Manifest Sharing with Session Types

Stephanie Balzer and Frank Pfenning

Work in Progress!

Department of Computer Science
Carnegie Mellon University

Theory and Applications of Behavioural Types
Dagstuhl Seminar 17051
February 2, 2017

Background

- Session types rooted in logic [CP'10,CPT'16,W'12]
- Curry-Howard correspondence
  - Linear propositions as session types
  - Sequent proofs as message-passing concurrent programs
  - Proof reduction as communication
- Many threads and contributors
  - Luís Caires, Bernardo Toninho, Jorge Pérez, Dennis Griffith, Henry DeYoung, Limin Jia, Hanna Gommerstadt, Stephanie Balzer, Max Willsey, Coşku Acay, Miguel Silva, Mário Florido
- Implementations
  - Concurrent C0 (Demo Thursday!)
  - SILL (Sessions in Intuitionistic Linear Logic)

SILL: the Judgments

- Interpreting the linear logical judgment and proofs

$$\overset{\mathcal{D}}{A_1, \ldots, A_n \vdash C} \qquad\qquad x_1{:}A_1, \ldots, x_n{:}A_n \vdash P :: (z : C)$$

- Cut as parallel composition (spawn P which provides along fresh channel x)

$$\frac{\Delta \vdash P_x :: (x : A) \quad \Delta', x{:}A \vdash Q_x :: (z : C)}{\Delta, \Delta' \vdash x \leftarrow P_x \,\mathbf{;}\, Q_x :: (z : C)}\ \text{cut}$$

- Identity as forward (x is implemented by y)

$$\frac{}{y{:}A \vdash x \leftarrow y :: (x : A)}\ \text{id}$$

SILL: the Operational Reading

- Every channel has a *provider* and a *client*
- Session is tied to provider's thread of control
- Provider and client perform complementary actions
- From the provider perspective:

$A \multimap B$	receive channel of type A, continue as B
$A \otimes B$	send channel of type A, continue as B
$\mathbf{1}$	send '`end`' and terminate
$\&\{\ell_i : A_i\}_{i \in I}$	receive label ℓ_k, continue as A_k
$\oplus\{\ell_i : A_i\}_{i \in I}$	send label ℓ_k, continue as A_k
$\forall x{:}\tau.B(x)$	receive value v of type τ, continue as $B(v)$
$\exists x{:}\tau.B(x)$	send value v of type τ, continue as $B(v)$

SILL: Examples

- Equirecursive types (must be contractive)

  $$\begin{aligned} \mathsf{list}\ A &= \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : A \otimes \mathsf{list}\ A\} \\ \mathsf{queue}\ A &= \&\{\ \mathsf{enq} : A \multimap \mathsf{queue}\ A, \\ &\qquad\quad \mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{queue}\ A\}\ \} \end{aligned}$$

- Recursive processes implement these interfaces

Sharing

- Programming experience in Concurrent C0 and SILL
  - Many fun and interesting concurrent programs
  - Limitations on shared resources
    - Database, input/output, event loop
- Wadler's challenge
  - Functional programming: recursive types recover the full expressive power of the untyped λ-calculus
  - Concurrent programming: how can we recover the full expressive power of the untyped π-calculus?
  - Note: SILL satisfies session-fidelity (preservation) and deadlock-freedom (global progress)

Of Course!

- $!A$ represents persistence/replication in linear logic
- Operationally [CP'10]

$c : !A$	send fresh persistent channel u along c then provide A along u persistently
$u : A$	receive fresh linear channel y then provide fresh instance of A along y

- This copying semantics is appropriate for pure functional programming, but just supports "read only" sharing

Decomposing $!A$

- $!A$ actually represents two interactions
- Decompose $!A = {\downarrow}^{\mathsf{S}}_{\mathsf{L}} {\uparrow}^{\mathsf{S}}_{\mathsf{L}}\, A$
- Two layers of propositions/types (as in LNL [Benton'94], Adjoint Logic [Reed'09])

$$\begin{array}{llcl} \text{Shared} & A_{\mathsf{S}} & ::= & {\uparrow}^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}} \;\; [\; \mid A_{\mathsf{S}} \to B_{\mathsf{S}} \mid A_{\mathsf{S}} \times B_{\mathsf{S}} \mid \cdots] \\ \text{Linear} & A_{\mathsf{L}} & ::= & {\downarrow}^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}} \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \mathbf{1} \mid \cdots \end{array}$$

- Each modality now is one interaction
- Shifts form an adjunction
- $\downarrow\uparrow$ forms a comonad, $\uparrow\downarrow$ a strong monad

Sharing: Independence Principle

- Shared channels may not depend on linear channels

Providing shared channel	$\Gamma_{\mathsf{S}} \vdash P :: (x_{\mathsf{S}} : A_{\mathsf{S}})$
Providing linear channel	$\Gamma_{\mathsf{S}} \,\mathbf{;}\, \Delta_{\mathsf{L}} \vdash P :: (x_{\mathsf{L}} : A_{\mathsf{L}})$

- Shared channels are typed as if persistent
  - Can be used arbitrarily often
- Proof reduction semantics is still copying

Sharing: Logical Rules

- Omitting shared identity and cut
- Logically, exactly the same as for persistence
- Modal rules, based on independence principle

$$\frac{\Gamma \,;\, \cdot \vdash A_\mathsf{L}}{\Gamma \vdash {\uparrow}_\mathsf{L}^\mathsf{S} A_\mathsf{L}}\ {\uparrow}_\mathsf{L}^\mathsf{S} R \qquad \frac{\Gamma, {\uparrow}_\mathsf{L}^\mathsf{S} A_\mathsf{L} \,;\, \Delta, A_\mathsf{L} \vdash C_\mathsf{L}}{\Gamma, {\uparrow}_\mathsf{L}^\mathsf{S} A_\mathsf{L} \,;\, \Delta \vdash C_\mathsf{L}}\ {\uparrow}_\mathsf{L}^\mathsf{S} L$$

$$\frac{\Gamma \vdash A_\mathsf{S}}{\Gamma \,;\, \cdot \vdash {\downarrow}_\mathsf{L}^\mathsf{S} A_\mathsf{S}}\ {\downarrow}_\mathsf{L}^\mathsf{S} R \qquad \frac{\Gamma, A_\mathsf{S} \,;\, \Delta \vdash C_\mathsf{L}}{\Gamma \,;\, \Delta {\downarrow}_\mathsf{L}^\mathsf{S} A_\mathsf{S} \vdash C_\mathsf{L}}\ {\downarrow}_\mathsf{L}^\mathsf{S} L$$

Sharing: Process Expressions

- From provider perspective

$x_\mathsf{S} : \uparrow_\mathsf{L}^\mathsf{S} A_\mathsf{L}$	accept connection request along x_S continue along linear x_0
$x_i : \downarrow_\mathsf{L}^\mathsf{S} A_\mathsf{S}$	detach from linear client x_i continue along shared x_S

- Matching constructs

$x \leftarrow \mathsf{accept}\ s\ ;\ P_x$	provider $s : \uparrow_\mathsf{L}^\mathsf{S} A_\mathsf{L}$
$x \leftarrow \mathsf{acquire}\ s\ ;\ Q_x$	client $s : \uparrow_\mathsf{L}^\mathsf{S} A_\mathsf{L}$
$s \leftarrow \mathsf{detach}\ x\ ;\ P_s$	provider $x : \downarrow_\mathsf{L}^\mathsf{S} A_\mathsf{L}$
$s \leftarrow \mathsf{release}\ x\ ;\ Q_s$	client $x : \downarrow_\mathsf{L}^\mathsf{S} A_\mathsf{L}$

Example: Producer/Consumer Type

- Earlier purely linear type

$$\begin{aligned}\text{queue } A = \&\{\ & \text{enq} : A \multimap \text{queue } A,\\ & \text{deq} : \oplus\{\text{none} : \mathbf{1}, \text{some} : A \otimes \text{queue } A\}\ \}\end{aligned}$$

- Modify type to make sharing manifest

$$\begin{aligned}\text{squeue } A = {\uparrow}^{\mathsf{S}}_{\mathsf{L}}\ \&\{\ & \text{enq} : A \multimap {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\ \text{squeue } A,\\ & \text{deq} : \oplus\{\ \text{none} : {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\ \text{squeue } A,\\ & \qquad\qquad \text{some} : A \otimes {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\ \text{squeue } A\}\ \}\end{aligned}$$

- Require types to be equi-synchronizing
  - Release (if there is one) is at the same type as acquire
  - On any continuation path from ${\uparrow}^{\mathsf{S}}_{\mathsf{L}}A$ to ${\downarrow}^{\mathsf{S}}_{\mathsf{L}}B$ we have $B = {\uparrow}^{\mathsf{S}}_{\mathsf{L}}A$.
  - Otherwise, dynamic type checking would be required

Example: Producer/Consume Code

$$\begin{array}{l} \text{squeue } A = \uparrow_{\mathsf{L}}^{\mathsf{S}} \& \{ \text{ enq} : A \multimap \downarrow_{\mathsf{L}}^{\mathsf{S}} \text{squeue } A, \\ \qquad\qquad\qquad \text{deq} : \oplus\{ \text{ none} : \downarrow_{\mathsf{L}}^{\mathsf{S}} \text{squeue } A, \\ \qquad\qquad\qquad\qquad\qquad \text{some} : A \otimes \downarrow_{\mathsf{L}}^{\mathsf{S}} \text{squeue } A \} \ \} \end{array}$$

```
elem : squeue A <- A, squeue A
#s <- elem #x #t =
  c <- accept #s
  case c of
  | enq => #y <- recv c
           d <- acquire #t     % begin critical region
           d.enq ; send d #y
           #t <- release d     % end critical region
           #s <- detach c
           #s <- elem #x #t    % recurse
  | deq => c.some ; send c #x
           #s <- detach c
           #s <- #t            % implement #s by #t
```

Example: Producer/Consume Code

$$
\begin{aligned}
\text{squeue } A = {\uparrow}^{\mathsf{S}}_{\mathsf{L}}\ \&\{\ & \text{enq} : A \multimap {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\ \text{squeue } A, \\
& \text{deq} : \oplus\{\ \text{none} : {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\ \text{squeue } A, \\
& \qquad\qquad \text{some} : A \otimes {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\ \text{squeue } A\}\ \}
\end{aligned}
$$

```
empty : squeue A
#s <- empty =
  c <- accept #s
  case c of
   | enq => #y <- recv c
            #s <- detach c
            #e <- empty
            #s <- elem #y #e  % continue as elem #y
   | deq => c.none
            #s <- detach c
            #s <- empty       % continue as empty
```

Typing Rules

- $\uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}}$: accept (provider) and acquire (client)

$$\frac{\Gamma \,\mathbf{;}\, \cdot \vdash P_x :: (x : A_{\mathsf{L}})}{\Gamma \vdash x \leftarrow \mathsf{accept}\ s \,\mathbf{;}\, P_x :: (s : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}})} \uparrow_{\mathsf{L}}^{\mathsf{S}} R$$

$$\frac{\Gamma, s{:}\uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}} \,\mathbf{;}\, \Delta, x{:}A_{\mathsf{L}} \vdash Q_x :: (z : C_{\mathsf{L}})}{\Gamma, s{:}\uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}} \,\mathbf{;}\, \Delta \vdash x \leftarrow \mathsf{acquire}\ s \,\mathbf{;}\, Q_x :: (z : C_{\mathsf{L}})} \uparrow_{\mathsf{L}}^{\mathsf{S}} L$$

Typing Rules

- $\downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}}$: detach (provider) and release (client)

$$\frac{\Gamma \vdash P_s :: (s : A_{\mathsf{S}})}{\Gamma \,;\, \cdot \vdash s \leftarrow \mathsf{detach}\ x \,;\, P_s :: (x : \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}})} \downarrow_{\mathsf{L}}^{\mathsf{S}} R$$

$$\frac{\Gamma, s{:}A_{\mathsf{S}} \,;\, \Delta \vdash Q_s :: (z : C_{\mathsf{L}})}{\Gamma \,;\, \Delta, x{:}\downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}} \vdash s \leftarrow \mathsf{release}\ x \,;\, Q_s :: (z : C_{\mathsf{L}})} \downarrow_{\mathsf{L}}^{\mathsf{S}} L$$

Operational Semantics

- Asynchronous message passing
- Specify with multiset rewriting over predicates

$\mathrm{proc}(x_i, P)$	P provides along linear channel x, generation i
$\mathrm{proc}(x_{\mathsf{S}}, P)$	P provides along shared channel x_{S}
$\mathrm{unavail}(x_{\mathsf{S}})$	no process currently providing x_{S}
$\mathrm{msg}(x_i, M)$	message P is sent along x_i

- Processes cycle through linear (x_i) and shared (x_{S}) phases
- Think of i as index into message buffer for x
- Think of S as lock on shared process

Properties

- Session fidelity (type preservation)
  - Need either equi-synchronous types or runtime checking
  - Proof is somewhat complex due to forwarding
- "Progress"
  - Failure of progress only due to deadlock in acquiring shared channels
- Proofs are in progress as we speak ...

Interpreting the Asynchronous π-Calculus

- A π-calculus channel may have multiple endpoints
- Represent a π channel by a buffer process
- Version 1: single-element buffer of shared type U

$$U = \uparrow_{\mathsf{L}}^{\mathsf{S}} \oplus\{\ \mathsf{none} : \&\{U \multimap \downarrow_{\mathsf{L}}^{\mathsf{S}} U, \mathsf{loop} : \downarrow_{\mathsf{L}}^{\mathsf{S}} U\},\ \mathsf{some} : \&\{U \otimes \downarrow_{\mathsf{L}}^{\mathsf{S}} U, \mathsf{loop} : \downarrow_{\mathsf{L}}^{\mathsf{S}} U\}\ \}$$

- Version 2: multi-element buffer (like shared queue)
  - Use coin flip to model nondeterminism in buffer
  - Coin flip implementable with a simple shared channel
- Process is represented by a worker process of type $\mathbf{1}$
- Replication is modeled by recursion
- Conjecture: constitute asynchronous bisimulation (of some form ...)

Semantics in the Asynchronous π-Calculus

- Use continuations channels as part of every interaction
- Represent a SILL channel by a pair of π-channels (a_i, a_S)
  - a_i is *i*th continuation of linear channel a
  - a_S is associated shared channel
- Conjecture: constitute bisimulation (of some form ...)

Summary and Questions

- Model sharing by decomposing $!A$ into $\downarrow_{\mathsf{L}}^{\mathsf{S}} \uparrow_{\mathsf{L}}^{\mathsf{S}} A$
- Typing rules just as in LNL [Benton'94]
- Depart from the Curry-Howard isomorphism in operational semantics
  - Computation interleaves proof construction and proof reduction
  - Deadlock if proof cannot be constructed
  - Is this a general approach to multi-party session types?
- Expressive enough to interpret the asynchronous π-calculus
- Examples include dining philosophers, producer/consumer, database
- Elegant criteria for freedom from deadlock?