( \pi \): Example

- Let \( P = [\text{def } x() \triangleright 0 \text{ in } a(x) | x()]_{x\in} \)
  - and \( Q = [\text{def } y() \triangleright 0 \text{ in } a(y)]_{y\in} \)
  - the two encoded processes are equivalent
  - \( P \) and \( Q \) are not

- Take the \( \pi \)-context \( va.(a(u).u().b() | [\_]) \)
  - then \( va.(a(u).u().b() | P | b) \)
  - while \( \neg (va.(a(u).u().b() | Q | b) \cup b) \)
Naïve Encoding: \( \pi \) in Join

- Each \( \pi \)-channel \( x \) is simulated by two ports
  - \( x \) for output (where emitters send values)
  - \( x \) for input (the receiver defines a name \( k \) for its continuation and sends it as a reception offer on \( x \))
  - \( [x(v)] \) \( \xrightarrow{\delta} \) \( x_k(\nu(x, v)) \)
  - \( [x(u), x] \) \( \xrightarrow{k} \) \( [x(\nu(x, v))] \)
  - \( [x(u), x] \) \( \xrightarrow{k} \) \( x_k(v) \)
  - \( [x(u), x] \) \( \xrightarrow{k} \) \( [x(v)] \)

- Not closed under Join contexts!!!
  - problems with free names and input barbs

Problems with Full Abstraction of \( \pi \) in Join: Examples

- \( [x(a) \mid x(b) \mid x(u).y(u)] \)
  - cannot reduce because there is no englobing \( v_x \)
  - exhibits a barb on \( x \) that reveals the presence of an input on \( x \)

Features (as distributed programming language)

- Extends a higher-order functional language
  - parallelism in expressions (fork calls)
  - parallelism in function patterns (join patterns)
    - jointly defined function provide the same capabilities as synchronous channels or concurrent objects
    - join patterns are more consistent with lexical scope
    - static binding of function calls to the code
    - as opposed to dynamic binding of messages to receptors

Polyphonic C# (C\( \omega \))

- Methods can be
  - synchronous
    - (caller is blocked)
  - asynchronous
    - (no result, caller can proceed almost immediately)

- Key feature
  - The same body (called a chord) can be associated with a set of asynchronous and (at most one) synchronous methods
  - A method can appear in the header of several chords
  - The body is executed only if all methods in its header have been called
Unbounded Size Buffer

```java
public class Buffer {
    public String get()
    & public async put(String s) {
        return s;
    }
}
```

One-Place Buffer

```java
public class OnePlaceBuffer {
    public OnePlaceBuffer() {
        empty();
    }
    public void put(String s)
    & private async empty() {
        return;
    }
    public String get()
    & private async contains(String s) {
        empty();
        return s;
    }
}
```

Reader-Writer Lock

```java
class ReaderWriter {
    ReaderWriter() {
        Idle();
    }
    void Exclusive()
    & private async Idle() {
        }
    void ReleaseExclusive() {
        Idle();
    }
    void Shared()
    & private async Idle() {
        S(1);
    }
    void Shared()
    & private async S(int n) {
        S(n+1);
    }
    void ReleaseShared()
    & private async S(int n) {
        if (n == 1) Idle(); else S(n-1);
    }
}
```

References

- The reflexive chemical abstract machine and the Join calculus (Proc. POPL’96, ACM, pp. 372-385)
- C. Fournet, G. Gonthier