CS 6112 (Fall 2011)

Foundations of Concurrency
13 October 2011
Scribe: Mark Reitblatt

Cornell University
Department of Computer Science

1 Chemical Abstraction Machine

The Chemical Abstract Machine (CHAM) is a model of computation for concurrent calculi.

"Intuitively, the state of a system is like a chemical solution in which the floating molecules can interact with each other according to reaction rules; a magical mechanism stirs the solution, allowing for possible contacts between molecules"

Reaction rules are static, specified before computation. The reaction rules specify the signature of concurrent calculus we are modeling. We can think of the rules as *catalysts* of reactions, and also as *places* where molecules must travel to react.

Unfortunately, specifying a fixed set of reaction rules at the beginning of the computation limits how we can model calculi.

- All communication is constrained to the fixed set of reaction sites. "Catalysts are bottlenecks"
- All pattern matching in the core calculus must be in the fixed set of rules, which may complicate dynamic elements of matching. "Catalysts clog up"

1.1 Reflexive CHAM

To solve both of these problems, we permit molecules to add new reactions to their environment. This gives us the Reflexive CHAM.

Reflexive CHAM processes P are either:

- An emission of an asynchronous polyadic message $x\langle\vec{v}\rangle$
- A definition of new names $\mathrm{def}\ D$ in P
- A parallel composition of processes $P \mid P$

A join-pattern J consists of:

- Receipt of an asynchronous polyadic message $x\langle\vec{v}\rangle$
- Parallel composition of joins $J \mid J$

A definition D consists of:

- An elementary definition that match patterns to guarded processes $J \triangleright P$
- A conjunction of definitions $D \wedge D$

Names that appear in a process P may be captured by an enclosing definition. The only binder is the join pattern J, but the scope of its names depeds on their position in messages. The formal parameters that are received are bound in the correspond guarded process. The defined port names are bound in the whole defining process, that is, the main process and recursively all the guarded processes. Received variables $rv(J)$, defined variables $dv(D)$, and free variables $fv(D)$ and $fv(P)$ are specified by structural induction. Notice the syntactic restriction for processes: No received variable may appear twice in the same pattern J. This rules out any comparison on names, and guarantess that join patterns are linear.

$$rv(x\langle\vec{v}\rangle) \triangleq \{u \in \vec{v}\} \qquad rv(J \mid J') \triangleq rv(J) \uplus rv(J')$$

$$dv(x\langle\vec{v}\rangle) \triangleq \{x\} \qquad dv(J \mid J') \triangleq dv(J) \cup dv(J')$$

$$dv(J \rhd P) \triangleq \{dv(J) \qquad dv(D \wedge D') \triangleq dv(D) \cup dv(D')$$

$$fv(J \rhd P) \triangleq dv(J) \cup (fv(P) - rv(J)) \qquad fv(D \wedge D') \triangleq fv(D) \cup fv(D')$$

$$fv(x\langle\vec{v}\rangle) \triangleq \{x\} \cup \{u \in \vec{v}\} \quad fv(\text{def } D \text{ in } P) \triangleq (fv(P) \cup fv(D)) - dv(D) \quad fv(P \mid P') \triangleq fv(P) \cup fv(P')$$

Rules operate on solutions $\mathcal{R} \vdash \mathcal{M}$ consisting of a multiset $\mathcal{M}$ of molecules (active processes) and a multiset $\mathcal{R}$ of reactions (active definitions). There are reversible structural "heating/cooling" rules ($\rightleftharpoons$) and one irreversible reaction rule ($\rightarrow$).

$$\begin{array}{rr}
\mathcal{R} \vdash \mathcal{M}, P \mid Q \rightleftharpoons & \mathcal{R} \vdash \mathcal{M}, P, Q \\
\mathcal{R}, D \vdash E \vdash \mathcal{M} \rightleftharpoons & \mathcal{R}, D, E \vdash \mathcal{M} \\
\mathcal{R} \vdash \mathcal{M}, \text{def } D \text{ in } P \rightleftharpoons & \mathcal{R}, D\sigma_{dv} \vdash \mathcal{M}, P\sigma_{dv} \\
\mathcal{R}, J \rhd P \vdash \mathcal{M}, J\sigma_{rv} \rightarrow & \mathcal{R}, J \rhd P \vdash \mathcal{M}, P\sigma_{rv}
\end{array}$$

Side Conditions:

- $\text{dom}(\sigma_{dv}) \subseteq dv(D)$ and substitutes for these variables distinct, fresh names. Fresh means $\text{rng}(\sigma_{dv}) \cap (fv(\mathcal{R} \cup fv(\mathcal{M} \cup fv(\text{def } D \text{ in } P)) =$
- $\text{dom}(\sigma_{Rv}) \subseteq rv(J)$

2 Join-Calculus

We can think of the CHAM as a computational model derived from process calculi like the π-calculus. We can also go in the opposite direction and produce a process calculus, the join-calculus, from the reflecive CHAM. The terms of the join calculus are just the molecules of the reflexive CHAM, and the structural equivalence and transition rules correspond to the reaction rules of the CHAM.

However, we're going to look at a smaller version of the full join calculus, the core join calculus. It turns out that we lose no expressive power.

$$P \ ::= \ x\langle u\rangle \mid (P_1|P_2) \mid \mathrm{def}\ (x\langle u\rangle|y\langle v\rangle \rhd P_1) \text{ in } P_2$$

The join calculus includes a structural congruence relation ($\equiv$)

$$\begin{aligned}
P|Q &\equiv Q|P \\
P|(Q|R) &\equiv (P|Q)|R \\
P|\mathrm{def}\ D \text{ in } Q &\equiv \mathrm{def}\ D \text{ in } P|Q \\
\mathrm{def}\ D \text{ in def } D' \text{ in } P &\equiv \mathrm{def}\ D' \text{ in def } D \text{ in } P \\
P \equiv_\alpha P' &\implies P \equiv P' \\
P \equiv Q &\implies (P|R) \equiv (Q|R) \\
R \equiv S, P \equiv Q &\implies \mathrm{def}\ J \rhd R \text{ in } P \equiv \mathrm{def}\ J \rhd S \text{ in } Q
\end{aligned}$$

Semantics are given by a labelled transition relation $\xrightarrow{\delta}$ where δ ranges over $D \cup \{\tau\}$. The relation is the smallest such that

- Forall $D = x\langle u\rangle|y\langle v\rangle \rhd R$, we have $x\langle s\rangle|y\langle t\rangle \xrightarrow{D} R[s/x, t/y]$
- If $P \xrightarrow{\delta} P'$, then
  - $P|Q \xrightarrow{\delta} P'|Q$
  - def D in $P \xrightarrow{\delta}$ def D in P' (if $fv(D) \cap dv(\delta) =$)
  - def δ in $P \xrightarrow{\tau}$ def δ in P' (if $\delta \neq \tau$)
  - $Q \xrightarrow{\delta} Q'$ (if $P \equiv Q$ and $P' \equiv Q'$)

2.1 The Join-Calculus and the Reflexive CHAM

Lemma 1. *The structurual congruence $\equiv$ is the smallest congruence that contains all pairs of processes P, Q such that $\vdash P \rightleftharpoons^* \vdash Q$. The silent transition relation $\xrightarrow{\tau}$ contains exactly the pairs of processes P, Q up to $\equiv$ such that $\vdash P \rightarrow \vdash Q$*

2.2 Encodings in the Join-Calculus

2.2.1 CBN Lambda Calculus

$$\begin{aligned}
[\![x]\!]_v &:= x\langle v\rangle \\
[\![\lambda x.T]\!]_v &:= \mathrm{def}\ \kappa\langle x, w\rangle \rhd [\![T]\!]_w \text{ in } v\langle\kappa\rangle \\
[\![TU]\!]_v &:= \mathrm{def}\ x\langle u\rangle \rhd [\![U]\!]_u \text{ in def } w\langle\kappa\rangle \rhd \kappa\langle x, v\rangle \text{ in } [\![T]\!]_w
\end{aligned}$$

The interpretation $[\![T]\!]_v$ means that T should send its value on channel v. A value is a channel κ which, if sent $\langle x, w\rangle$, uses x to look up an argument and sends the result of applying itself to the argument on w. To lookup the value of x, send z to x, and the value of x will be sent on z.

2.2.2 Parallel CBV Lambda Calculus

$$\begin{aligned}
[\![x]\!]_v &:= v\langle x\rangle \\
[\![\lambda x.T]\!]_v &:= \text{def } \kappa\langle x, w\rangle \triangleright [\![T]\!]_w \text{ in } v\langle\kappa\rangle \\
[\![TU]\!]_v &:= \text{def } t\langle\kappa\rangle | u\langle w\rangle \triangleright \kappa\langle w, v\rangle \text{ in } [\![T]\!]_t | [\![U]\!]_u
\end{aligned}$$

This differs in that instead of sending a channel which serves up the unreduced argument, we send the actual argument in the 'application request'. We can do this in the definition of application because we wait until both of the terms in the application have converged to a value.

3 Join-Calculus and the π-calculus

We can think of the join calculus as an asynchronous version of the π-calculus, except with restrictions:

- All binding happens with one construct, the defintion.
- Synchronization only happens with defined names; processes cannot pass messages over free names. Even though one can write on a channel with a free name, communication only occurs when a defined rule executes.
- For every defined name, there is exactly one replicated read. If we think of the calculus in a distributed setting, this means that for every defined name, there is exactly one place here synchronization can occur for that name.

3.1 Asynchronous π-calculus in Join-Calculus

$$P \ ::= \ (P|Q) \mid \text{new } u \text{ in } P \mid \bar{x}\langle u\rangle \mid x(u).P \mid !x(u).P$$

Naive representation of the π-calculus

$$\begin{aligned}
[\![P|Q]\!]_\pi &:= [\![P]\!]_\pi | [\![Q]\!]_\pi \\
[\![\text{new } x \text{ in } P]\!]_\pi &:= \text{def } x_o\langle v_o, v_i\rangle | x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle \text{ in } [\![P]\!]\pi \\
[\![\bar{x}\langle v\rangle]\!]_\pi &:= x_o\langle v_o, v_i\rangle \\
[\![x(v).P]\!]_\pi &:= \text{def } \kappa\langle v_o, v_i\rangle \triangleright [\![P]\!]_\pi \text{ in } x_i\langle\kappa\rangle \\
[\![!x(v).P]\!]_\pi &:= \text{def } \kappa\langle v_o, v_i\rangle \triangleright x_i\langle\kappa\rangle | [\![P]\!]_\pi \text{ in } x_i\langle\kappa\rangle
\end{aligned}$$

But this translation doesn't quite work. For instance, $[\![\bar{x}\langle a\rangle|\bar{x}\langle b\rangle|x(u).\bar{y}\langle u\rangle]\!]_\pi$ is stuck, even though $\bar{x}\langle a\rangle|\bar{x}\langle b\rangle|x(u).\bar{y}\langle u\rangle$ is not stuck in the π-calculus. This is because we have no enclosing $\text{new } x \text{ in }$ ... to provide the definition that allows synchronization to occur.

Additionally, this encoding is not fully abstract in the sense that it is not robust given an arbitrary concurrent context that the encoded process lives in. Even if we make sure all of our free names have been appropriately defined, a channel we are reading on might be written to by some malicious process, with the message being a free name. Again, if we try to do anything to that free name, we become stuck.

We use a tool called the *equator* to fix this problem.

$$M^{\pi}_{x,y} := !x(u).\bar{y}\langle u\rangle | !y(u).\bar{x}\langle u\rangle$$

This is an asynchronous π-calculus program which conflates the two channels x and y. If $M^{\pi}_{x,y}$ is in the 'solution', then no process can tell the difference between them.

To protect the translation from hostile contexts, the encoding must set-up a "firewall" that enforces the protocol. We refine our first approach: each channel x is now represented as several pairs x_o, x_i from the naive encoding that cannot be distinguished from the outside. Two pairs are merged by repeatedly communicating their pending messages to one another. New pairs are defined at run-time according to the following secure protocol:

- Whenever a pair of names is received from the outside, the firewall defines a new, correct proxy pair, merges it to the external pair, and transmits the new pair instead
- Whenever a pair of names is sent to the outside, a new firewall is inserted to set up proxies for future messages on this pair

We use the following contexts to build the firewall on top of the naive translation:

$$\begin{aligned}
\mathcal{P}_x[P] &:= \text{def } x_i\langle v_o, v_i\rangle | x_i\langle \kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle \text{ in def } x_o\langle v_o, v_i\rangle \triangleright p\langle v_o, v_o, x_i\rangle \text{ in } P \\
\mathcal{E}_x[P] &:= \mathcal{P}_x[x_e\langle x_o, x_i\rangle | P] \\
\mathcal{M}[P] &:= \text{def } p(x_o, x_i, \kappa) \triangleright \mathcal{P}_y[\kappa\langle y_o, y_i\rangle | [\![M^{\pi}_{x,y}]\!]_{\pi}] \text{ in } P \\
\mathcal{E}[P] &:= \mathcal{M}[\mathcal{E}_{x_1}[\ldots \mathcal{E}_{x_n}[P]\ldots]] \quad \text{for } x_1, \ldots, x_n = fv(P)
\end{aligned}$$

For every free name x, $\mathcal{P}_x$ encodes the creation of a new proxy for its output. $\mathcal{E}_x$ does the same, and also exports the proxy on a conventional free name x_e. Finally, $\mathcal{M}$ defines the proxy creator p for the whole translation.

Full Abstraction theorem:

Theorem 1. $Q \approx_{\pi} R \iff \mathcal{E}[[\![Q]\!]_{\pi}] \approx \mathcal{E}[[\![R]\!]_{\pi}]$

3.2 Join-Calculus in π-Calculus

Naive representation of the Join-Calculus

$$\begin{aligned}
[\![P|Q]\!]_j &:= [\![P]\!]_j | [\![Q]\!]_j \\
[\![x\langle v\rangle]\!]_j &:= \bar{x}\langle v\rangle \\
[\![\text{def } x\langle u\rangle | y\langle v\rangle \triangleright P \text{ in } Q]\!]_k &:= \text{new } x, y \text{ in } ((!x(u).y(v).[\![P]\!]_j) | [\![Q]\!]_j)
\end{aligned}$$

But this translation, like the previous one, is not robust up to arbitray contexts. We use a similar trick as before. We define a relay process and firewall definitions

$$\begin{aligned}
R_{x,y} &:= !x(v).\text{new } v_e \text{ in } \bar{r}\langle v_e, v\rangle | \bar{y}\langle v_e\rangle \\
\mathcal{R}[P] &:= \text{new } r \text{ in } (!r(x, x_e).R_{x,x_e} | P) \\
\mathcal{E}^{\pi}_x[P] &:= \text{new } x \text{ in } (R_{x,x_e} | P) \\
\mathcal{E}^{\pi}[P] &:= \mathcal{R}[\mathcal{E}^{\pi}_{x_1}[\ldots \mathcal{E}^{\pi}_{x_n}[P]\ldots]] \quad \text{for } x_1, \ldots, x_n = fv(P)
\end{aligned}$$

Full Abstraction theorem:

Theorem 2. $P \approx Q \iff \mathcal{E}^{\pi}[[\![P]\!]_j] \approx_{\pi} \mathcal{E}^{\pi}[[\![Q]\!]_j]$