The Distributed Join-Calculus

Jason Reed

October 23, 2001

The Join Calculus

$$
\begin{array}{llcl}
\text{(str-join)} & \vdash P|Q & \rightleftharpoons & \vdash P, Q \\
\text{(str-null)} & \vdash 0 & \rightleftharpoons & \vdash \\
\text{(str-and)} & D \wedge E \vdash & \rightleftharpoons & D, E \vdash \\
\text{(str-and)} & \top \vdash & \rightleftharpoons & \vdash \\
\text{(str-def)} & \vdash \mathbf{def}\, D \,\mathbf{in}\, P & \rightleftharpoons & D\sigma_{dv} \vdash P\sigma_{dv} \\
\text{(red)} & J \Rrightarrow P \vdash J\sigma_{rv} & \rightarrow & J \Rrightarrow P \vdash P\sigma_{rv}
\end{array}
$$

Side conditions:

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

Imperative Style via CPS (1)

Suppose we have a definition of a 'print server', e.g.

$$\mathbf{def}\, \mathsf{print}\langle x\rangle \Rrightarrow \ldots \mathbf{in}\, \mathsf{print}\langle A\rangle$$

Further suppose we want to print two things in sequence. We could add a continuation argument to print and have the body of the definition of print call it at some point.

$$\mathbf{def}\, \mathsf{print}\,\langle x, \kappa\rangle \Rrightarrow \ldots k\langle\rangle \ldots \mathbf{in}\,\mathbf{def}\, k \Rrightarrow \mathsf{print}\,\langle B, k'\rangle\, \mathbf{in}\, \mathsf{print}\,\langle A, k\rangle$$

But this is really ugly syntactically!

Imperative Style via CPS (2)

What we really want to write is

$$\textbf{def}\ \mathsf{print}(x) \Rrightarrow \ldots \textbf{reply to}\ \mathsf{print} \ldots \textbf{in}\ \mathsf{print}\ A; \mathsf{print}\ B$$

where print has an implicit continuation κ_{print} and **reply to** print calls it, i.e. returns from the function.

We can also extend this notation to handle returns that aren't of type unit.

Imperative Style via CPS (3)

Inside definitions, we make the following syntactic-sugar definitions:

$$\begin{array}{rcll} \mathsf{x}(\vec{v}) & := & x\langle \vec{v}, \kappa_{\mathsf{x}} \rangle & \text{(In join patterns } J\text{)} \\ \mathbf{reply}\,\vec{V}\,\mathbf{to}\,\mathsf{x} & := & \kappa_{\mathsf{x}}\langle \vec{V} \rangle & \text{(In guarded processes } P\text{)} \end{array}$$

More generally, we make the abbreviations

$$\begin{array}{rcl} \mathbf{let}\,\vec{u} = \vec{v}\,\mathbf{in}\,P & := & [\vec{v}/\vec{u}]P \\ \mathbf{let}\,\vec{u} = \mathsf{x}(\vec{V})\,\mathbf{in}\,P & := & \mathbf{def}\,k\langle \vec{u} \rangle \Rrightarrow P\,\mathbf{in}\,x\langle \vec{V}, k \rangle \\ \mathsf{x}(\vec{V}); P & := & \mathbf{def}\,k\langle \rangle \Rrightarrow P\,\mathbf{in}\,x\langle \vec{V}, k \rangle \end{array}$$

Flat Distribution

A DRCHAM is a $\|$ -separated list of CHAMS.

$$\begin{array}{lll} \text{Solutions } S & ::= & D_1, \ldots, D_n \vdash P_1, \ldots, P_m \\ \text{Multisolutions } M & ::= & \cdot \mid (M \parallel S) \end{array}$$

We add a new reduction rule to permit messages to travel to the definition that created them:

$$\text{(comm)} \quad \cdot \vdash x\langle\vec{v}\rangle \parallel J \Rrightarrow P \vdash \cdot \quad \rightarrow \quad \cdot \vdash \cdot \parallel J \Rrightarrow P \vdash x\langle\vec{v}\rangle$$

(under the side condition that $x \in dv(J)$)

This rule illustrates that *the DRCHAM abstracts away the details of message routing* but it exposes synchronization.

Hierarchical Distribution

Assume we have a countably infinite collection of locations, denoted by $\mathcal{L} = \{a, b, c, \ldots\}$. We use $\varphi, \psi, \ldots \in \mathcal{L}^*$ to denote strings over $\mathcal{L}$. We redefine the grammar of multisolutions as follows:

$$\begin{array}{lcl}\text{Solutions } S & ::= & D_1, \ldots, D_n \vdash_\varphi P_1, \ldots, P_m \\ \text{Multisolutions } M & ::= & \cdot \mid (M \parallel S)\end{array}$$

(under the side condition that the solution locations appearing in a valid multisolution are prefix closed and uniquely determined by their last character, if any)

Programming with Locations

Locations are first-class values just like channel names. We add a rule to create new locations:

$$D ::= \ldots \mid a[D : P]$$

The bidirectional heating/cooling rule for this construct is

$$\text{(str-loc)} \quad a[D : P] \vdash_{\varphi} \cdot \quad \rightleftharpoons \quad \cdot \vdash_{\varphi} \cdot \parallel D \vdash_{\varphi a} P$$

(under the side condition that a is 'frozen', i.e. that there are no solutions containing $\vdash_{\varphi a \psi}$ for nonempty ψ)

We also make the definition

$$dv(a[D : P]) = \{a\} \uplus dv(D)$$

So that $\mathbf{def}\, a[D : P]\, \mathbf{in}\, P'$ makes a a *fresh* location name.

Mobility

We add a motion command as new kind of process:

$$P ::= \mathsf{go}\,\langle b, \kappa \rangle$$

Its behavior is given by the new reaction rule

$$\text{(move)} \quad a[D : P|\,\mathsf{go}\,\langle b, \kappa\rangle] \vdash_{\varphi} \cdot \parallel \cdot \vdash_{\psi b} \cdot \quad \rightarrow$$
$$\cdot \vdash_{\varphi} \cdot \parallel a[D : P|\kappa\langle\rangle] \vdash_{\psi b} \cdot$$

By convention, we use go as one would expect if go were a channel name, so that we can say $\mathsf{go}(b); P$ to mean

$$\mathbf{def}\; k\langle\rangle \Rrightarrow P\; \mathbf{in}\, \mathsf{go}\,\langle p, k\rangle$$

Failure (is not an option!)

The authors claim that the semantics of *failure of locations* is of critical importance in making sense of distribution. Their inability to produce a sensible failure semantics for the π-calculus is what drove them to invent the join calculus.

We represent when location a has failed by replacing it with Ωa. Possible failure of a location a is denoted by εa, (That is, εa can be either a or Ωa) and $\varphi, \psi, \ldots$ denote strings over $\{\varepsilon a \,|\, a \in \mathcal{L}\}$ in the sequel.

Failure is modeled by the inability for any communication to leave a location which has failed. This is achieved by adding side conditions on rules (comm) and (move) requiring the source location to be *alive*, i.e. contain no Ω. Note that $\vdash$ is immune to failure, because its location string is empty.

Failure Primitives

New processes:

$$P ::= \cdots \mid \mathsf{fail}\,\langle a, \kappa\rangle \mid \mathsf{halt}\langle\rangle$$

New rules:

$$\begin{array}{llcl}
\text{(halt)} & a[D : P \mid \mathsf{halt}\langle\rangle] \vdash_\varphi \cdot & \rightarrow & \Omega a[D : P] \vdash_\varphi \cdot \\
\text{(detect)} & \cdot \vdash_\varphi \mathsf{fail}\,\langle a, \kappa\rangle \parallel \cdot \vdash_{\psi\varepsilon a} \cdot & \rightarrow & \cdot \vdash_\varphi \kappa \parallel \cdot \vdash_{\psi\varepsilon a} \cdot
\end{array}$$

((detect) occurs under the side condition that $\psi\varepsilon a$ is dead, i.e. contains Ω)

Examples — Remote Printing

$$\mathsf{print}(x) \Rrightarrow \ldots \vdash_s \;\; \| \;\; \vdash_c \mathsf{print}(3); \ldots$$

The (comm) rule causes the $\mathsf{print}\,\langle 3, \kappa \rangle$ message to travel to location s and react, and subsequently the continuation κ is called. By definition of the syntactic sugar, κ is defined at c, so the notification of the completion of printing travels back there.

Examples — Remote Procedure Call

$$\mathsf{f}(y) \Rrightarrow \textbf{reply}\ \mathsf{computation}(y)\ \textbf{to}\ \mathsf{f} \vdash_s$$

$$\parallel\ \vdash_c \textbf{def}\ \mathsf{rpc}(g, x) \Rrightarrow \textbf{reply}\ \mathsf{g}(x)\ \textbf{to}\ \mathsf{rpc}\ \textbf{in} \ldots \mathsf{rpc}(\mathsf{f}, 3) \ldots$$

Note that replying with a synchronous function call amounts to passing along the current continuation.

Examples — The CASA

$$
\begin{array}{l}
\textbf{def}\ c[\mathsf{f}(x,s) \Rrightarrow \\
\quad \textbf{def}\ a[\top : \mathsf{go}(s); \\
\quad\quad \textbf{let}\ \mathsf{get}, \mathsf{set} = \mathsf{new\_cell}(a) \\
\quad\quad \textbf{in}\ \mathsf{set}(\mathsf{computation}(x)); \mathsf{go}(c); \textbf{reply}\ \mathsf{get}\ \textbf{to}\ \mathsf{f}] \\
\quad \textbf{in}\ \ldots : 0] \\
\textbf{in}\ \ldots \mathsf{f}(3, server) \ldots
\end{array}
$$

The client sends an agent to the server to perform computation, and the agent returns to the client's location when the computation is done. All resulting data (which may still be 'live') is stored in the agent's location.

Examples – Code Request

$$\mathsf{load\_applet}(a) \Rrightarrow \mathbf{def}\, b[\mathsf{applet}(y) \Rrightarrow \mathbf{reply} \ldots \mathbf{to}\, \mathsf{applet}$$

$$: \mathsf{go}(a); \mathbf{reply}\, \mathsf{applet}\, \mathbf{to}\, \mathsf{load\_applet}]\, \mathbf{in}\, 0 \vdash_s$$

$$\| \ \vdash_c \mathbf{let}\, \mathsf{f} = \mathsf{load\_applet}(c)\, \mathbf{in} \ldots \mathsf{f}(3) \ldots$$

Here the client requests that an applet be started at a particular location.

Examples – Kill

We can also forcibly cause a running applet to terminate.

$$\mathbf{def}\, b[\mathit{kill}\langle\rangle \Rrightarrow \mathit{halt}\langle\rangle : \mathit{starting\_timer}\langle \mathit{kill}, 5\rangle]$$

$$\mathbf{in}\,\mathbf{let}\, \mathsf{f} = \mathsf{load\_applet}(b)\, \mathbf{in} \ldots \mathsf{f}(3) \ldots$$

The location of the running applet is critical here, since it (the location) may fail.

Examples – Failure Recovery

$$\mathsf{f}(x, s) \Rrightarrow \mathbf{def}\, a[\ldots]\, \mathbf{in}(\mathsf{fail}(a); \mathbf{reply}\, \mathsf{f}(x, s')\, \mathbf{to}\, \mathsf{f})$$

If the agent a ever dies, then the failure continuation can trigger, (recall that $\mathsf{fail}(a); P$ really means $\mathbf{def}\, k \Rrightarrow P\, \mathbf{in}\, \mathsf{fail}\, \langle a, k\rangle$) which creates a new agent and sends it to a different server.

Internal Encoding

Not too surprisingly, the DRCHAM can be implemented in the Join Calculus, but the encoding is somewhat complicated, and uses syntactic sugar which appears to be defined nowhere.

Jocaml

“Jocaml is a system for mobile agents build inside the Objective-Caml Language. Jocaml eases the development of concurrent, distributed and mobile agent based applications, by expressing useful distribution abstractions using a small set of simple but powerful primitives taken from the Join-Calculus.”

(A)synchronous Channels

The syntactic sugar we used before to hide implicit continuations is so useful in practice, that Jocaml makes that style of programming the default!

$$\texttt{let def}\ name(args) = P$$

is the concrete syntax for **def** name$(args)$ **in** P and

$$\texttt{let def}\ !name(args) = P$$

is concrete syntax for the elementary join calculus definition **def** name$\langle args\rangle$ **in** P. The bang is only in the definition not in future use.

This is the distinction between *synchronous* (continuation-carrying) and *asynchronous* (bare) names.