Our sponsors make OPLSS possible.

Diamond Sponsor

https://www.janestreet.com/

Platinum Sponsors

https://www.nsf.gov/

https://www.sigplan.org/

Silver Sponsors

Session Types & Mailbox Types
Lecture 2: Session Types and Functional Programming

Simon Gay
School of Computing Science
University of Glasgow, UK

This lecture

- Combining session types with the standard structures of a higher-order functional language.
- Subtyping for session types, to increase flexibility.

Linear typing (1)

- The fact that most session types are handled linearly forces linearity into the functional constructs of the language.
- We want product types.
- If c is a channel endpoint with a linear session type, and we construct the pair (c,v) for some value v, then sharing the pair also shares c.
- Therefore we introduce the linear product type $T \times_1 U$.
- In a full-scale language we would also want the unrestricted product type $T \times_\omega U$, but we omit it for now.

Linear typing (2)

- Consider a curried function with two parameters, in which the first parameter has a linear type: $\mathsf{f} : !\mathbf{Int}.\mathsf{end}_! \to \mathbf{Int} \to \mathbf{unit}$
- The partial application $\mathsf{f}\ \mathsf{c}$, where c is a channel endpoint of type $!\mathbf{Int}.\mathsf{end}_!$, is a closure of type $\mathbf{Int} \to \mathbf{unit}$ that contains c.
- The definition of f must communicate on c (because of linearity), so sharing $\mathsf{f}\ \mathsf{c}$ would allow multiple communications on c, violating type safety.
- We must therefore distinguish between the linear function type $\mathsf{T} \to_1 \mathsf{U}$ and the unrestricted function type $\mathsf{T} \to_\omega \mathsf{U}$.
- Constructing an unrestricted function requires that there are no linear variables in its closure.
- We can make working with both $\mathsf{T} \to_1 \mathsf{U}$ and $\mathsf{T} \to_\omega \mathsf{U}$ less inconvenient by using subtyping: $\mathsf{T} \to_\omega \mathsf{U} <: \mathsf{T} \to_1 \mathsf{U}$.

Example: online bookshop

Natural protocol:

$$\textbf{type}\ \ \mathsf{Shop} = \&\{\mathsf{add} : ?\,\mathsf{Book}\,.\,\mathsf{Shop}\,,\ \mathsf{checkout} : ?\,\mathsf{Card}\,.\,?\,\mathsf{Address}\,.\,\mathsf{end}_?\}$$

assuming data types Book, Card, Address.

Dual protocol:

$$\textbf{type}\ \ \mathsf{Shopper} = \oplus\{\mathsf{add} : !\,\mathsf{Book}\,.\,\mathsf{Shopper}\,,\ \mathsf{checkout} : !\,\mathsf{Card}\,.\,!\,\mathsf{Address}\,.\,\mathsf{end}_!\}$$

Key idea: communication operations return the channel

- send $v\ c \to c$
  (assuming there is a receiver on a linked endpoint)
- send: $T \to_\omega\ !T.S \to_1 S$
- The channel is returned with its new type.
- receive $c \to (v, c)$
  (assuming there is a sender providing v on a linked endpoint)
- receive: $?T.S \to_\omega\ T \times_1 S$
- select $k\ c \to c$
- select $k : \oplus\{l : S_l\}^{l \in L} \to_\omega S_k$ with $k \in L$
-

$$\frac{\Gamma_1 \vdash e : \&\{l : T_l\}^{l \in L} \qquad \Gamma_2 \vdash e_l : T_l \to_1 T \qquad (\forall l \in L)}{\Gamma_1, \Gamma_2 \vdash \text{match } e \text{ with } \{l : e_l\}^{l \in L} : T} \text{(T-Match)}$$

Example: online bookshop

A shopper, mother, implements the Shopper protocol.

```
mother : Book →ω Card →ω Address →ω Shopper →ω unit
mother book card address toShop =
  let toShop = select add toShop in
  let toShop = send book toShop in
  let toShop = select checkout toShop in
  let toShop = send card toShop in
  let toShop = send address toShop in
  close toShop
```

Each toShop is a fresh bound variable, masking previous occurrences.

Each toShop has a different type, progressing through the protocol.

The programming style can be improved using monads, but I won't talk about that.

Example: online bookshop

The shop implements the Shop protocol.

```
shop : Shop →ω unit
shop s =
  match s with {
    add s →
      let (book, s) = receive s in shop s,
    checkout s →
      let (card, s) = receive s in
      let (address, s) = receive s in
      wait s
  }
```

We are not yet showing how mother and shop are put into communication with each other.

Example: online bookshop

Mother wants to allow her son to choose a book, but without giving him free access to the channel.

She gives him a function of type $\mathsf{Book} \to_1 \mathsf{Book}$ which contains the channel and completes the purchase of one suitable book. This function works as a gift voucher.

Mother sends the gift voucher to her son on a channel of type

$$\textbf{type}\ \ \mathsf{Gift} = !(\mathsf{Book} \to_1 \mathsf{Book}).?\,\mathsf{Book}.\mathsf{end}_!$$

Son uses a dual protocol on that channel:

$$\textbf{type}\ \ \mathsf{Son} = ?(\mathsf{Book} \to_1 \mathsf{Book}).!\,\mathsf{Book}.\mathsf{end}_?$$

Example: online bookshop

The definition of mother:

```
mother : Book →ω Card →ω Address →ω Shopper →ω Gift →1 unit

mother book card address toShop toSon =
  let toSon = send (mkVoucher book card address toShop)
  let (book, toSon) = receive toSon in
  close toSon
```

The definition of son:

```
son : Book →ω Son →ω unit

son book toMother =
  let (voucher, toMother) = receive toMother in
  let toMother = send (voucher book) toMother in
  wait toMother
```

Example: online bookshop

Constructing the gift voucher:

```
mkVoucher :
  Book →ω Card →ω Address →ω Shopper →ω Book →1 Book

mkVoucher mBook card address toShop sBook =
  let toShop = select add toShop in
  let toShop =
    send
      (if isChildrensBook sBook then sBook else mBook)
        toShop in
  let toShop = select checkout toShop in
  let toShop = send card toShop in
  let toShop = send address toShop in
  let _ = close toShop in
  sBook
```

Example: online bookshop

The complete system consists of three threads and two channels.

new creates a pair of channel endpoints of dual types.

fork v creates a new thread, where v : Unit $\rightarrow_1$ Unit is applied to **unit** to give the behaviour of the thread.

This function generates the threads and channels:

```
system : Card →ω Address →ω Book →ω Book →ω unit
system mCard mAddress mBook sBook =
  let (toShopper, toShop) = new Shop in
  let (toSon, toMother) = new Gift in
  fork (λ_ . mother mBook mCard mAddress toShop toSon);
  fork (λ_ . son sBook toMother);
  shop toShopper
```

Serving on a shared channel

We want multiple clients to be able to access one server, each client obtaining a private channel for its session.

```
type ShopServer = ⋆!Shopper
```

abbreviates the recursive type **rec** a.!Shopper.a.

The server creates a Shop channel, sends one endpoint to the client, and keeps the other to run a linear interaction with the client via the function shop defined previously.

```
shopServer : ShopServer →ω unit

shopServer s =
  let (local, remote) = new Shop in
  send remote s;
  fork (λ_ . shop local);
  shopServer s
```

Serving on a shared channel: client side

The code for the mother must also be adapted.

```
motherClient :
  Book →ω Card →ω Address →ω ⋆?Shopper →ω Gift →ω unit

motherClient book card address toServer toSon =
  let (toShop, toServer) = receive toServer in
  mother book card address toShop toSon
```

The **receive** operation returns a pair (toShop,toServer), where toShop is passed to mother and toServer is of type $\star$!Shopper. This type, being unrestricted, can safely be discarded.

Serving on a shared channel: complete system

Type Shop is replaced by ShopServer and we use functions shopServer and motherClient in place of shop and mother.

```
system : Card →ω Address →ω Book →ω Book →ω unit

system mCard mAddress mBook sBook =
  let (toClient, toServer) = new ShopServer in
  let (toSon, toMother) = new Gift in
  fork (λ_ . motherClient mBook mCard mAddress
                                  toServer toSon);
  fork (λ_ . son sBook toMother);
  shopServer toClient
```

Subtyping

The shop introduces an option to remove a book from the basket.

```
type NewShop = &{add: ?Book.NewShop,
                 remove: ?Book.NewShop,
                 checkout: ?Card.?Address.end?}
```

The code now has one more branch in the **match** construct.

```
betterShop : Shop →ω unit
betterShop s =
  match s with {
    add s →
      let (book, s) = receive s in betterShop s,
    remove s →
      let (book, s) = receive s in betterShop s,
    checkout s →
      let (card, s) = receive s in
      let (address, s) = receive s in
      wait s
  }
```

Subtyping

The purpose of subtyping is to allow an old client to interact with the new shop.

We have Shop $<:$ NewShop and this means that from

Shopper $\perp$ Shop

we can conclude that Shopper is also compatible with NewShop.

We will return to subtyping later (if there is time).

Syntax: session types

$$
\begin{array}{lll}
S ::= & & \\
 & \mathsf{end}_? & \text{end wait} \\
 & \mathsf{end}_! & \text{end close} \\
 & ?T.S & \text{input} \\
 & !T.S & \text{output} \\
 & \&\{l\colon S_l\}^{l\in L} & \text{external choice} \\
 & \oplus\{l\colon S_l\}^{l\in L} & \text{internal choice} \\
 & \mathsf{rec}\,X.S & \text{recursive type} \\
 & X & \text{type identifier}
\end{array}
$$

Syntax: functional types and multiplicities

$$
\begin{array}{lll}
T ::= & & \\
 & \text{unit} & \text{unit} \\
 & T \to_m T & \text{function} \\
 & T \times_1 T & \text{product} \\
 & S & \text{session type} \\
 & & \\
m ::= & & \\
 & 1 & \text{linear} \\
 & \omega & \text{unrestricted}
\end{array}
$$

Type equivalence (1)

$$\frac{}{\mathsf{end}_? \simeq \mathsf{end}_?} \quad \text{(Eq-End?)}$$

$$\frac{}{\mathsf{end}_! \simeq \mathsf{end}_!} \quad \text{(Eq-End!)}$$

$$\frac{T \simeq V \qquad U \simeq W}{?T.U \simeq ?V.W} \quad \text{(Eq-?)}$$

$$\frac{T \simeq V \qquad U \simeq W}{!T.U \simeq !V.W} \quad \text{(Eq-!)}$$

$$\frac{T_l \simeq U_l \qquad (\forall l \in L)}{\&\{l\colon T_l\}^{l \in L} \simeq \&\{l\colon U_l\}^{l \in L}} \quad \text{(Eq-\&)}$$

$$\frac{T_l \simeq U_l \qquad (\forall l \in L)}{\oplus\{l\colon T_l\}^{l \in L} \simeq \oplus\{l\colon U_l\}^{l \in L}} \quad \text{(Eq-}\oplus\text{)}$$

Type equivalence (2)

$$\frac{T \simeq U[\text{rec } X.U/X]}{T \simeq \text{rec } X.U} \quad \text{(Eq-RecR)}$$

$$\frac{T_1[\text{rec } X.T_1/X] \simeq U}{\text{rec } X.T_1 \simeq U} \quad \text{(Eq-RecL)}$$

Equivalence can be defined alternatively as subtyping in both directions.

Duality (1)

$$\mathsf{end}_? \perp \mathsf{end}_! \qquad \text{(D-End?)}$$

$$\mathsf{end}_! \perp \mathsf{end}_? \qquad \text{(D-End!)}$$

$$\frac{T \simeq U \qquad R \perp S}{?T.R \perp !U.S} \qquad \text{(D-?)}$$

$$\frac{T \simeq U \qquad R \perp S}{!T.R \perp ?U.S} \qquad \text{(D-!)}$$

$$\frac{R_l \perp S_l \qquad (\forall l \in L)}{\&\{l\colon R_l\}^{l \in L} \perp \oplus\{l\colon S_l\}^{l \in L}} \qquad \text{(D-\&)}$$

$$\frac{R_l \perp S_l \qquad (\forall l \in L)}{\oplus\{l\colon R_l\}^{l \in L} \perp \&\{l\colon S_l\}^{l \in L}} \qquad \text{(D-}\oplus\text{)}$$

Duality (2)

$$\frac{R[\mathsf{rec}\, X.R/X] \perp S}{\mathsf{rec}\, X.R \perp S} \quad \text{(D-RecL)}$$

$$\frac{R \perp S[\mathsf{rec}\, X.S/X]}{R \perp \mathsf{rec}\, X.S} \quad \text{(D-RecR)}$$

Syntax: constants

c ::=	
unit	unit
wait	wait for channel closing
close	close a channel
receive	receive on a channel end
send	send on a channel end
select l	select an option on a channel end
fork	fork a process
fix	fixed point combinator

Syntax: expressions

$e ::=$	
x	variable
c	constant
$\lambda x\colon T.e$	abstraction
$e\,e$	application
(e, e)	pair
$\text{let } (x, x) = e \text{ in } e$	pair split
$\text{new } S$	channel creation
$\text{match } e \text{ with } \{l\colon e_l\}^{l \in L}$	match

Syntax: values

v ::=		
	x	variable
	c	constant
	$\lambda x\colon T.e$	abstraction
	(v, v)	pair of values
	send v	partially applied send

Semantics of expressions: evaluation

The first stage in defining an operational semantics for our functional language is to define the semantics of expressions.

We use a small-step call-by-value operational semantics with evaluation contexts, in the style of Wright and Felleisen.

Semantics of expressions: evaluation contexts

$$
\begin{array}{lll}
E ::= & & \\
 & [\,] & \text{hole} \\
 & E\ e & \text{application left} \\
 & v\ E & \text{application right} \\
 & (E, e) & \text{pair left} \\
 & (v, E) & \text{pair right} \\
 & \text{let } (x, y) = E \text{ in } e & \text{pair split} \\
 & \text{match } E \text{ with } \{l\colon e_l\}^{l \in L} & \text{match}
\end{array}
$$

Semantics of expressions: evaluation

$$(\lambda x\colon T.e)v \to_{\mathsf{v}} e[v/x] \qquad \text{(E-App)}$$

$$\mathsf{let}\ (x, y) = (u, v)\ \mathsf{in}\ e \to_{\mathsf{v}} e[u/x][v/y] \qquad \text{(E-Split)}$$

$$\mathsf{fix}\ (\lambda x\colon T.e) \to_{\mathsf{v}} e[(\mathsf{fix}\ (\lambda x\colon T.e))/x] \qquad \text{(E-Fix)}$$

$$\frac{e \to_{\mathsf{v}} e'}{E[e] \to_{\mathsf{v}} E[e']} \qquad \text{(E-Ctx)}$$

The process layer

Next we define processes, which are built from threads by parallel composition and scope restriction.

A thread is an executing expression.

This layer is similar to pi calculus, but there are no action prefixes because they are in the expression layer.

$$
\begin{array}{lll}
P & ::= & \\
 & \langle e \rangle & \text{thread} \\
 & P \mid P & \text{parallel composition} \\
 & (\nu xy)P & \text{scope restriction}
\end{array}
$$

The variables in scope restriction are runtime syntax, created by evaluation of new.

The process layer: structural congruence

This is adapted from pi calculus.

$$P \mid Q \equiv Q \mid P \qquad \text{(C-Comm)}$$

$$(P \mid Q) \mid M \equiv P \mid (Q \mid M) \qquad \text{(C-Assoc)}$$

$$P \mid \langle \mathsf{unit} \rangle \equiv P \qquad \text{(C-NeutF)}$$

$$(\nu xy)(P \mid Q) \equiv (\nu xy)P \mid Q \qquad \text{(C-Scope)}$$

$$(\nu xy)P \equiv (\nu yx)P \qquad \text{(C-SwapC)}$$

$$(\nu xy)(\nu zw)P \equiv (\nu zw)(\nu xy)P \qquad \text{(C-SwapB)}$$

Semantics of processes: communication

$$(\nu xy)(\langle E[\text{wait } x]\rangle \mid \langle E'[\text{close } y]\rangle) \to \langle E[\text{unit}]\rangle \mid \langle E'[\text{unit}]\rangle \qquad \text{(R-CloseF)}$$

$$(\nu xy)(\langle E[\text{receive } x]\rangle \mid \langle E'[\text{send } v\ y]\rangle \mid M) \to (\nu xy)(\langle E[(v, x)]\rangle \mid \langle E'[y]\rangle \mid M) \qquad \text{(R-ComF)}$$

$$(\nu xy)(\langle E[\text{match } x \text{ with } \{l\colon e_l\}^{l\in L}]\rangle \mid \langle E'[\text{select } k\ y]\rangle \mid M) \to (\nu xy)(\langle E[e_k x]\rangle \mid \langle E'[y]\rangle \mid M) \qquad \text{(R-ChoiceF)}$$

Semantics of processes: fork and new

$$\langle E[\text{fork } v]\rangle \to \langle E[\text{unit}]\rangle \mid \langle v\,\text{unit}\rangle \qquad \text{(R-Fork)}$$

$$\frac{x, y \notin \mathtt{fv}(E)}{\langle E[\text{new } S]\rangle \to (\nu xy)\langle E[(x, y)]\rangle} \qquad \text{(R-New)}$$

Semantics of processes: contextual

$$\frac{e \rightarrow_{\mathsf{v}} e'}{\langle e \rangle \rightarrow \langle e' \rangle} \qquad \text{(R-Thread)}$$

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{(R-Par)}$$

$$\frac{P \rightarrow Q}{(\nu xy)P \rightarrow (\nu xy)Q} \qquad \text{(R-Res)}$$

$$\frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q} \qquad \text{(R-Struct)}$$

Example

$$v_1 = \lambda\_ : \mathsf{unit}.\mathsf{let}\ (\_, x) = \mathsf{receive}\, x\ \mathsf{in}\ \mathsf{wait}\, x$$
$$e_2 = \mathsf{close}\,(\mathsf{send}\,\mathsf{unit}\, y)$$
$$e = \mathsf{let}\ (x, y) = \mathsf{new}\ ?\mathsf{unit}.\mathsf{end}_? \ \mathsf{in}\ \mathsf{fork}\, v_1; e_2$$

First step: creation of channel endpoints.

$$\langle e \rangle \rightarrow (\nu xy)\langle \mathsf{let}\ (x, y) = (x', y')\ \mathsf{in}\ \mathsf{fork}\, v_1; e_2 \rangle \qquad \text{(R-New)}$$

Rename the endpoints (α-conversion):

$$\langle e \rangle \rightarrow (\nu xy)\langle \mathsf{let}\ (x, y) = (x, y)\ \mathsf{in}\ \mathsf{fork}\, v_1; e_2 \rangle.$$
(R-New + R-Struct)

Example

Deriving the next reduction:

$$\dfrac{\dfrac{\dfrac{}{\text{let } (x,y) = (x,y) \text{ in fork } v_1; e_2 \to_{\mathsf{v}} \text{fork } v_1; e_2}\ \text{E-Split}}{\langle \text{let } (x,y) = (x,y) \text{ in fork } v_1; e_2 \rangle \to \langle \text{fork } v_1; e_2 \rangle}\ \text{R-Thread}}{(\nu xy)\langle \text{let } (x,y) = (x,y) \text{ in fork } v_1; e_2 \rangle \to (\nu xy)\langle \text{fork } v_1; e_2 \rangle}\ \text{R-Res}$$

Example

Continuing the reduction sequence:

$$
\begin{aligned}
(\nu xy)\langle \mathsf{fork}\, v_1; e_2\rangle &= (\nu xy)\langle (\lambda\_ : \mathsf{unit}.e_2)\,(\mathsf{fork}\, v_1)\rangle && \text{(R-Fork)}\\
&\to (\nu xy)(\langle (\lambda\_ : \mathsf{unit}.e_2)\,\mathsf{unit}\rangle \mid \langle v_1\,\mathsf{unit}\rangle) && \text{(E-App)}\\
&\to (\nu xy)(\langle e_2\rangle \mid \langle v_1\,\mathsf{unit}\rangle) && \text{(E-App)}\\
&\to (\nu xy)(\langle e_2\rangle \mid \langle \mathsf{let}\ (\_, x) = \mathsf{receive}\, x\ \mathsf{in}\ \mathsf{wait}\, x\rangle) && \text{(R-ComF)}\\
&\to (\nu xy)(\langle \mathsf{close}\, y\rangle \mid \langle \mathsf{let}\ (\_, x) = (\mathsf{unit}, x)\ \mathsf{in}\ \mathsf{wait}\, x\rangle) && \text{(E-Split)}\\
&\to (\nu xy)(\langle \mathsf{close}\, y\rangle \mid \langle \mathsf{wait}\, x\rangle) && \text{(R-CloseF)}\\
&\to \langle \mathsf{unit}\rangle \mid \langle \mathsf{unit}\rangle && \text{(C-NeutF)}\\
&\equiv \langle \mathsf{unit}\rangle
\end{aligned}
$$

Type system

The type system has two layers with two judgements.

Expression typing: $\Gamma \vdash e : T$

Process typing: $\Gamma \vdash P$

Typing contexts Γ have the same format as for pi calculus, with the addition of function and data types.

We have some unrestricted types, which are data types, unrestricted function types and stateless session types.

Typing: constants

$$
\begin{aligned}
\texttt{typeof}(\text{unit}) &= \text{unit} \\
\texttt{typeof}(\text{wait}) &= \text{end}_? \to_\omega \text{unit} \\
\texttt{typeof}(\text{close}) &= \text{end}_! \to_\omega \text{unit} \\
\texttt{typeof}(\text{receive}) &= ?T.S \to_\omega T \times_1 S \\
\texttt{typeof}(\text{send}) &= T \to_\omega !T.S \to_1 S \\
\texttt{typeof}(\text{select}\, k) &= \oplus\{l\colon S_l\}^{l \in L} \to_\omega S_k \text{ with } k \in L \\
\texttt{typeof}(\text{fork}) &= (\text{unit} \to_1 \text{unit}) \to_\omega \text{unit} \\
\texttt{typeof}(\text{fix}) &= (T \to_\omega T) \to_\omega T
\end{aligned}
$$

Typing: expressions (1)

$$x : T \vdash x : T \qquad \text{(T-Var)}$$

$$\vdash c : \texttt{typeof}(c) \qquad \text{(T-Const)}$$

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x : T.e : T \to_1 U} \qquad \text{(T-}\to_1\text{Intro)}$$

$$\frac{\Gamma, x : T \vdash e : U \qquad \Gamma\ \mathtt{un}}{\Gamma \vdash \lambda x : T.e : T \to_\omega U} \qquad \text{(T-}\to_\omega\text{Intro)}$$

$$\frac{\Gamma_1 \vdash e_1 : T \to_1 U \qquad \Gamma_2 \vdash e_2 : T}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : U} \qquad \text{(T-}\to_1\text{Elim)}$$

Typing: expressions (2)

$$\frac{\Gamma_1 \vdash e_1 : T \qquad \Gamma_2 \vdash e_2 : U}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : T \times_1 U} \quad \text{(T-}\times\text{Intro)}$$

$$\frac{\Gamma_1 \vdash e_1 : T \times_1 U \qquad \Gamma_2, x\colon T, y\colon U \vdash e_2 : V}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 : V} \quad \text{(T-}\times\text{Elim)}$$

$$\vdash \mathsf{new}\ S : S \times_1 S^{\perp} \quad \text{(T-New)}$$

$$\frac{\Gamma_1 \vdash e : \&\{l\colon T_l\}^{l \in L} \qquad \Gamma_2 \vdash e_l : T_l \to_1 T \qquad (\forall l \in L)}{\Gamma_1, \Gamma_2 \vdash \mathsf{match}\ e\ \mathsf{with}\ \{l\colon e_l\}^{l \in L} : T} \quad \text{(T-Match)}$$

Typing: expressions (3)

$$\frac{\Gamma \vdash e : U \qquad T \ \mathtt{un}}{\Gamma, x\colon T \vdash e : U} \quad \text{(T-WeakE)}$$

$$\frac{\Gamma, y\colon T, z\colon T \vdash e : U \qquad T \ \mathtt{un}}{\Gamma, x\colon T \vdash e[x/y][x/z] : U} \quad \text{(T-ContrE)}$$

$$\frac{\Gamma \vdash e : T \qquad T <: U}{\Gamma \vdash e : U} \quad \text{(T-SubE)}$$

Typing: processes (1)

$$\frac{\Gamma \vdash e : \mathsf{unit}}{\Gamma \vdash \langle e \rangle} \quad \text{(T-Thread)}$$

$$\frac{\Gamma \vdash P \qquad \Delta \vdash Q}{\Gamma, \Delta \vdash P \mid Q} \quad \text{(T-Par)}$$

$$\frac{\Gamma, x\colon T, y\colon U \vdash P \qquad T \perp U}{\Gamma \vdash (\nu xy)P} \quad \text{(T-Res)}$$

Typing: processes (2)

$$\frac{\Gamma \vdash P \qquad T\ \mathtt{un}}{\Gamma, x\colon T \vdash P} \qquad \text{(T-Weak)}$$

$$\frac{\Gamma, y\colon T, z\colon T \vdash P \qquad T\ \mathtt{un}}{\Gamma, x\colon T \vdash P[x/y][x/z]} \qquad \text{(T-Contr)}$$

Example

Consider a function that sends two channel endpoints of an unrestricted type U on a given channel and returns just the channel.

If T is the type $!U.!U.\mathsf{end}_!$, then the function's type should be $T \to_1 \mathsf{end}_!$ (or it could be $T \to_\omega \mathsf{end}_!$).

$$\lambda c\colon T.\mathsf{let}\ (x, y) = \mathsf{new}\ U\ \mathsf{in}\ \mathsf{send}\ x\ (\mathsf{send}\ x\ c)$$

Example

Work upwards to construct a typing derivation:

$$\dfrac{\dfrac{}{\vdash \mathsf{new}\ U : U \times_1 U^{\perp}} \qquad \dfrac{(*)}{c\colon T, x\colon U, y\colon U^{\perp} \vdash \mathsf{send}\ x\ (\mathsf{send}\ x\ c) : \mathsf{end}_!}}{\dfrac{c\colon T \vdash \mathsf{let}\ (x,y) = \mathsf{new}\ U\ \mathsf{in}\ \mathsf{send}\ x\ (\mathsf{send}\ x\ c) : \mathsf{end}_!}{\vdash \lambda c\colon T.\mathsf{let}\ (x,y) = \mathsf{new}\ U\ \mathsf{in}\ \mathsf{send}\ x\ (\mathsf{send}\ x\ c) : T \to_1 \mathsf{end}_!}\ \text{T-Abs}}\ \text{T-}\times\text{Elim}$$

To continue upwards from $(*)$ we cannot apply rule T-App without first contracting on x, for we would not be able to type both send x and send x c.

Example

$$\dfrac{\dfrac{\vdots \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{c : T \vdash c : T}\,\text{T-Var}}{c : T, y : U \vdash c : T}\,\text{T-WeakE}}{c : T, x'' : U, y : U^{\perp} \vdash \mathsf{send}\ x''\ c : !U.\mathsf{end}_!}\,\text{T-App}}{c : T, x' : U, x'' : U, y : U^{\perp} \vdash \mathsf{send}\ x'\ (\mathsf{send}\ x''\ c) : \mathsf{end}_!}\,\text{T-App}}{c : T, x : U, y : U^{\perp} \vdash \mathsf{send}\ x\ (\mathsf{send}\ x\ c) : \mathsf{end}_!}\,\text{T-ContrE}$$

The $\vdots$ are derivations of

$$x' : U \vdash \mathsf{send}\, x' : !U.\mathsf{end}_! \to_1 \mathsf{end}_!$$

$$x'' : U \vdash \mathsf{send}\, x'' : T \to_1 !U.\mathsf{end}_!$$

Type safety

Similarly to the pi calculus, the strategy is to prove two results.

Absence of immediate errors If $\Gamma \vdash P$ then P is not a runtime error.

Preservation If $\Gamma \vdash P$ and $P \rightarrow Q$ then $\Gamma \vdash Q$.

The details are more complicated because we have to work at the expression layer and then the process layer.

Also similarly to the pi calculus, the type system does not guarantee deadlock-freedom.

Defining runtime errors (1): canonical forms

Canonical forms are processes of the form

$$(\nu x_1 y_1) \ldots (\nu x_n y_n)(\langle e_1 \rangle \mid \ldots \mid \langle e_m \rangle) \qquad (n \geqslant 0, m \geqslant 1).$$

Lemma: Every process is structurally congruent to a process in canonical form.

Defining runtime errors (2): subject / redex

Channel endpoint x is the *subject* of the following processes.

$$\langle E[\text{wait } x]\rangle \quad \langle E[\text{receive } x]\rangle \quad \langle E[\text{match } x \text{ with } \{l\colon e_l\}^{l\in L}]\rangle$$
$$\langle E[\text{close } x]\rangle \quad \langle E[\text{send } v\ x]\rangle \quad \langle E[\text{select } l\ x]\rangle$$

A process $(\nu xy)(\langle E[e]\rangle \mid \langle E'[e']\rangle \mid M)$ is a *redex* if:

- e is wait x and e' is close y, or
- e is receive x and e' is send v y, or
- e is match x with $\{l\colon e_l\}^{l\in L}$ and e' is select k y and $k \in L$,

plus symmetrical cases for $(\nu yx)(\langle E[e']\rangle \mid \langle E[e]\rangle \mid M)$.

P is a redex if it can reduce by R-CloseF, R-ComF or R-ChoiceF.

Defining runtime errors (3)

A process

$$(\nu x_1 y_1) \ldots (\nu x_n y_n)(\langle e_1 \rangle \mid \ldots \mid \langle e_m \rangle)$$

in canonical form is a *runtime error* if there are i, j, k such that

- the subject of $\langle e_i \rangle$ is x_k,
- the subject of $\langle e_j \rangle$ is y_k, and
- $(\nu x_k y_k)(\langle e_i \rangle \mid \langle e_j \rangle)$ is not a redex.

A runtime error is a process in which channel endpoints x_k and y_k are bound together by ν and within the binding there are processes with subjects x_k and y_k but these processes cannot reduce by R-CloseF, R-ComF or R-ChoiceF because the communication operations of their subjects are not complementary.

Absence of immediate errors

Theorem: If $\Gamma \vdash P$ then P is not a runtime error.

Proof: Straightforward, along the lines of the proof for pi calculus.

Preservation

First we need preservation for the expression layer.

Lemma (Substitution):
If $\Gamma_1, x : T \vdash e : U$ and $\Gamma_2 \vdash v : T$ and Γ_1, Γ_2 is defined then $\Gamma_1, \Gamma_2 \vdash e[v/x] : U$.

Proof: By rule induction on $\Gamma_1, x : T \vdash e : U$.

Preservation

Lemma (Typability of Subterms):
If $\Gamma \vdash E[e] : T$ then $\Gamma = \Delta_1, \Delta_2, (\Theta_1, \Theta_2)\sigma$ with Δ_i `lin` and Θ_i `un` and $\Delta_1, \Theta_1 \vdash e' : U$ and $\Delta_2, \Theta_2 \vdash_U E' : T$ and $(E'[e'])\sigma = E[e]$.

Proof: By induction on the structure of E.

The judgement $\Delta_2, \Theta_2 \vdash_U E' : T$ means that E' has type T in context Δ_2, Θ_2, assuming that the hole has type U.

This context typing judgement is defined by rules similar to those for expression typing.

Preservation

Lemma (Replacement):
If $\Gamma_1 \vdash_T E : U$ and $\Gamma_2 \vdash e : T$ and Γ_1, Γ_2 is defined then $\Gamma_1, \Gamma_2 \vdash E[e] : U$.

Proof: By rule induction on $\Gamma_1 \vdash_T E : U$.

Preservation

Lemma (Preservation for Evaluation):
If $\Gamma \vdash e : T$ and $e \rightarrow_{\mathsf{v}} e'$, then $\Gamma \vdash e' : T$.

Proof: By rule induction on $e \rightarrow_{\mathsf{v}} e'$, using the previous lemmas.

Preservation

Lemma (Preservation for $\equiv$):
If $P \equiv Q$ then $\Gamma \vdash P$ if and only if $\Gamma \vdash Q$.

Proof: Similar to the corresponding result for pi calculus.

Preservation

Theorem:
If $\Gamma \vdash P$ and $P \rightarrow Q$ then $\Gamma \vdash Q$.

Proof: By rule induction on $P \rightarrow Q$, using the previous lemmas.

Practical functional programming with session types

There have been several projects over the years to implement session types in functional languages, either as special-purpose languages or as extensions/libraries for existing languages.

Adding session types to existing languages has become easier as those languages have acquired richer type systems.

- Haskell now has linear types.
- Rust has affine types.

Special purpose language: FreeST

The language FreeST, developed by Vasco Vasconcelos and his collaborators, is a functional language designed for programming with session types and as a vehicle for further research on session types.

It uses polymorphic context-free session types, which are an extension of the session types we have discussed.

It has a rich functional type system including higher-order polymorphism.

The examples in this lecture (Chapter 7 of the book) work in FreeST.

`https://freest-lang.github.io/`

Library implementations: FuSE

FuSE is a session type library for OCaml, developed by Luca Padovani.

It combines static checking of session types with dynamic checking of linearity conditions (unique ownership of channels).

FuSE implements polymorphic context-free session types and takes advantage of OCaml type inference.

`github.com/boystrange/FuSe`

Library implementations: Priority Sesh

Priority Sesh is a session type library for Haskell, developed by Wen Kokke.

It uses Haskell's linear types, and uses a system of priorities to guarantee deadlock-freedom.

> *Wen Kokke and Ornela Dardha. Deadlock-free session types in linear Haskell. Haskell Symposium 2021.*

`github.com/wenkokke/priority-sesh`

Orchard and Yoshida survey other Haskell-based implementations.

> *Dominic Orchard and Nobuko Yoshida. Session types with linearity in Haskell. In Simon Gay and António Ravara, editors, Behavioural Types: from Theory to Tools. River Publishers, 2017.*

Library implementations: Ferrite

Ferrite is an implementation of session types in Rust, developed by Ruofei Chen, Stephanie Balzer and Bernardo Toninho.

It's based on a different approach to session types, which we will discuss in the next lecture.

> *Ruofei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite: a judgmental embedding of session types in Rust. ECOOP 2022.*

`drops.dagstuhl.de/entities/document/10.4230/DARTS.8.2.14`

Supplement: subtyping (1)

$$\frac{}{\mathsf{end}_? <: \mathsf{end}_?} \quad \text{(S-End?)}$$

$$\frac{}{\mathsf{end}_! <: \mathsf{end}_!} \quad \text{(S-End!)}$$

$$\frac{T <: U \qquad R <: S}{?T.R <: ?U.S} \quad \text{(S-?)}$$

$$\frac{U <: T \qquad R <: S}{!T.R <: !U.S} \quad \text{(S-!)}$$

Supplement: subtyping (2)

$$\frac{L \subseteq K \qquad (R_l <: S_l)^{l \in L}}{\&\{k : R_k\}^{k \in K} <: \&\{l : S_l\}^{l \in L}} \quad \text{(S-\&)}$$

$$\frac{L \subseteq K \qquad (R_l <: S_l)^{l \in L}}{\oplus\{k : R_k\}^{k \in K} <: \oplus\{l : S_l\}^{l \in L}} \quad \text{(S-}\oplus\text{)}$$

Supplement: subtyping (3)

$$\frac{R[\mathsf{rec}\, X.R/X] <: S}{\mathsf{rec}\, X.R <: S} \quad \text{(S-RecL)}$$

$$\frac{R <: S[\mathsf{rec}\, X.S/X]}{R <: \mathsf{rec}\, X.S} \quad \text{(S-RecR)}$$

Our sponsors make OPLSS possible.

Diamond Sponsor

https://www.janestreet.com/

Platinum Sponsors

https://www.nsf.gov/

https://www.sigplan.org/

Silver Sponsors