Part 3: Type Systems and Concurrency

Einar Broch Johnsen, S. Lizeth Tapia Tarifa, Eduard Kamburjan, Juliane Päßler

November 4, 2023

University of Oslo

Reminder

Setting up a Type System

- A type syntax (T) and a subtyping relation ($T <: T'$)
- A typing environment ($\Gamma : \mathsf{Var} \mapsto T$)
- A type judgment ($\Gamma \vdash \mathtt{s} : T$)
- A set of type rules and a notion of type soundness
- For concurrency: Some notion of splitting the environment and ordering actions

Agenda Today

- Final theoretical lecture on types
- Main ideas behind session types: expressive protocols on channels
- Uniqueness types: linearity for references

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	`chan int`	–	–	Data Safety	Minimal	No communication patterns

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	`chan int`	–	–	Data Safety	Minimal	No communication patterns
Modes	$\texttt{chan}_{?!}\ \texttt{int}$	Implicit in rules	Implicit in rules arbitrary often	–	Minimal Only interfaces	Distinguishes reader from writer

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	`chan int`	–	–	Data Safety	Minimal	No communication patterns
Modes	$\texttt{chan}_{?!}\ \texttt{int}$	Implicit in rules	Implicit in rules arbitrary often	–	Minimal Only interfaces	Distinguishes reader from writer
Linear Types	$\texttt{chan}_{?1,!1}\ \texttt{int}$	Implicit in rules	Implicit in rules once	No DL on single channels	Minimal Only interfaces	Single-use channels, distinguishes reader from writer

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	`chan int`	–	–	Data Safety	Minimal	No communication patterns
Modes	$\texttt{chan}_{?!}\ \texttt{int}$	Implicit in rules	Implicit in rules arbitrary often	–	Minimal Only interfaces	Distinguishes reader from writer
Linear Types	$\texttt{chan}_{?1,!1}\ \texttt{int}$	Implicit in rules	Implicit in rules once	No DL on single channels	Minimal Only interfaces	Single-use channels, distinguishes reader from writer
Usage Types	$\texttt{chan}_{!.?.0+?.!.0}\ \texttt{int}$	Explicit in spec.	Explicit in spec.	–	Considerate No consistency checking	Simple protocols, more than 2 participants

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	`chan int`	–	–	Data Safety	Minimal	No communication patterns
Modes	$\texttt{chan}_{?!}\ \texttt{int}$	Implicit in rules	Implicit in rules arbitrary often	–	Minimal Only interfaces	Distinguishes reader from writer
Linear Types	$\texttt{chan}_{?1,!1}\ \texttt{int}$	Implicit in rules	Implicit in rules once	No DL on single channels	Minimal Only interfaces	Single-use channels, distinguishes reader from writer
Usage Types	$\texttt{chan}_{!.?.0+?.!.0}\ \texttt{int}$	Explicit in spec.	Explicit in spec.	–	Considerate No consistency checking	Simple protocols, more than 2 participants
Binary ST	`chan !int.?string.0`	Implicit at declaration	Explicit in spec.	No DL on single channels	Medium effort Consistency checked	Complex protocols with branching between 2 participants

Binary Session Types

Requirements for Session Types

Session

A *session* is a sequence of related interactions between ≥ 2 parties over a certain time frame.

- Idea: a channel is only used for a single session
- A linear type describes a session with a single interaction
- A usage types describes a complex session and distributes interactions using $+$

Requirements for Session Types

Session

A *session* is a sequence of related interactions between ≥ 2 parties over a certain time frame.

- Idea: a channel is only used for a single session
- A linear type describes a session with a single interaction
- A usage types describes a complex session and distributes interactions using $+$

Requirements for a Type System for Sessions

- Specify precisely possible orders of operations as protocols
- Clarify roles in sessions
- Must be able to handle branching in protocols
- Must be able to send different data types during protocol

Two Views on Channels

1. A channel is a global store, where accesses are synchronizing
    - Each variable points to this store
    - The type of this variable defines a *local view and access point* on it
    - $v : \mathtt{chan}_{?0,!1}\mathtt{int} \rightarrow$ *a global store of integers where I can read once using this access point*
    - Access points are variables

Two Views on Channels

1. A channel is a global store, where accesses are synchronizing
    - Each variable points to this store
    - The type of this variable defines a *local view and access point* on it
    - $v : \mathtt{chan}_{?0,!1}\mathtt{int} \rightarrow$ *a global store of integers where I can read once using this access point*
    - Access points are variables
2. A channel is a global store with at least two access points, where accesses are synchronizing
    - Each variable points to an access point
    - The type of this variable is the type of the access point
    - $v : \mathtt{chan}_{?0,!1}\mathtt{int} \rightarrow$ *a global store of integers where I can read once using this access point*
    - Access points are values

Two Views on Channels

Establishing a Session

Creating a channel results in two values, for two endpoint

$$(\mathtt{x}, \mathtt{y}) \ := \ \mathtt{make}(\mathtt{chan}\ \mathtt{T}_1, \mathtt{chan}\ \mathtt{T}_2)$$

- The values of `x`, `y` have the “same” channel.
- While non-Go, this is the style of channel creation in, e.g., Rust.

Two Views on Channels

Establishing a Session

Creating a channel results in two values, for two endpoint

$$(\mathtt{x}, \mathtt{y}) \ := \ \mathtt{make}(\mathtt{chan}\ \mathtt{T}_1, \mathtt{chan}\ \mathtt{T}_2)$$

- The values of `x`, `y` have the "same" channel.
- While non-Go, this is the style of channel creation in, e.g., Rust.

Binary Session Types

- Make sure types T_1, T_2 match using *duality*
- Channel is used for one session described by T_1, T_2 and completed on termination

Two Views on Channels

Establishing a Session

Creating a channel results in two values, for two endpoint

$$(\mathtt{x}, \mathtt{y}) \;:=\; \mathtt{make}(\mathtt{chan}\ \mathtt{T}_1, \mathtt{chan}\ \mathtt{T}_2)$$

- The values of `x`, `y` have the “same” channel.
- While non-Go, this is the style of channel creation in, e.g., Rust.

History

- Introduced by Kohei Honda in 1992 for binary synchronous sessions
- Extended to multi-party asynchronous setting in 2008
- Can ensure deadlock-freedom
- Various theoretical extensions, implemented for many languages as libraries

Binary Session Types

Type Syntax

- Data type of sent values is now part of protocol/session type
- Additional difference to usage types: no $+$

$T ::= S$	Session Type
$\mid \texttt{chan}\ T$	Channel Type
$\mid D$	Data types
$S ::= \,!T.S$	Send
$\mid ?T.S$	Receive
$\mid 0$	Termination
$\ldots$	(next page)

Session to tend an integer and get some Boolean answer: `chan !int.?bool.0`

Binary Session Types

Choice in Session Types

Choice is not symmetrical in protocols

- One party decides on how to continue the session/protocol
- This choice must be communicated over the (session) channel
- Other party must follow this choice

Reminder: usage types had a symmetrical branching that fails to encode who chooses the branch, e.g., !&?+!&?

- The branch is communicated using a special kind of values: labels
- Session types have two branching operators: internal choice and external choice

Binary Session Types

Type Syntax

$$\begin{array}{rll} S ::= & \ldots & \text{(previous page)} \\ & \mid \ \oplus\{l_1 : S_1, \ldots, l_n : s_n\} & \text{Internal choice} \\ & \mid \ \&\{l_1 : S_1, \ldots, l_n : s_n\} & \text{External choice} \end{array}$$

(repetition $*$ can be added if needed)

- Intuition: The party using $\oplus$ decides on branch and send the label
- Intuition: The party using & receives the label and continues with the corresponding branch

Binary Session Types

Internal Choice

Internal choice models that this endpoint *makes* the choice and communicates it by sending the label.

$$\oplus \left\{ \begin{array}{ll} l_1 : & S_1 \\ \dots & \\ l_n : & S_n \end{array} \right\}$$

- Send l_i and continue with S_i
- All labels must be different
- Also called *active choice* or *selection*

Binary Session Types

External Choice

External choice models that this endpoint *reacts* to the choice after reading it.

$$\& \left\{ \begin{array}{ll} l_1 : & S_1 \\ \ldots & \\ l_n : & S_n \end{array} \right\}$$

- Receive l_i and continue with S_i
- All labels must be different
- Also called *passive choice*

Example

Client

The client send the name of the product (as a string), receives its price (as an integer), and either accepts the offer and sends its address (as a string), or rejects it.

$$!\texttt{string}.?\texttt{int}.\oplus\left\{\begin{array}{ll}\texttt{accept}: & !\texttt{string}.0\\ \texttt{reject}: & 0\end{array}\right\}$$

Example

Client

The client send the name of the product (as a string), receives its price (as an integer), and either accepts the offer and sends its address (as a string), or rejects it.

$$!\mathtt{string}.?\mathtt{int}.\oplus\left\{\begin{array}{ll}\mathtt{accept}: & !\mathtt{string}.0\\ \mathtt{reject}: & 0\end{array}\right\}$$

Server

The server receives the name of the product (as a string), send its price (as an integer), and either receives the address of the client (as a string) upon acceptance, or rejects terminates the session upon rejection.

$$?\mathtt{string}.!\mathtt{int}.\&\left\{\begin{array}{ll}\mathtt{accept}: & ?\mathtt{string}.0\\ \mathtt{reject}: & 0\end{array}\right\}$$

Example

```
func client (){
 (ch1, ch2) :=
   make(chan !string.?int.+{accept: !string.0, reject: 0},
          chan ?string.!int.&{accept: ?string.0, reject: 0})
  go server(ch2)
  ch <- "Types␣and␣Programming␣Languages"
  price := <-ch
  if(price <= 10){
      ch <- accept
      ch <- "Problemveien␣1,␣0313␣Oslo"
  } else ch <- reject
}
```

Example

```
func server(ch chan ?string.!int.&{accept: ?string.0, reject: 0}){
  product : string = <-ch
  ch <- productToPriceMap(product)
  switch <-ch {
    accept -> sendToAddress(<-ch)
    reject -> skip
  }
}
```

- Endpoints must be used by different threads
- How to make sure we declare them in a matching way?
- How to allow subtyping?

Binary Session Types

Duality

- Ensures that both parties communicating over a channel have a *symmetric* or dual view.
- Given a binary session type, we can syntactically construct its dual.
- Alternatively: Given two binary session types, we chan check whether they are duals

$$\begin{aligned}
\overline{0} &= 0 \\
\overline{!T.S} &= ?T.\overline{S} \\
\overline{?T.S} &= !T.\overline{S} \\
\overline{\&\{l_1 : S_1, \ldots, l_n : S_n\}} &= \oplus\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\} \\
\overline{\oplus\{l_1 : S_1, \ldots, l_n : S_n\}} &= \&\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}
\end{aligned}$$

Binary Session Types

Example 1:

$$\overline{!\mathtt{string}.!\mathtt{int}.0} =$$

Binary Session Types

Example 1:

$$\overline{!\mathtt{string}.!\mathtt{int}.0} = ?\mathtt{string}.\overline{!\mathtt{int}.0} =$$

Binary Session Types

Example 1:

$$\overline{\texttt{!string.!int.}0} = \texttt{?string.}\overline{\texttt{!int.}0} = \texttt{?string.?int.}\overline{0} =$$

Binary Session Types

Example 1:

$$\overline{!\texttt{string}.!\texttt{int}.0} = ?\texttt{string}.\overline{!\texttt{int}.0} = ?\texttt{string}.?\texttt{int}.\overline{0} = ?\texttt{string}.?\texttt{int}.0$$

Binary Session Types

Example 1:

$$\overline{!\mathtt{string}.!\mathtt{int}.0} = ?\mathtt{string}.\overline{!\mathtt{int}.0} = ?\mathtt{string}.?\mathtt{int}.\overline{0} = ?\mathtt{string}.?\mathtt{int}.0$$

Example 2:

$$\overline{\oplus \left\{ \begin{array}{l} l_1 : ?\mathtt{int}.\& \left\{ \begin{array}{l} l_4 : !\mathtt{int}.0 \\ l_5 : 0 \end{array} \right\} \\ l_2 : 0 \\ l_3 : !\mathtt{int}.0 \end{array} \right\}} =$$

Binary Session Types

Example 1:

$$\overline{!\mathtt{string}.!\mathtt{int}.0} = ?\mathtt{string}.\overline{!\mathtt{int}.0} = ?\mathtt{string}.?\mathtt{int}.\overline{0} = ?\mathtt{string}.?\mathtt{int}.0$$

Example 2:

$$\overline{\oplus\left\{\begin{array}{l} l_1 : ?\mathtt{int}.\&\left\{\begin{array}{l} l_4 : !\mathtt{int}.0 \\ l_5 : 0 \end{array}\right\} \\ l_2 : 0 \\ l_3 : !\mathtt{int}.0 \end{array}\right\}} = \&\left\{\begin{array}{l} l_1 : \overline{?\mathtt{int}.\&\left\{\begin{array}{l} l_4 : !\mathtt{int}.0 \\ l_5 : 0 \end{array}\right\}} \\ l_2 : \overline{0} \\ l_3 : \overline{!\mathtt{int}.0} \end{array}\right\}$$

$$=$$

Binary Session Types

Example 1:

$$\overline{!\mathtt{string}.!\mathtt{int}.0} = ?\mathtt{string}.\overline{!\mathtt{int}.0} = ?\mathtt{string}.?\mathtt{int}.\overline{0} = ?\mathtt{string}.?\mathtt{int}.0$$

Example 2:

$$\overline{\oplus \left\{ \begin{array}{l} l_1 : ?\mathtt{int}.\& \left\{ \begin{array}{l} l_4 : !\mathtt{int}.0 \\ l_5 : 0 \end{array} \right\} \\ l_2 : 0 \\ l_3 : !\mathtt{int}.0 \end{array} \right\}} = \& \left\{ \begin{array}{l} l_1 : \overline{?\mathtt{int}.\& \left\{ \begin{array}{l} l_4 : !\mathtt{int}.0 \\ l_5 : 0 \end{array} \right\}} \\ l_2 : \overline{0} \\ l_3 : \overline{!\mathtt{int}.0} \end{array} \right\}$$

$$= \& \left\{ \begin{array}{l} l_1 : !\mathtt{int}.\overline{\& \left\{ \begin{array}{l} l_4 : !\mathtt{int}.0 \\ l_5 : 0 \end{array} \right\}} \\ l_2 : 0 \\ l_3 : ?\mathtt{int}.0 \end{array} \right\} =$$

Binary Session Types

Example 1:

$$\overline{!\mathtt{string}.!\mathtt{int}.0} = ?\mathtt{string}.\overline{!\mathtt{int}.0} = ?\mathtt{string}.?\mathtt{int}.\overline{0} = ?\mathtt{string}.?\mathtt{int}.0$$

Example 2:

$$\overline{\oplus\left\{\begin{array}{l} l_1 :?\mathtt{int}.\&\left\{\begin{array}{l} l_4 :!\mathtt{int}.0 \\ l_5 : 0 \end{array}\right\} \\ l_2 : 0 \\ l_3 :!\mathtt{int}.0 \end{array}\right\}} = \&\left\{\begin{array}{l} l_1 : \overline{?\mathtt{int}.\&\left\{\begin{array}{l} l_4 :!\mathtt{int}.0 \\ l_5 : 0 \end{array}\right\}} \\ l_2 : \overline{0} \\ l_3 : \overline{!\mathtt{int}.0} \end{array}\right\}$$

$$= \&\left\{\begin{array}{l} l_1 :!\mathtt{int}.\overline{\&\left\{\begin{array}{l} l_4 :!\mathtt{int}.0 \\ l_5 : 0 \end{array}\right\}} \\ l_2 : 0 \\ l_3 :?\mathtt{int}.0 \end{array}\right\} = \&\left\{\begin{array}{l} l_1 :!\mathtt{int}.\oplus\left\{\begin{array}{l} l_4 :?\mathtt{int}.0 \\ l_5 : 0 \end{array}\right\} \\ l_2 : 0 \\ l_3 :?\mathtt{int}.0 \end{array}\right\}$$

Nested Sessions

Scenario

- Client does not know the address of the buyer, but gets a channel where it will be communicated.
- Server does not read the address, but gets the channel and then reads the address from it

$$!\mathtt{string}.?\mathtt{int}.\oplus\left\{\begin{array}{ll}\mathtt{accept}: & !(\mathtt{chan}\ ?\mathtt{string}.0).0\\ \mathtt{reject}: & 0\end{array}\right\}$$

$$?\mathtt{string}.!\mathtt{int}.\&\left\{\begin{array}{ll}\mathtt{accept}: & ?(\mathtt{chan}\ ?\mathtt{string}.0).0\\ \mathtt{reject}: & 0\end{array}\right\}$$

Duality is not propagated into parameters!

$$\overline{!T.S} \neq ?\overline{T}.\overline{S}$$

Nested Sessions: Server

```
func server(ch
      chan ?string.!int.&{accept: ?(chan ?string.0).0, reject: 0}){
  product : string = <-ch
  ch <- productToPriceMap(product)
  switch <-ch {
    accept -> {
        adCh := <-ch
        sendToAddress(<-adCh)
    }
    reject -> skip
  }
}
```

Nested Sessions: Client

```
func client (adCh chan ?string.0){
  (ch1, ch2) :=
   make(chan !string.?int.+{accept: !(chan ?string.0).0, reject: 0},
        chan ?string.!int.&{accept: ?(chan ?string.0).0, reject: 0})
  go server(ch2)
  ch <- ”Types␣and␣Programming␣Languages”
  price := <-ch
  if(price <= 10){
      ch <- accept
      ch <- adCh
  } else ch <- reject
}
```

Nested Sessions: Another Client

Warning

- Session Types are not for security or privacy modeling
- Modeling a scenario as an abstract protocol only fixes the communication pattern

```
func client (adCh chan ?string.0){
      ...
   (myCh1, myCh2) := make(chan !string.0, chan ?string.0)
   <-adCh
   ...
   if(price <= 10){
         ch <- accept;  ch <- myCh2
         myCh2 <- "Pilestredet␣46,␣0167␣Oslo"
   } else ch <- reject
}
```

Binary Session Types

Establishing a Session with Duality - Type Checking Declaration

$$(\mathtt{x}, \mathtt{y}) \;:=\; \mathtt{make}(\mathtt{chan}\ \mathtt{S}_1, \mathtt{chan}\ \mathtt{S}_2)$$

Where $\mathtt{S}_2 = \overline{\mathtt{S}_1}$.

- Well-formed example:

```
(x, y) = make(chan !int.0, chan ?int.0)
                go func() { x <- 1 }()
                fmt.println(<-y) //prints "1"
```

Binary Session Types

Establishing a Session with Duality - Type Checking Declaration

$$(\mathtt{x}, \mathtt{y}) \;:=\; \mathtt{make}(\mathtt{chan}\ \mathtt{S}_1, \mathtt{chan}\ \mathtt{S}_2)$$

Where $\mathtt{S}_2 = \overline{\mathtt{S}_1}$.

- Ill-formed example

```
//operators mismatch
                (x,y) = make(chan !int.0, chan !int.0)
                //communicated types mismatch
                (x,y) = make(chan !int.0, chan ?string.0)
                //labels mismatch
                (x,y) =
                make(chan &{ok: 0; no: 0}, chan +(ok: 0; label: 0 ))
```

Binary Session Types

- The type is already split into the types for the endpoints
- Environment is split between variables

Typing Environment

Each restricted variable is split into exactly one sub-environment.

$$\begin{aligned} \Gamma_1(x) = \Gamma_2(x) = (\Gamma_1 + \Gamma_2)(x) & \qquad \text{if } \mathrm{un}(\Gamma(x)) \\ (\Gamma_1 + \Gamma_2)(x) = \Gamma_1(x) & \qquad \text{if } \neg\mathrm{un}(\Gamma_1(x)) \text{ and } x \notin \mathbf{dom}\ \Gamma_2 \\ (\Gamma_1 + \Gamma_2)(x) = \Gamma_2(x) & \qquad \text{if } \neg\mathrm{un}(\Gamma_2(x)) \text{ and } x \notin \mathbf{dom}\ \Gamma_1 \end{aligned}$$

Where $\mathrm{un}(T)$ holds if T is a data type or 0.

$$\begin{aligned} & \{x \mapsto 0, y \mapsto !\mathtt{int}.0, z : \mathtt{int}\} \\ = \quad & \{x \mapsto 0, z : \mathtt{int}\} \\ & + \{x \mapsto 0, y \mapsto !\mathtt{int}.0, z : \mathtt{int}\} \end{aligned}$$

Binary Session Types

- Rules are slightly simplified to avoid technical but obvious details
- Using one endpoint requires that the other endpoint was passed to another thread (cf. last lecture)
- No nested session types

Binary Session Types

- Rules are slightly simplified to avoid technical but obvious details
- Using one endpoint requires that the other endpoint was passed to another thread (cf. last lecture)
- No nested session types

Read (Unrestricted Values)

- Uses up a ? read of the fitting data type T
- Usual subtyping for target variable with type T'

$$\frac{\Gamma + \{c : \texttt{chan S}\} \vdash \texttt{s} : \texttt{Unit} \qquad \Gamma \vdash \texttt{v} : T' \qquad T <: T' \qquad \mathsf{un}(T)}{\Gamma + \{c : \texttt{chan ?T.S}\} \vdash \texttt{v} =\!<\!-\texttt{c};\ \texttt{s} : \texttt{Unit}}$$

Binary Session Types

- Rules are slightly simplified to avoid technical but obvious details
- Using one endpoint requires that the other endpoint was passed to another thread (cf. last lecture)
- No nested session types

Write (Unrestricted Values)

- Uses up a ! write of the fitting data type T
- Usual subtyping for sent expression with type T'

$$\frac{\Gamma \vdash \mathtt{e} : T' \qquad \Gamma + \{c : \mathtt{chan\ S}\} \vdash \mathtt{s} : \mathtt{Unit} \qquad T' <: T \qquad \mathtt{un}(T)}{\Gamma + \{c : \mathtt{chan\ !T.S}\} \vdash \mathtt{c} \mathtt{<\!\!-e;}\ \mathtt{s} : \mathtt{Unit}}$$

Binary Session Types

- The rules for parallel operations, termination, assignment/declaration are standard as for the prior systems

Parallel

Starting a new thread splits the environment

$$\frac{\Gamma_2 \vdash \mathtt{s}_2 : \mathtt{Unit} \qquad \Gamma_1 \vdash \mathtt{s}_1 : \mathtt{Unit}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{go}\{\mathtt{s}_1\}; \mathtt{s}_2 : \mathtt{Unit}}$$

Termination

Termination requires that all sessions are either finished, or sent to another thread

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathtt{skip} : \mathtt{Unit}}$$

Example

$$\begin{aligned}
\Gamma &= \{c \mapsto \texttt{chan !int}.0, d \mapsto \texttt{chan ?int}.0, e \mapsto \texttt{int}\}\\
\Gamma^1 &= \{d \mapsto \texttt{chan ?int}.0, e \mapsto \texttt{int}\}\\
\Gamma^2 &= \{c \mapsto \texttt{chan !int}.0, e \mapsto \texttt{int}\}\\
\Gamma^0 &= \{e \mapsto \texttt{int}\}
\end{aligned}$$

$$\dfrac{\dfrac{\vdots}{\Gamma^2 \vdash \texttt{c} \;\; <-\texttt{e};\;\; \texttt{skip} : \texttt{Unit}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{un}(\Gamma^0)}}{\Gamma^0 \vdash \texttt{skip} : \texttt{Unit}} \qquad \dfrac{}{\Gamma^0 \vdash \texttt{e} : \texttt{Int}} \qquad \dfrac{}{\mathsf{un}(\texttt{Int})}}{\Gamma^0 + \{d \mapsto \texttt{chan ?int}.0\} \vdash \texttt{e} \;\; =<-\texttt{d};\;\; \texttt{skip} : \texttt{Unit}}}{\Gamma^1 \vdash \texttt{e} \;\; =<-\texttt{d};\;\; \texttt{skip} : \texttt{Unit}}}{\Gamma \vdash \texttt{go}\ \{\texttt{c} \;\; <-\texttt{e};\;\; \texttt{skip}\};\;\; \texttt{e} \;\; =<-\texttt{d};\;\; \texttt{skip} : \texttt{Unit}}$$

Binary Session Types

- Labels are either enum/own data type/constants (shown here)
- ... or a special primitive (see paper)

Internal Choice

$$\frac{\Gamma + \{c : \texttt{chan}\ S_j\} \vdash \texttt{s} : \texttt{Unit}}{\Gamma + \{c : \texttt{chan}\ \oplus\{\texttt{l}_1 : S_1, \ldots, \texttt{l}_\texttt{j} : S_j, \ldots, \texttt{l}_\texttt{n} : S_n\}\} \vdash \texttt{c} <\!\!-\texttt{l}_\texttt{j};\ \texttt{s} : \texttt{Unit}}$$

External Choice

External choice matches on two operations: reading the label, and picking a branch

$$\frac{\begin{array}{c}\Gamma + \{c : \texttt{chan}\ S_1\} \vdash \texttt{s}_1;\ \texttt{s} : \texttt{Unit} \\ \vdots \\ \Gamma + \{c : \texttt{chan}\ S_n\} \vdash \texttt{s}_\texttt{n};\ \texttt{s} : \texttt{Unit}\end{array}}{\Gamma + \{c : \texttt{chan}\ \&\{\texttt{l}_1 : S_1, \ldots, \texttt{l}_\texttt{n} : S_n\}\} \vdash \texttt{switch} <\!\!-\texttt{c}\{\texttt{l}_1 : \texttt{s}_1; \ldots \texttt{l}_\texttt{n} : \texttt{s}_\texttt{n}\};\ \texttt{s} : \texttt{Unit}}$$

Example

Branching

- For branching, one can consider scoping and split along the .
- Alternatively, no split and no scoping
- For a change, here we show the second way:

$$\frac{\Gamma \vdash \texttt{e} : \texttt{bool} \qquad \Gamma \vdash \texttt{s}_1;\ \texttt{s}_3 : \texttt{Unit} \qquad \Gamma \vdash \texttt{s}_2;\ \texttt{s}_3 : \texttt{Unit}}{\Gamma \vdash \texttt{if(e)}\{\texttt{s}_1;\ \texttt{skip}\}\texttt{else}\{\texttt{s}_2;\ \texttt{skip}\}\ \texttt{s}_3 : \texttt{Unit}}$$

- Note that we do not match on internal choice
- Instead, the internal choice rules picks the branch once the label is sent
- In several branches of the `if`, the same branch of the protocol may be chosen

Example

$$\begin{aligned} \Gamma &= \{\texttt{ch} \mapsto \oplus\{\texttt{accept} : S_1, \texttt{reject} : S_2\}\} \\ \Gamma^1 &= \{\texttt{ch} \mapsto S_1\} \\ \Gamma^2 &= \{\texttt{ch} \mapsto S_2\} \end{aligned}$$

$$\dfrac{\vdots \qquad \dfrac{\dfrac{\vdots}{\Gamma^1 \vdash \texttt{s}_1;\ \texttt{s}_3 : \texttt{Unit}}}{\Gamma \vdash \texttt{ch} < -\texttt{accept};\ \texttt{s}_1;\ \texttt{s}_3 : \texttt{Unit}} \qquad \dfrac{\dfrac{\vdots}{\Gamma^2 \vdash \texttt{s}_2;\ \texttt{s}_3 : \texttt{Unit}}}{\Gamma \vdash \texttt{ch} < -\texttt{reject};\ \texttt{s}_2;\ \texttt{s}_3 : \texttt{Unit}}}{\Gamma \vdash \texttt{if}(\texttt{price} <= \texttt{10})\{\texttt{ch} < -\texttt{accept};\ \texttt{s}_1\}\texttt{else}\{\texttt{ch} < -\texttt{reject};\ \texttt{s}_2\}\ \texttt{s}_3 : \texttt{Unit}}$$

Subtyping and Type Soundness

Subtyping

Subtyping has same idea of duality as the type.

- Internal choice can have more branches
  Intuition: active choice to never take these branches.
- External choice can have less branches
  Intuition: these branches are never chosen anyway.

$$\begin{array}{ll} \oplus\{l_i : S_i\}_{i \in I} <: \oplus\{l_i : S_i'\}_{i \in I'} & \text{iff } I \supseteq I' \wedge \forall i \in I.\ S_i <: S_i' \\ \&\{l_i : S_i\}_{i \in I} <: \&\{l_i : S_i'\}_{i \in I'} & \text{iff } I \subseteq I' \wedge \forall i \in I'.\ S_i <: S_i' \end{array}$$

Subtyping and Type Soundness

- This subtyping allows one to specify interface with the actually implemented behavior and declare channels with the possible behavior.
- Reminder: all external choices must be implemented to be type-safe, but not all internal choices must be!

Subtyping and Type Soundness

- This subtyping allows one to specify interface with the actually implemented behavior and declare channels with the possible behavior.
- Reminder: all external choices must be implemented to be type-safe, but not all internal choices must be!

Example 1

Here, the channel can handle more choices, but one choice (`ask`) is never made,

```
func decide(ch chan +{accept:?string.0, reject:0}) { ... }

    (chl, _) := make(chan +{accept:?string.0,
                            reject:0,
                            ask: !int.?string.0}, ... )
    decide(chl) //declared type is subtype of require type
```

Subtyping and Type Soundness

- This subtyping allows one to specify interface with the actually implemented behavior and declare channels with the possible behavior.
- Reminder: all external choices must be implemented to be type-safe, but not all internal choices must be!

Example 2

Here, the channel can handle less choices, but one choice (`reject`) is never made, so `decided` can implement a branch for it (which is never taken)

```
func decided(ch chan &{accept:?string.0, reject:0}) { ... }

    (chl, _) := make(chan &{accept:?string.0}, ... )
    decide(chl) //declared type is subtype of require type
```

Subtyping and Type Soundness

Type Soundness

Binary session types ensure that the session is lock free: a single session never blocks.

- Session delegation
- Global deadlock freedom require additional analysis

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	`chan int`	–	–	Data Safety	Minimal	No communication patterns
Modes	`chan`$_{?!}$ `int`	Implicit in rules	Implicit in rules arbitrary often	–	Minimal Only interfaces	Distinguishes reader from writer
Linear Types	`chan`$_{?1,!1}$ `int`	Implicit in rules	Implicit in rules once	No DL on single channels	Minimal Only interfaces	Single-use channels, distinguishes reader from writer
Usage Types	`chan`$_{!.?.0+?.!.0}$ `int`	Explicit in spec.	Explicit in spec.	–	Considerate No consistency checking	Simple protocols, more than 2 participants
Binary ST	`chan !int.?string.0`	Implicit at declaration	Explicit in spec.	No DL on single channels	Medium effort Consistency checked	Complex protocols with branching between 2 participants

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	`chan int`	–	–	Data Safety	Minimal	No communication patterns
Modes	`chan`$_{?!}$ `int`	Implicit in rules	Implicit in rules arbitrary often	–	Minimal Only interfaces	Distinguishes reader from writer
Linear Types	`chan`$_{?1,!1}$ `int`	Implicit in rules	Implicit in rules once	No DL on single channels	Minimal Only interfaces	Single-use channels, distinguishes reader from writer
Usage Types	`chan`$_{!.?.0+?.!.0}$ `int`	Explicit in spec.	Explicit in spec.	–	Considerate No consistency checking	Simple protocols, more than 2 participants
Binary ST	`chan` $!$`int`$.?$`string`$.0$	Implicit at declaration	Explicit in spec.	No DL on single channels	Medium effort Consistency checked	Complex protocols with branching between 2 participants
Multi-Party ST	`chan` $p \to q :$ `int`$.0$	Extra mechanism	Explicit in spec.	No DL on single channels	Considerate Consistency checked	Complex protocols with branching between n participants

Multi-Party Session Types

Multi-Party Session Types

Motivation

Multi-Party Session Types (MPST) generalize to situation with more than two parties.

- How can more than 2 parties communicate on a channel?
- How can we specify such protocols?
- How can we generalize duality?

Multi-Party Session Types

Motivation

Multi-Party Session Types (MPST) generalize to situation with more than two parties.

- How can more than 2 parties communicate on a channel?
- How can we specify such protocols?
- How can we generalize duality?

- We require an operation to wait on a label, so 2 parties can synchronize while the others wait

```
c <- l,e //write label l and value e
l,e <- c //read value e, once l is send
```

Multi-Party Session Types

Motivation

Multi-Party Session Types (MPST) generalize to situation with more than two parties.

- How can more than 2 parties communicate on a channel?
- How can we specify such protocols?
- How can we generalize duality?

- We require an operation to wait on a label, so 2 parties can synchronize while the others wait

  ```
  c <- l,e //write label l and value e
  l,e <- c //read value e, once l is send
  ```

- Task: Specify, implement and check scenario with three participants (Alice, Bob, Carol), which pass an integer token in a ring.

Multi-Party Session Types

Roles

A role is a point of view on the session and corresponds to one endpoint of the channel. Consequently, a channel can now have *n* endpoints, not just two.

Global and Local Types

MPST uses two kinds of specifications/types

- Global types give an overview on the whole session from a global view
- Global types describe at what point communication takes place between at least 2 parties
- Local types describe the session from the view of a single role
- Local types do not contain communication not visible to the considered role

Multi-Party Session Types

Type Syntax

Unify all constructs into one type expressing that p sends a label l_i together with a data value of type T_i to q, and the communication continues as S_i.

$$S ::= 0 \mid p \to q : \{l_1(T_1) : S_1, \dots, l_n(T_n) : S_n\}$$

We omit the outermost parentheses if $n = 1$.

Multi-Party Session Types

Type Syntax

Unify all constructs into one type expressing that p sends a label l_i together with a data value of type T_i to q, and the communication continues as S_i.

$$S ::= 0 \mid p \to q : \{l_1(T_1) : S_1, \ldots, l_n(T_n) : S_n\}$$

We omit the outermost parentheses if $n = 1$.

$$\text{Alice} \to \text{Bob} : \left\{ l_1(\texttt{int}) : \text{Bob} \to \text{Carol} : \left\{ \begin{array}{ll} l_2(\texttt{int}) : & \text{Carol} \to \text{Alice} : l_4(\texttt{int}).0 \\ l_3(\texttt{int}) : & \text{Carol} \to \text{Alice} : l_4(\texttt{int}).0 \end{array} \right\} \right\}$$

Multi-Party Session Types

Local Types

Two actions, which are the unification of internal choice and sending, and the unification of external choice and receiving.

$$
\begin{aligned}
L ::= & 0 \\
& \mid \&\{p_1?l_1(T_1).L_1, \ldots, p_n?l_n(T_n).L_n\} \\
& \mid \oplus\{q_1!l_1(T_1).L_1, \ldots, q_n!l_n(T_n).L_n\}
\end{aligned}
$$

Multi-Party Session Types

Local Types

Two actions, which are the unification of internal choice and sending, and the unification of external choice and receiving.

$$\begin{aligned} L ::= & 0 \\ & \mid \&\{p_1?l_1(T_1).L_1, \ldots, p_n?l_n(T_n).L_n\} \\ & \mid \oplus\{q_1!l_1(T_1).L_1, \ldots, q_n!l_n(T_n).L_n\} \end{aligned}$$

$$L_{\text{Alice}} = \text{Bob}!l_1(\texttt{int}).\text{Carol}?l_4(\texttt{int}).0$$

$$L_{\text{Bob}} = \text{Alice}?l_1(\texttt{int}). \oplus \left\{ \begin{array}{c} \text{Carol}!l_2(\texttt{int}).0 \\ \text{Carol}!l_3(\texttt{int}).0 \end{array} \right\}$$

$$L_{\text{Carol}} = \& \left\{ \begin{array}{c} \text{Bob}?l_2(\texttt{int}).\text{Alice}!l_4(\texttt{int}).0 \\ \text{Bob}?l_3(\texttt{int}).\text{Alice}!l_4(\texttt{int}).0 \end{array} \right\}$$

Multi-Party Session Types

- Duality is generalized to *projection*: generate a local type for each role from a global type
- When projecting on receiver q, turn $p \to q$ into a ?
- When projecting on sender p, turn $p \to q$ into a !
- When projecting on any one else, each branch must be the same – this party is not communicated which branch is taken

Projection

Generate local type $L_p = G \upharpoonright p$ from global type G and role p

$$0 \upharpoonright p = 0$$

$$p \to q : \{l_1(T_1) : S_1, \ldots, l_n(T_n) : S_n\}_{\upharpoonright} p = \oplus\{q!l_1(T_1).(S_1 \upharpoonright p), \ldots, q!l_1(T_1).(S_1 \upharpoonright p)\}$$

$$p \to q : \{l_1(T_1) : S_1, \ldots, l_n(T_n) : S_n\} \upharpoonright q = \&\{p?l_1(T_1).(S_1 \upharpoonright q), \ldots, p?l_n(T_n).(S_n \upharpoonright q)\}$$

$$p \to q : \{l_1(T_1) : S_1, \ldots, l_n(T_n) : S_n\} \upharpoonright r = L \text{ where } \forall i.\ L = S_i \upharpoonright r$$

Multi-Party Session Types

- Not all labels must be different
- Same guarantees as binary types, and *Session Fidelity*:
  A session indeed follows the communication by the global type.
- Type systems assigns role upon a new parallel process.
- Vast number of extensions, variants, alternative designs, implementations, systems for different concurrency models (including actors), synchronous and asynchronous communication
- Tightly connected to choreographic programming: given a protocol, generate a program that implements it

Uniqueness Types

Uniqueness Types

Linear Types

So far, we have used linear types for channels, and limited the use of *reading* and *writing*. Passing the channel around was no problem, but required to split the type environment.

- What about other kinds of usages?
- What about limiting the use of *passing*?

Uniqueness Types

A uniqueness type system ensures that every value (channel etc.) has at most one usable reference pointing to it.

- Sometimes used interchangeably with linear types, when every use of a variable is considered creating a new reference

Uniqueness Types

Every value is associated with a single variable

```
i : T = ...;
j : T = i;  //uses up i
k : T = i;  //error
```

Calls are considered creating a new, external reference

Uniqueness Types

Every value is associated with a single variable

```
i : T = ...;
j : T = i;  //uses up i
k : T = i;  //error
```

Calls are considered creating a new, external reference

```
func drive(param T) = ...
car := ...
drive(car); //passes reference out, uses up car
drive(car); //error
```

Uniqueness Types

Every value is associated with a single variable

```
i : T = ...;
j : T = i;  //uses up i
k : T = i;  //error
```

Calls are considered creating a new, external reference

```
car := ...
car = drive(car); //passes reference out, uses up car
                  //gets new reference!
drive(car); //allowed
```

Uniqueness Types

Uniqueness and Threads

In a concurrent setting, uniqueness types are used to ensure that only one thread has access to a shared resource.

Trivially removes data races, as only one thread can modify an any time – all other references are considered used up.

```
car := ...
go { car.wheels = 5 } //passes reference out
                      //also uses up car!
car.wheels = 6;       //error
```

Uniqueness Types

- Type system is a variant of affine/linear types
- Each type T has now the form T_1 or T_0
- Split again operates on the parameter: $T_{m+n} = T_n + T_m$
- Every use (read, write) requires T_1
- Split on parallel operator

Wrap-Up

- How to set up a type system
- How types mirror reasoning about concurrent systems
- Communication patterns expressed by the different type systems

Next Lecture: Rust

- Linear and uniqueness: Rust and concurrency in Rust
- Session types in practice with a rust library

Wrap-Up

- How to set up a type system
- How types mirror reasoning about concurrent systems
- Communication patterns expressed by the different type systems

Next Lecture: Rust

- Linear and uniqueness: Rust and concurrency in Rust
- Session types in practice with a rust library

No exercise session this week, next exercise will be uploaded end of the week