So what's the difference between a session type and an ordinary type anyway?

Frank Pfenning

Computer Science Department
Carnegie Mellon University

Thirty Years of Session Types
October 22, 2023
Apologies for impressionistic style and lack of references

What's not really different?

- Ordinary: data type vs. phrase type
- Session: message type vs. behavioral type
- Ordinary: intuitionistic propositions as simple types
- Session: linear propositions as session types
- Ordinary: preservation and progress
- Session: session fidelity and deadlock freedom

So what is special?

1. Integration of global and local types
   - Global types $\sim$ specifications
   - Local types $\sim$ implementations
2. Substructural (linear or affine) types
   - Reflect process state
   - Channel types evolve during communication

This talk focuses on 2

What have we learned more broadly?

Example: a Store (or Network)

$$\begin{aligned}\mathsf{store}_A = \&\{\ & \mathbf{ins} : A \multimap \mathsf{store}_A, \\ & \mathbf{del} : \oplus\{\ \mathbf{none} : \mathbf{1}, \mathbf{some} : A \otimes \mathsf{store}_A\ \}\ \}\end{aligned}$$

- Typing judgment for processes $\Delta \vdash P :: (x : A)$
  - Process P provides channel x of type A
  - P is client to channels in $\Delta = (x_1 : A_1, \ldots, x_n : A_n)$
- In linear logic / process calculus

prop/type		**provider action**	**continuation**
$A \& B$	external choice	receive choice	A or B
$A \multimap B$	implication	receive channel $a : A$	B
$A \oplus B$	internal choice	send choice	A or B
$A \otimes B$	conjunction	send channnel $a : A$	B
$\mathbf{1}$	unit	send unit	(none)

Type Evolution

$$\text{store}_A = \&\{\, \textbf{ins} : A \multimap \text{store}_A, \\ \qquad\qquad \textbf{del} : \oplus\{\, \textbf{none} : \mathbf{1}, \textbf{some} : A \otimes \text{store}_A \,\}\,\}$$

$$\begin{array}{ll}
\textit{server} :: (s : \text{store}_A) = & \\
\quad \textbf{recv}\ s\ (\,\textbf{ins} \Rightarrow & \%\ s : A \multimap \text{store}_A \\
\qquad\qquad \textbf{recv}\ s\ (x \Rightarrow & \%\ s : \text{store}_A \\
\qquad\qquad \ldots) & \\
\qquad \mid \textbf{del} \Rightarrow & \%\ s : \oplus\{\, \textbf{none} : \mathbf{1}, \textbf{some} : A \otimes \text{store}_A \,\} \\
\qquad\qquad \textbf{send}\ s\ \textbf{some}\ ; & \%\ s : A \otimes \text{store}_A \\
\qquad\qquad \textbf{send}\ s\ y\ ; & \%\ s : \text{store}_A \\
\qquad\qquad \ldots\,) &
\end{array}$$

- Even in a languages like Go, channels have a fixed type
- But see Ferrite session type library for Rust!

Sample Rules (External Choice)

$$\frac{\Delta \vdash P_\ell :: (x : A_\ell) \quad (\forall \ell \in L)}{\Delta \vdash \textbf{recv}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{k \in L \quad \Delta, x : A_k \vdash Q :: (z : C)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash \textbf{send}\ x\ k\ ;\ Q :: (z : C)}\ \&L$$

Preservation and Progress

- A *configuration* is a collection of semantic objects $\mathsf{proc}(P)$
- Dynamics specified using multiset rewriting

$$\begin{array}{rll} & \mathsf{proc}(\textbf{recv}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), & \mathsf{proc}(\textbf{send}\ c\ k\ ;\ Q) \quad (k \in L) \\ \longrightarrow & \mathsf{proc}(P_k), & \mathsf{proc}(Q) \end{array}$$

- Type evolves from $c : \&\{\ell : A_\ell\}$ to $c : A_k$
- Server and client agree on type change
- c is a private channel between the two processes
  - Action is internal to the configuration
- Preservation (= session fidelity) holds
- Progress (= deadlock freedom) also holds

Did we back ourselves into a corner?

- A lot of communication is not synchronous
- A lot of computation is not linear (eg, reuses data)
- A lot of communication is not dyadic (eg, multicast)
- Fortunately, the principles of (local) session types extend
- Generalize from synchronous/linear/dyadic

Step 1: Asynchronous Communication

- Messages as processes
- Requires continuation channels for type safety
- Example: internal choice
  - From

$$\frac{\Delta, x : A_\ell \vdash Q_\ell :: (z : C) \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \textbf{recv}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)} \oplus L$$

  - To

$$\frac{\Delta, x' : A_\ell \vdash Q_\ell(x') :: (z : C) \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \textbf{recv}\ x\ (\ell(x') \Rightarrow Q_\ell(x')) :: (z : C)} \oplus L$$

- Right rule now types a message as process

$$\frac{k \in L}{x' : A_k \vdash \textbf{send}\ x\ k(x') :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})} \oplus R$$

Step 1: Asynchronous Dynamics

- Message has continuation channel
- Receiver has a continuation process

$$\begin{array}{ll} & \mathsf{proc}(\textbf{send}\ c\ k(c')),\ \mathsf{proc}(\textbf{recv}\ c\ (\ell(x') \Rightarrow Q_\ell(x'))_{\ell \in L}) \quad (k \in L) \\ \longrightarrow & \mathsf{proc}(Q_k(c')) \end{array}$$

- We can still track the provenance of a channel
- Ultimately yields data layout, functionally

Example Revisited

$$\text{store}_A = \&\{\, \textbf{ins} : A \multimap \text{store}_A,\; \textbf{del} : \oplus\{\, \textbf{none} : \mathbf{1}, \textbf{some} : A \otimes \text{store}_A \,\}\,\}$$

$$
\begin{array}{ll}
\textit{server} :: (s : \text{store}_A) = & \\
\quad \textbf{recv}\ s\ (\, \textbf{ins}(s') \Rightarrow & \%\ s' : A \multimap \text{store}_A \\
\qquad\qquad \textbf{recv}\ s'\ ((x, s'') \Rightarrow & \%\ s'' : \text{store}_A \\
\qquad\qquad \ldots) & \\
\qquad \mid \textbf{del}(s') \Rightarrow & \%\ s' : \oplus\{\, \textbf{none} : \mathbf{1}, \textbf{some} : A \otimes \text{store}_A \,\} \\
\qquad\qquad \textbf{send}\ s'\ \textbf{some}(s'')\ ; & \%\ s'' : A \otimes \text{store}_A \\
\qquad\qquad \textbf{send}\ s''\ (y, s''')\ ; & \%\ s''' : \text{store}_A \\
\qquad\qquad \ldots\,) &
\end{array}
$$

Step 2: Multicast

- Distinguish linear channels x_{L} and nonlinear channels $\mathbf{x}_{\mathsf{S}}$
- Distinguish ephemeral semantic objects $\mathsf{proc}(P)$, $\mathsf{msg}(P)$ and persistent semantic objects $!\mathsf{msg}(P)$.
  - Ephemeral objects are consumed during transitions
  - Persistent objects are subject to garbage collection
- We can model multicast using persistent messages
- Sample rules: internal choice / sending a label

$$\mathsf{proc}(\mathbf{send}\ c_{\mathsf{L}}\ k(c'_{\mathsf{L}})) \longrightarrow \mathsf{msg}(\mathbf{send}\ c_{\mathsf{L}}\ k(c'_{\mathsf{L}}))$$
$$\mathsf{msg}(\mathbf{send}\ c_{\mathsf{L}}\ k(c'_{\mathsf{L}})), \mathsf{proc}(\mathbf{recv}\ c_{\mathsf{L}}\ (\ell(x'_{\mathsf{L}}) \Rightarrow Q_\ell(x'_{\mathsf{L}}))_\ell) \longrightarrow \mathsf{proc}(Q_k(c'_{\mathsf{L}}))$$

$$\mathsf{proc}(\mathbf{send}\ \mathbf{c}_{\mathsf{S}}\ k(\mathbf{c}'_{\mathsf{S}})) \longrightarrow !\mathsf{msg}(\mathbf{send}\ \mathbf{c}_{\mathsf{S}}\ k(\mathbf{c}'_{\mathsf{S}}))$$
$$!\mathsf{msg}(\mathbf{send}\ \mathbf{c}_{\mathsf{S}}\ k(\mathbf{c}'_{\mathsf{S}})), \mathsf{proc}(\mathbf{recv}\ \mathbf{c}_{\mathsf{S}}\ (\ell(\mathbf{x}'_{\mathsf{S}}) \Rightarrow Q_\ell(\mathbf{x}'_{\mathsf{S}}))_\ell) \longrightarrow \mathsf{proc}(Q_k(\mathbf{c}'_{\mathsf{S}}))$$

Step 2: Shared Service

- Symmetric with multicast
- The server is now persistent, not the message
- Spawns a fresh copy of itself upon message receipt
- Sample rules: external choice / receiving a label

$$\mathsf{proc}(\textbf{recv}\ \mathbf{c}_{\mathsf{S}}\ (\ell(\mathbf{x}'_{\mathsf{S}}) \Rightarrow P_\ell(\mathbf{x}'_{\mathsf{S}}))) \longrightarrow !\mathsf{srv}(\textbf{recv}\ \mathbf{c}_{\mathsf{S}}\ (\ell(\mathbf{x}'_{\mathsf{S}}) \Rightarrow P_\ell(\mathbf{x}'_{\mathsf{S}})))$$

$$!\mathsf{srv}(\textbf{recv}\ \mathbf{c}_{\mathsf{S}}\ (\ell(\mathbf{x}'_{\mathsf{S}}) \Rightarrow P_\ell(\mathbf{x}'_{\mathsf{S}}))), \mathsf{msg}(\textbf{send}\ \mathbf{c}_{\mathsf{S}}\ k(\mathbf{c}'_{\mathsf{S}})) \longrightarrow \mathsf{proc}(P_k(\mathbf{c}'_{\mathsf{S}}))$$

- We can still track provenance

Step 3: Combining Linear and Nonlinear Types

- We use shift to mediate between linear and nonlinear layers

$$\begin{array}{llll} \text{Nonlinear} & A_{\mathsf{S}} & ::= & A_{\mathsf{S}} \rightarrow B_{\mathsf{S}} \mid A_{\mathsf{S}} \times B_{\mathsf{S}} \mid \ldots \mid \uparrow A_{\mathsf{L}} \\ \text{Linear} & A_{\mathsf{L}} & ::= & A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \ldots \mid \downarrow A_{\mathsf{S}} \end{array}$$

- No need to distinguish the syntax of types or processes
- The mode signifies dyadic or variadic channel
- Mode determines:
  - Garbage collection for nonlinear processes and messages
  - No garbage collection for linear processes and messages
- This difference is significant

Taking Stock

- Starting point:
  - Synchronous linear session types
  - Channel type evolves during communication
- Now:
  - Asynchronous session types with continuation channels
  - Combined linear (no gc) and nonlinear (with gc)
  - Types do not evolve, due to continuation channels
  - Provenance can be tracked
- Next:
  - What's the connection to ordinary types?

Process Composition

- Process composition $x_m \leftarrow P(x) \; ; \; Q(x)$
- Dynamics (for linear x and a)

$$\mathsf{proc}(x \leftarrow P(x) \; ; \; Q(x)) \longrightarrow \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)) \qquad a \text{ fresh}$$

- Statics (all variables and propositions linear except Γ_s)

$$\frac{\Gamma_\mathsf{s}, \Delta \vdash A \quad \Gamma_\mathsf{s}, \Delta', A \vdash C}{\Gamma_\mathsf{s}, \Delta, \Delta' \vdash C}\ \mathsf{cut}$$

$$\frac{\Gamma_\mathsf{s}, \Delta \vdash P(x) :: (x : A) \quad \Gamma_\mathsf{s}, \Delta', x : A \vdash Q(x) :: (z : C)}{\Gamma_\mathsf{s}, \Delta, \Delta' \vdash (x \leftarrow P(x) \; ; \; Q(x)) :: (z : C)}\ \mathsf{cut}$$

Compiling Functional Programs

- At this point, session types $\sim$ ordinary types
- Compile functional expressions with a *destination* d

$$[\![e]\!]\, d = P$$

  where $\Gamma \vdash e : A_m$ implies $\Gamma \vdash [\![e]\!]\, d :: (d : A_m)$
- Translation is compositional

$$\begin{array}{lcl} [\![e_1\, e_2]\!]\, d & = & x_1 \leftarrow [\![e_1]\!]\, x_1 \;; \\ & & x_2 \leftarrow [\![e_2]\!]\, x_2 \;; \\ & & \textbf{send}\; x_1\; (x_2, d) \\ \\ [\![\lambda x.\, e]\!] d & = & \textbf{recv}\; d\; ((x, d') \Rightarrow [\![e]\!]\, d') \\ \\ [\![x]\!] d & = & \textbf{fwd}\; d\; x \end{array}$$

- Example

$$[\![\lambda x.\, x]\!]\, d = \textbf{recv}\; d\; ((x, d') \Rightarrow \textbf{fwd}\; d'\; x)$$

Sequential Interpretation

- Parallelism/concurrency is possible, but not necessary
- Example: call-by-need

$$
\begin{aligned}
[\![e_1\ e_2]\!]\ d = {} & x_1 \leftarrow [\![e_1]\!]\ x_1\ ; && \%\ \text{run } [\![e_1]\!]\ x_1 \text{ until it blocks on } x_1 \\
& x_2 \leftarrow [\![e_2]\!]\ x_2\ ; && \%\ \text{suspend } [\![e_2]\!]\ x_2 \\
& \mathbf{send}\ x_1\ (x_2, d) && \%\ \text{pass } x_2 \text{ and } d \text{ to function } x_1
\end{aligned}
$$

$$[\![\lambda x.\, e]\!] d = \mathbf{recv}\ d\ ((x, d') \Rightarrow [\![e]\!]\ d')$$

$$[\![x]\!] d \quad = \mathbf{fwd}\ d\ x$$

- Can also represent call-by-value and futures

Circling back: so what is special?

1. Integration of global and local types
   - Global types ∼ specifications
   - Local types ∼ implementations
2. Substructural (linear or affine) types
   - Reflect process state
   - Channel types evolve during communication
3. Revise and extend
   - Asynchronous communication
   - Continuation channels (with channel provenance)
   - Nonlinear types (shared servers and multicast)
   - Combining linear and nonlinear types
4. Import to "ordinary" functional programming
   - With futures, call-by-need, call-by-value
   - Cannibalized session types for mixed linear/nonlinear types (significant for memory (re)use)
   - Cannibalized continuation channels for data layout

What I have learned

- The significance of linear types
- The significance of mixed linear/nonlinear types
- The elegance of futures
- The connection between channel provenance and data layout

What I still don't know

- Fundamentally, what are global session types?
- How are they connected to local session types?
- What does this mean beyond process communication?