Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Inferencia de tipos sesión probabilísticos

Tesis de Licenciatura en Ciencias de la Computación

Iván Pondal

Director: Hernán Melgratti

Buenos Aires, 2021

INFERENCIA DE TIPOS SESIÓN PROBABILÍSTICOS

Los últimos años atestiguan un auge en el desarrollo de técnicas de descripción de interfaces y soporte a nivel de lenguajes de programación para el desarrollo de aplicaciones correctas por construcción. El desarrollo de tipos comportamentales y, en especial, los tipos de sesión se han consolidado como un formalismo central para el análisis modular de aplicaciones distribuidas basadas en procesos que comunican a través de canales.

En [IMP+20] se propuso el uso de tipos de sesión para razonar probabilísticamente sobre propiedades de alcanzabilidad. Aquel trabajo presenta un sistema de chequeo de tipos que permite determinar la probabilidad con la que una sesión en particular termina exitosamente.

La investigación presenta una implementación del sistema de tipos probabilísticos presentado en [IMP+20] extendiendo FuSe [Pad17], una implementación de tipos sesión escrita en OCaml.

Palabras claves: tipos sesión probabilísticos, validación, tipos comportamentales, inferencia.

AGRADECIMIENTOS

A lo largo de este camino, compartí todo tipo de momentos y experiencias de las cuales siempre algo aprendí. Esto se lo debo a todas las personas que fueron parte del proceso, sin quienes no habría llegado hasta acá.

Comienzo por agradecer al excelente cuerpo de profesores/as, ayudantes, docentes; no solo me llevo sus lecciones, también los momentos de humor, empatía y principalmente vocación.

A toda mi familia, siempre bancando los trapos; fuera bien, fuera mal. La compañía durante largos domingos de estudio, las sobras de la cena, preguntar por qué tanto lío por unos dibujos con flechas o cuánto falta para que nos dominen los algoritmos.

Después está la segunda familia, la que se elige. Porque entre mates y cafés, alguno sugirió un fútbol cinco y todos dijimos que sí. Espero otros se diviertan tanto como yo con este grupo que hizo los momentos difíciles más amenos y los buenos siempre motivo de festejo.

A Jess, quien me crucé ya para el final del camino; animándome en el último pique y acompañando tanto los días buenos como los malos. Por su paciencia y escucharme siempre, hasta cuando hablo incoherencias.

Por último, gracias a Hernán que de principio a fin me guió en este tramo con una constancia y profesionalismo al cual solo puedo aspirar.

A mi abuelo, con tiempo y paciencia

Índice general

1. Introducción 1

2. Tipos sesión 3
   2.1. Introducción 3
   2.2. Gramática 3
   2.3. Interfaz programática 4
   2.4. Ejemplos básicos 5

3. Tipos sesión probabilísticos 7
   3.1. Gramática 7
   3.2. Interfaz programática 8
      3.2.1. Modificación al sistema de tipos de FuSe 8
   3.3. Ejemplos básicos 9

4. Implementación 11
   4.1. Codificación 11
      4.1.1. Codificación de probabilidad 13
      4.1.2. Representando naturales y racionales 13

5. Elecciones multi-sesión 15
   5.1. Limitaciones con `pick` de una sesión 15
   5.2. Combinador de tipo probabilístico 16
   5.3. Reglas de tipado para elecciones 16
   5.4. Extensión para elecciones multi-sesión 18

6. Composición de tipos sesión duales 20
   6.1. Ejemplos de uso 20
   6.2. Interacción con sesiones multi-sesión 21

7. Probabilidad de éxito de una sesión 24
   7.1. Extendiendo decodificación para tipos sesión probabilísticos 24
   7.2. Calculando probabilidad de absorción 25

8. Conclusión 29

1. INTRODUCCIÓN

El desarrollo de software distribuido, ya sea en sus versiones tradicionales como en sus variantes más recientes, tales como el *Cloud/Fog Computing* o la Internet de las cosas (IoT por su sigla en inglés), se enfoca principalmente en la integración, cooperación y comunicación de componentes, generalmente existentes y desarrollados independientemente.

Mas allá del desafío tecnológico que tal integración presupone, el problema principal, desde el punto de vista del desarrollo, es razonar sobre las propiedades de tales sistemas en términos de las propiedades de sus componentes y de los mecanismos utilizados para su composición.

Esto último dio lugar al desarrollo de *técnicas de descripción* de interfaces y de *soporte a nivel de lenguajes de programación para el desarrollo de aplicaciones correctas por construcción*. Los tipos comportamentales, tales como los tipos de sesión o *type-states* constituyen un ejemplo paradigmático de esta línea de trabajo. La idea central consiste en la utilización de sistemas de tipo (como extensión de los utilizados en los lenguajes existentes) para garantizar estáticamente propiedades relacionadas con la interacción de componentes.

Los últimos años vieron un auge en el desarrollo de tipos comportamentales dentro de los cuales los *tipos de sesión* se mostraron como un formalismo central para el estudio de aplicaciones distribuidas basadas en procesos que se comunican a través de canales (introducción en [HLV$^+$16]). A continuación listamos artículos publicados en los úlitmos años en las mayores conferencias del área [GPP$^+$21, HBK20, CY20, INYY20, Fow20, JY20, VCAM20, Hor20, DP20, DBH$^+$21, MFYZ21, GHK$^+$20, MYZ20, IMP$^+$20].

Los tipos de sesión fueron originalmente propuestos en [Hon93], y proponen estructurar la comunicación entre componentes alrededor del concepto de *sesión*. Una sesión es un canal privado que permite conectar a dos (a veces más) procesos, donde cada uno posee un *endpoint* de la sesión y lo debe usar siguiendo un protocolo bien especificado, llamado *tipo de sesión*, que restringe la secuencia de mensajes que se pueden enviar y recibir a través de ese endpoint. Dependiendo de si el mecanismo de tipado es estático o dinámico, el tipo sesión se utiliza durante la compilación o la ejecución para asegurar que un programa utiliza el canal de sesión de acuerdo a su tipo. Como en cualquier sistema de tipos, un programa bien tipado goza de las propiedades que garantiza el sistema propuesto. En general, se asegura la propiedad de *type-safety* (es decir los procesos intercambian información que tiene el tipo esperado), la adherencia al protocolo (las secuencias de acciones sobre un canal siguen el orden descripto por el protocolo), pero también pueden garantizar propiedades como la ausencia de *deadlocks* y *livelocks* [HLV$^+$16].

En [IMP$^+$20] se propuso el uso de tipos de sesión para razonar probabilísticamente sobre propiedades de alcanzabilidad, donde se dio una interpretación *probabilística* a los operadores de selección. Más específicamente, se pasa de una interpretación no determinística a una probabilística y se estudia un sistema de tipos que permita determinar la probabilidad con la que una sesión en particular termina *exitosamente*. Dado que no existe una interpretación universal de “terminación exitosa”, se diferencia la terminación exitosa de la infructuosa a través de un constructor dedicado.

Si bien la línea trabajo sobre tipos sesión tiene raíces en resultados teóricos sobre modelos fundacionales de la computación concurrente, como los son CCS y el cálculo π, ac-

tualmente existen extensiones concretas de sistemas de tipos de lenguajes de programación de escala industrial, tales como Scala [SYB19], Dotty [SYB19], Go [LNTY18, LNTY18], Rust [Kok19, LNY20], Haskell [OY17, LM16], OCaml [Pad17, LNY20, INYY20], Erlang [Fow16] y han sido aplicados, entre otros, al estudio de *smart contracts* y tecnologías *blockchain* [COE+20, DBHP19].

Este no es el caso para los tipos sesión probabilísticos, dado que la propuesta en [IMP+20] presenta un sistema de chequeo de tipos, pero deja abierta la definición de un algoritmo de inferencia y su implementación en un lenguaje de programación. En [DWH20] se presentó una variante de tipos sesión probabilísticos *resource-aware* [DHP18], el mismo cuenta con la implementación de un nuevo lenguaje desarrollado para demostrar las aplicaciones del sistema de tipos pero no cuenta con una extensión para lenguajes de programación existentes.

En este trabajo presentamos una implementación del sistema de tipos probabilísticos presentado en [IMP+20]. La misma es una extensión de `FuSe` [Pad17], una biblioteca escrita en `OCaml` que permite modelar tipos sesión mediante una codificación que no depende de funcionalidades avanzadas del lenguaje anfitrión. La elección fue motivada por la modularidad del proyecto y codificación que permitieron una extensión al cálculo probabilístico sin perder las garantías del sistema original. El código de la extensión se encuentra disponible en un repositorio `git` público.[1]

Contribuciones y estructura del trabajo.

- Extendimos la codificación de tipos sesión presente en `FuSe` para capturar la gramática de los tipos sesión probabilísticos y su combinación (Capítulo 3). La misma permite usufructuar el motor de inferencia estándar de `OCaml` sin necesidad de modificarlo.
- Modificamos las primitivas de comunicación de `FuSe` para incorporar elecciones probabilísticas y poder diferenciar una terminación exitosa de la fallida (Capítulo 4, Capítulo 5 y Capítulo 6). Se considerarán principalmente elecciones binarias (evitando el uso de elecciones generalizadas en término de variantes polimórficas).
- Utilizamos los tipos inferidos por el compilador anotados con elecciones probabilísticas para calcular la probabilidad de terminación exitosa de una sesión mediante el cómputo de la probabilidad de absorción de la cadena de Markov asociada al tipo de sesión inferido (Capítulo 7).
- Implementamos un conjunto de ejemplos básicos, algunos presentes en [IMP+20], para ilustrar la aplicabilidad de la propuesta.

[1] `https://github.com/ivanpondal/probabilistic-binary-sessions`

2. TIPOS SESIÓN

2.1. Introducción

Los tipos sesión se han consolidado como un formalismo para el análisis modular de comunicación entre procesos. Una *sesión* es un canal privado que conecta dos procesos, cada uno con su conector o *endpoint* que habilita la comunicación estructurada mediante una especificación; el *tipo sesión*. A modo de ejemplo, el tipo sesión

$$T = \texttt{!int}.\&[\texttt{True} : \texttt{end},\ \texttt{False} : \texttt{?int}.\oplus[\texttt{True} : \texttt{end},\ \texttt{False} : T]] \tag{2.1}$$

podría describir (parte de) un protocolo que debe seguir un proceso comprador para participar en una subasta: el proceso debe enviar un valor entero que representa su oferta (`!int`) y esperar una decisión del subastador. El protocolo puede proceder en este punto en dos formas distintas correspondientes a las dos ramas del operador &. El subastador puede declarar que el artículo se vende (elección que representamos con `True`), en cuyo caso la sesión termina inmediatamente (`end`), o no aceptar la oferta (rama `False`) y enviar un contraoferta, recibida como un entero por el comprador (`?int`). En ese momento, el comprador puede elegir ($\oplus$) terminar la subasta o reiniciar el protocolo con otra oferta, aquí denotado por T. Las elecciones posibles son representadas mediante etiquetas (en este caso `True` y `False`), & describe las elecciones que puede recibir y $\oplus$ las que puede enviar.

2.2. Gramática

Formalmente la sintaxis de los tipos sesión está dada por la siguiente gramática como fue presentada en [MP17]

$$\begin{array}{rcl} t, s & ::= & \texttt{bool} \mid \texttt{int} \mid \alpha \mid T \mid [\texttt{l}_i : t_i]_{i\in I} \mid \cdots \\ T, S & ::= & \texttt{end} \mid \texttt{!}t.T \mid \texttt{?}t.T \mid \&[\texttt{l}_i : T_i]_{i\in I} \mid \oplus[\texttt{l}_i : T_i]_{i\in I} \mid A \mid \overline{A} \end{array}$$

donde t y s son utilizados para representar los tipos básicos, variables libres, tipos sesión, unión disjunta y otros. Los tipos sesión quedan descritos por T y S. Una comunicación finalizada queda descrita por el constructor de terminación (`end`). Las operaciones de escritura y lectura ($\texttt{!}t.T$ y $\texttt{?}t.T$) describen endpoints para enviar y recibir mensajes de tipo t y continuar de acuerdo con T. Las ramas ($\&[\texttt{l}_i : T_i]_{i\in I}$) describen endpoints con varias continuaciones posibles (T_i) identificadas por etiquetas ($\texttt{l}_i$), el progreso queda dado por la elección externa de una de ellas. Las elecciones ($\oplus[\texttt{l}_i : T_i]_{i\in I}$) describen endpoints que pueden seleccionar mediante una elección interna una etiqueta $\texttt{l}_i$ y continuar como T_i. Como consecuencia, la elección externa de una rama (&) queda determinada por su correspondiente elección interna ($\oplus$).

Por último, A y $\overline{A}$ representan variables libres y su dual. El *dual* de un tipo sesión T, escrito como $\overline{T}$, se obtiene intercambiando las operaciones de lectura y escritura. Esta operación nos permite obtener para cualquier endpoint, el tipo sesión de su contraparte en la comunicación que describe. La misma queda definida por las siguientes ecuaciones:

$$\begin{array}{rclcrclcrcl} \overline{\overline{A}} & = & A & \quad & \overline{(\texttt{?}t.T)} & = & \texttt{!}t.\overline{T} & \quad & \overline{\&[\texttt{l}_i : T_i]_{i\in I}} & = & \oplus[\texttt{l}_i : \overline{T_i}]_{i\in I} \\ \overline{\texttt{end}} & = & \texttt{end} & & \overline{(\texttt{!}t.T)} & = & \texttt{?}t.\overline{T} & & \overline{\oplus[\texttt{l}_i : T_i]_{i\in I}} & = & \&[\texttt{l}_i : \overline{T_i}]_{i\in I} \end{array}$$

Esta gramática permite describir también tipos sesión infinitos. Para ello pedimos se cumplan las siguientes condiciones:

Regularidad Requerimos que todo árbol consista de un número finito de subárboles *distinguibles*. Esta condición asegura que los tipos sesión sean representables mediante la notación μ [Pie02] o como soluciones a un conjunto finito de ecuaciones [Cou83].

Alcanzabilidad Requerimos que todo sub-árbol T de un tipo sesión contenga una *hoja alcanzable* etiquetada por `end`. Esta condición asegura que siempre sea posible terminar una sesión independientemente de hace cuánto esté corriendo.

2.3. Interfaz programática

```
val create  : unit → A × Ā
val close   : end → unit
val send    : α → !α.A → A
val receive : ?α.A → α × A
val select  : (Ā_k → [l_i : Ā_i]_{i∈I}) → ⊕[l_i : A_i]_{i∈I} → A_k
val branch  : &[l_i : A_i]_{i∈I} → [l_i : A_i]_{i∈I}
```

Tab. 2.1: Interfaz programática para tipos sesión

En [Pad17] se presentó el desarrollo de `FuSe`, una implementación en `OCaml` de la interfaz para la programación con sesiones en Tabla 2.1.

- `create`: crea una nueva sesión y devuelve un par de endpoints con tipos sesión duales.
- `close`: señaliza el fin de una sesión.
- `send`: envía mensaje de tipo α sobre endpoint con tipo $!\alpha.A$, retorna endpoint con tipo A para denotar el avance en la comunicación.
- `receive`: espera recibir mensaje de tipo α sobre endpoint con tipo $?\alpha.A$, retorna un par con mensaje y endpoint de tipo A.

Las primitivas `branch` y `select` permiten tratar con sesiones que presentan caminos con interacciones alternativas, cada camino siendo identificado por una *etiqueta* l_i.

- `select`: envía etiqueta l_k con $k \in I$ sobre endpoint con tipo $\oplus[l_i : A_i]_{i\in I}$, retorna un endpoint con el tipo A_k asociado a la etiqueta seleccionada. En `OCaml`, el etiquetado es representado mediante una función que *inyecta* un endpoint (ejemplo, el tipo $\overline{A_k}$) en una unión disjunta de tipo $[l_i : \overline{A_i}]_{i\in I}$ donde $k \in I$.
- `branch`: dualmente, espera recibir la comunicación de una elección en un endpoint de tipo $\&[l_i : A_i]_{i\in I}$, retorna la elección recibida como un endpoint inyectado a la unión disjunta descrita por su tipo.

2.4. Ejemplos básicos

A continuación damos algunos ejemplos sencillos de esta interfaz utilizados en [Pad17]. El siguiente código implementa el cliente de un servicio “eco” que espera un mensaje y lo devuelve al cliente.

```
let echo_client ep x =
  let ep = Session.send x ep in
  let res, ep = Session.receive ep in
  Session.close ep;
  res
```

El argumento `ep` tiene tipo $!\alpha.?\beta.\mathtt{end}$ y `x` tipo α. La función `echo_client` comienza enviando un mensaje `x` sobre el endpoint `ep`. La primitiva `let` reasocia `ep` al endpoint retornado por `send`, que ahora tiene tipo $?\beta.\mathtt{end}$.

El endpoint es después utilizado para recibir un mensaje de tipo β del servicio. Finalmente, `echo_client` cierra la sesión y devuelve el mensaje.

El servicio de eco está implementado por el `echo_service` definido a continuación, el mismo utiliza el argumento `ep` de tipo $?\alpha.!\alpha.\mathtt{end}$ para recibir un mensaje `x` y retornarlo al cliente previo al cierre de sesión.

```
let echo_service ep =
  let x, ep = Session.receive ep in
  let ep = Session.send x ep in
  Session.close ep
```

Podemos notar que $!\alpha.?\beta.\mathtt{end}$ no es el dual de $?\alpha.!\alpha.\mathtt{end}$ según la definición de dualidad presentada anteriormente: para conectar el cliente y servidor, β debe ser *unificado* con α. El código que conecta ambas partes es el siguiente:

```
let _ =
  let a, b = Session.create () in
  let _ = Thread.create echo_service a in
  print_endline (echo_client b "Hola mundo")
```

Este código crea una nueva sesión cuyos endpoints se encuentran ligados a los nombres `a` y `b`. Luego, lanza un thread que aplica `echo_service` al endpoint `a`. Finalmente, aplica `echo_client` al endpoint restante `b`.

Ahora deseamos generalizar el servicio para que un cliente pueda escoger consumirlo o finalizar la interacción sin uso. Un servicio que ofrece esta decisión se ilustra a continuación:

```
let opt_echo_service ep =
  match Session.branch ep with
  | `Msg ep → echo_service ep
  | `End ep → Session.close ep
```

En este caso, el servicio utiliza la primitiva `branch` para esperar la etiqueta seleccionada por el cliente.

El tipo inicial de `ep` es ahora $\&[\mathtt{End} : \mathtt{end}, \mathtt{Msg} : ?\alpha.!\alpha.\mathtt{end}]$ y el valor retornado por `branch` es $[\mathtt{End} : \mathtt{end}, \mathtt{Msg} : ?\alpha.!\alpha.\mathtt{end}]$. En la rama `Msg` el servicio se comporta como antes. En `End` el servicio finaliza la sesión.

La siguiente función representa un posible cliente para `opt_echo_service`:

```
let opt_echo_client ep opt x =
  if opt then
    let ep = Session.select (fun x → `Msg x) ep
    in echo_client ep x
  else
    let ep = Session.select (fun x → `End x) ep
    in Session.close ep; x
```

Esta función tiene tipo $\oplus$[End : end, Msg : !α.?α.end] $\rightarrow$ bool $\rightarrow$ α $\rightarrow$ α y su comportamiento depende del argumento booleano `opt`.

3. TIPOS SESIÓN PROBABILÍSTICOS

En [IMP+20] se propuso el uso de tipos de sesión para razonar probabilísticamente sobre propiedades de alcanzabilidad, donde se dio una interpretación *probabilística* a los operadores de selección. Más específicamente, se pasa de una interpretación no determinística a una probabilística y se estudia un sistema de tipos que permita determinar la probabilidad con la que una sesión en particular termina *exitosamente* (más precisiones sobre esta noción se darán en el Capítulo 7). Dado que no existe una interpretación universal de "terminación exitosa", se diferencia la terminación exitosa de la infructuosa por medio de un constructor dedicado.

Por ejemplo, podemos refinar (2.1) como

$$T = \texttt{!int.}\,{}_{p}\texttt{\&[True} : \texttt{done},\ \texttt{False} : \texttt{?int.}\,{}_{q}\oplus\texttt{[True} : \texttt{idle},\ \texttt{False} : T\texttt{]]} \tag{3.1}$$

donde el tipo sesión `done` indica una terminación exitosa y las ramas se anotan con probabilidades p y q. En particular, el subastador declara que el artículo se vende con probabilidad p y responde con una contraoferta con probabilidad $1 - p$, mientras que el comprador decide abandonar la subasta con probabilidad q o volver a ofertar con probabilidad $1 - q$.

3.1. Gramática

Para esta interpretación probabilística consideramos la siguiente gramática

$$\begin{array}{rcl} t, s & ::= & \texttt{bool} \mid \texttt{int} \mid \alpha \mid T \mid \langle p \rangle \mid \texttt{[True} : T,\ \texttt{False} : S\texttt{]} \mid \cdots \\ T, S & ::= & \texttt{done} \mid \texttt{idle} \mid \texttt{!}t\texttt{.}T \mid \texttt{?}t\texttt{.}T \mid \\ & & {}_{p}\texttt{\&[True} : T,\ \texttt{False} : S\texttt{]} \mid {}_{p}\oplus\texttt{[True} : T,\ \texttt{False} : S\texttt{]} \mid A \mid \overline{A} \end{array}$$

donde t y s son utilizados para representar los básicos, variables libres, sesión para endpoints, sesiones con probabilidad de éxito p y otros. Las sesiones representadas por $\langle p \rangle$ resultan de la composición o "unión" de un par de endpoints con sesión duales T y $\overline{T}$ ambos con probabilidad de éxito p. La *probabilidad de éxito* de un tipo sesión es la probabilidad con la que el protocolo descrito termina exitosamente.

A diferencia de la gramática descrita en la Sección 2.2, utilizamos dos constructores distintos para marcar el fin de una comunicación, `idle` y `done`. Utilizamos `done` para denotar los puntos de terminación exitosa de un protocolo y `idle` para los de terminación infructuosa, la semántica de terminación para ambos siendo dependiente del dominio del problema.

Los tipos sesión $\texttt{!}t\texttt{.}T$ y $\texttt{?}t\texttt{.}T$ describen endpoints para enviar y recibir mensajes al igual que en la Sección 2.2. Luego, ${}_{p}\oplus$ y ${}_{p}\&$ describen ramas y elecciones ahora anotadas con una probabilidad y limitadas a una elección binaria que es "izquierda" con probabilidad p y "derecha" con probabiliadad $1 - p$. El endpoint es luego consumido acorde T o S según corresponda. La probabilidad de ${}_{p}\&$ queda determinada por su correspondiente elección interna. Por último, A y $\overline{A}$ representan variables libres y su dual.

El *dual* de un tipo sesión probabilístico T, escrito como $\overline{T}$, extiende la definición presentada en la Sección 2.2; para los constructores de terminación es la identidad y las

operaciones de lectura y escritura son intercambiadas. Este intercambio también se realiza para las elecciones internas y externas sin afectar las probabilidades anotadas.

Por último, es necesario modificar la condición de alcanzabilidad para considerar los nuevos constructores de terminación:

Alcanzabilidad Requerimos que todo sub-árbol T de un tipo sesión contenga una *hoja alcanzable* etiquetada por `done` o `idle`.

3.2. Interfaz programática

Para la extensión a tipos sesión probabilísticos de FuSe se hicieron algunas modificaciones a la interfaz presentada en Tabla 2.1.

```
val close        : done → unit
val idle         : idle → unit
val select_true  : 1⊕[True : A, False : B] → A
val select_false : 0⊕[True : A, False : B] → B
val pick         : p → (0⊕[True : A, False : B] → α)
                     → (1⊕[True : A, False : B] → α)
                     → p⊕[True : A, False : B] → α
val branch       : p&[True : A, False : B]
                     → [True : A, False : B]
```

Tab. 3.1: Interfaz para tipos sesión probabilísticos

Comenzamos con los cambios necesarios para la distinción de una terminación exitosa, donde `close` adquiere la semántica de terminación exitosa mientras que `idle` es utilizada para denotar el fin de la comunicación.

Procedemos con las modificaciones y nuevas primitivas para la selección de caminos con cierta probabilidad. Las primitivas `select_true` y `select_false` permiten avanzar la comunicación sobre un endpoint que presenta una elección determinística por `True` o `False` respectivamente.

La función `pick` (dada una probabilidad p) escoge `True` con probabilidad p y `False` con $p - 1$ sobre el endpoint con tipo ${}_p\oplus[\texttt{True} : A,\ \texttt{False} : B]$. El comportamiento sobre la rama seleccionada queda definido por la función ${}_1\oplus[\texttt{True} : A,\ \texttt{False} : B] \to \alpha$ para el caso `True` y ${}_0\oplus[\texttt{True} : A,\ \texttt{False} : B] \to \alpha$ cuando es `False`, ambas proceden con una elección determinística sobre la rama escogida.

Por último `branch` opera sobre una elección externa ${}_p\&[\texttt{True} : A,\ \texttt{False} : B]$ y retorna ambos caminos como unión disjunta.

3.2.1. Modificación al sistema de tipos de FuSe

Los cambios presentados a la interfaz implican modificaciones al sistema de tipos utilizado por FuSe. La sintaxis y semántica es esencialmente la estándar para lenguajes de la familia ML sin ninguna extensión específica al chequeo de tipos sesión.

Describiremos los cambios necesarios haciendo referencia al trabajo de FuSe [Pad17]. Las reglas T-LEFT y T-RIGHT utilizadas para tipar la expresión correspondiente al tipo unión disjunta $[\texttt{L} : T,\ \texttt{R} : S]$ se renombran a T-TRUE-BRANCH y T-FALSE-BRANCH para coincidir con el

tipo `[True` : T, `False` : S`]`. Los cambios restantes quedan restringidos a la regla T-CONST, que describe los tipos de las primitivas de comunicación. En este caso las modificaciones son análogas a las realizadas para la interfaz programática.

3.3. Ejemplos básicos

Procedemos con la presentación de algunos ejemplos donde se aplican estas nuevas primitivas. Para ello readaptaremos el servicio de eco presentado anteriormente al modelo probabilístico.

```
let echo_service ep =
  match Session.branch ep with
  | `True ep →
      let x, ep = Session.receive ep in
      let ep = Session.send x ep in
      Session.close ep
  | `False ep → Session.idle ep
```

Comenzamos con un servicio de eco que presenta dos caminos posibles: brindar la función de eco o finalizar la comunicación. El argumento `ep` tiene tipo $_p$`&[True` : `?`α`.!`α`.done`, `False` : `idle]` donde la probabilidad p queda libre hasta que el servicio sea unido a un cliente.

Para ello, podemos implementar el siguiente cliente que selecciona determinísticamente la rama `True`.

```
let echo_client ep x =
  let ep = Session.select_true ep in
  let ep = Session.send x ep in
  let x, ep = Session.receive ep in
  Session.close ep;
  x
```

En este caso, el argumento `ep` tiene tipo $_1$`⊕[True` : `!`α`.?`β`.done`, `False` : A`]` y `x` tipo α. Podemos observar que para el tipo sesión de este endpoint la probabilidad se encuentra instanciada en 1 acorde a la selección determinística mientras que la rama `False` queda libre haste ser asociada a su par.

Otra posibilidad es un cliente que selecciona determinísticamente la rama `False`.

```
let idle_client ep =
  let ep = Session.select_false ep in
  Session.idle ep
```

Aquí el argumento `ep` tiene tipo $_0$`⊕[True` : A, `False` : `idle]`, dejando libre la rama `True`.

Por úlitmo presentamos un cliente que mediante la primitiva `pick` selecciona con probabilidad $\frac{1}{2}$ una rama o la otra mediante la composición de los clientes definidos en esta sección.

```
let coin_flip_echo_client ep x =
  Session.pick Rational.one_half
    (fun ep →
      idle_client ep;
```

```
        None)
      (fun ep → Some (echo_client ep x))
      ep
```

El argumento `ep` tiene tipo ${}_{\frac{1}{2}}\oplus$[True : !α.?β.done, False : idle]. El código que conecta este último cliente con el servicio de eco es el siguiente:

```
  let _ =
    let a, b = Session.create () in
    let _ = Thread.create echo_server a in
    coin_flip_echo_client b 42
```

Esta composición fuerza la unificación de tipos para los endpoints `a` y `b` de forma tal que el endpoint `a` asociado al servicio de eco toma el tipo ${}_{\frac{1}{2}}\&$[True : ?int.!int.done, False : idle] y su dual `b`, asociado al cliente, recibe el tipo ${}_{\frac{1}{2}}\oplus$[True : !int.?int.done, False : idle].

4. IMPLEMENTACIÓN

La implementación toma de base el trabajo realizado en `FuSe`. En esta sección presentaremos los cambios necesarios en la codificación y tipado de las primitivas para modelar elecciones probabilísticas y la información adicional necesaria para el cálculo de terminación exitosa de un tipo sesión.

4.1. Codificación

`FuSe` toma la codificación propuesta en [DGS12] posteriormente refinada [Pad17]. La idea principal es que una secuencia de mensajes sobre una sesión puede modelarse como una serie de comunicaciones mediante canales de único uso. En este modelo cada mensaje lleva un valor acompañado de un canal fresco sobre el cual continuar la comunicación.

Esta codificación depende de los siguientes tipos:

- $\mathbb{0}$ representa la ausencia de valor.
- $\langle \rho, \sigma \rangle$ describe canales que reciben mensajes del tipo ρ y envían de tipo σ. Tanto ρ como σ pueden ser instanciados con $\mathbb{0}$ para indicar que no se recibe y/o envía ningún mensaje.

De esta última representación, las siguientes instancias son utilizadas para describir las terminaciones posibles (a diferencia de `FuSe`, sumamos el símbolo $\mathbb{1}$ para distinguir una de otra):

- $\langle \mathbb{1}, \mathbb{1} \rangle$ representa la terminación exitosa.
- $\langle \mathbb{0}, \mathbb{0} \rangle$ representa la terminación no exitosa.

La relación entre los tipos sesión T y los tipos de la forma $\langle t, s \rangle$ está dada por la función $[\![\cdot]\!]$ definida debajo

Codificación de tipos sesión probabilísticos

$$
\begin{aligned}
[\![\texttt{done}]\!] &= \langle \mathbb{1}, \mathbb{1} \rangle \\
[\![\texttt{idle}]\!] &= \langle \mathbb{0}, \mathbb{0} \rangle \\
[\![?t.T]\!] &= \langle [\![t]\!] \times [\![T]\!], \mathbb{0} \rangle \\
[\![!t.T]\!] &= \langle \mathbb{0}, [\![t]\!] \times [\![\overline{T}]\!] \rangle \\
[\![{}_p\&[\texttt{True} : T,\ \texttt{False} : S]]\!] &= \langle [\texttt{True} : [\![T]\!],\ \texttt{False} : [\![S]\!]] \times p, \mathbb{0} \rangle \\
[\![{}_p\oplus[\texttt{True} : T,\ \texttt{False} : S]]\!] &= \langle \mathbb{0}, [\texttt{True} : [\![\overline{T}]\!],\ \texttt{False} : [\![\overline{S}]\!]] \times p \rangle \\
[\![A]\!] &= \langle \rho_A, \sigma_A \rangle \\
[\![\overline{A}]\!] &= \langle \sigma_A, \rho_A \rangle
\end{aligned}
$$

extendida homomórficamente a todos los tipos restantes. Asumimos que para cada variable de tipo sesión A existen dos variables de tipo σ_A y ρ_A distintas entre sí y de cualquier tipo de variable α.

Como ejemplo, el tipo sesión $?\alpha.A$ se codifica como $\langle \alpha \times \langle \rho_A, \sigma_A \rangle, \mathbb{0} \rangle$, describiendo un canal para recibir un mensaje de tipo $\alpha \times \langle \rho_A, \sigma_A \rangle$; una componente de tipo α (el valor

a comunicar) y otra de tipo $\langle \rho_A, \sigma_A \rangle$ (el canal de continuación sobre el cual avanza la comunicación).

Para la codificación de los tipos sesión que envían un valor o una elección tenemos la particularidad de que la continuación está dualizada. Esto se debe a que el tipo asociado a la continuación del canal describe el comportamiento del *receptor*. Además, este detalle permite una forma sencilla de expresar la dualidad entre tipos sesión aun cuando no se encuentran completamente instanciados.

Como ejemplo, tomemos la versión probabilística del tipo T dado en la codificación de [Pad17] $T = {}_p\oplus[\texttt{True} : !\alpha.?\beta.\texttt{done}, \texttt{False} : \texttt{idle}]$

$$\begin{aligned} [\![T]\!] &= \langle \mathbb{0}, [\texttt{True} : [\![?\alpha.!\beta.\texttt{done}]\!], \texttt{False} : [\![\texttt{idle}]\!]] \times p \rangle \\ &= \langle \mathbb{0}, [\texttt{True} : \langle \alpha \times [\![!\beta.\texttt{done}]\!], \mathbb{0} \rangle, \texttt{False} : \langle \mathbb{0}, \mathbb{0} \rangle] \times p \rangle \\ &= \langle \mathbb{0}, [\texttt{True} : \langle \alpha \times \langle \mathbb{0}, \beta \times [\![\texttt{done}]\!] \rangle, \mathbb{0} \rangle, \texttt{False} : \langle \mathbb{0}, \mathbb{0} \rangle] \times p \rangle \\ &= \langle \mathbb{0}, [\texttt{True} : \langle \alpha \times \langle \mathbb{0}, \beta \times \langle \mathbb{1}, \mathbb{1} \rangle \rangle, \mathbb{0} \rangle, \texttt{False} : \langle \mathbb{0}, \mathbb{0} \rangle] \times p \rangle \end{aligned}$$

Luego la de su dual $\overline{T} = {}_p\&[\texttt{True} : ?\alpha.!\beta.\texttt{end}, \texttt{False} : \texttt{idle}]$

$$\begin{aligned} [\![\overline{T}]\!] &= \langle [\texttt{True} : [\![?\alpha.!\beta.\texttt{done}]\!], \texttt{False} : [\![\texttt{idle}]\!]] \times p, \mathbb{0} \rangle \\ &= \langle [\texttt{True} : \langle \alpha \times [\![!\beta.\texttt{done}]\!], \mathbb{0} \rangle, \texttt{False} : \langle \mathbb{0}, \mathbb{0} \rangle] \times p, \mathbb{0} \rangle \\ &= \langle [\texttt{True} : \langle \alpha \times \langle \mathbb{0}, \beta \times [\![\texttt{done}]\!] \rangle, \mathbb{0} \rangle, \texttt{False} : \langle \mathbb{0}, \mathbb{0} \rangle] \times p, \mathbb{0} \rangle \\ &= \langle [\texttt{True} : \langle \alpha \times \langle \mathbb{0}, \beta \times \langle \mathbb{1}, \mathbb{1} \rangle \rangle, \mathbb{0} \rangle, \texttt{False} : \langle \mathbb{0}, \mathbb{0} \rangle] \times p, \mathbb{0} \rangle \end{aligned}$$

Con este último ejemplo podemos notar que la codificación de $\overline{T}$ puede obtenerse de T permutando las componentes de su par codificado. Al igual que para los tipos sesión, podemos probar también la siguiente propiedad sobre los tipos probabilísticos:

Teorema 1: Si $[\![T]\!] = \langle t, s \rangle$, entonces $[\![\overline{T}]\!] = \langle s, t \rangle$.

Demostración 1: Sigue mutatis mutandis la prueba original en [DGS12].

De forma equivalente podemos decir que si $[\![T]\!] = \langle t_1, t_2 \rangle$ y $[\![S]\!] = \langle s_1, s_2 \rangle$, entonces

$$T = \overline{S} \iff [\![T]\!] = [\![\overline{S}]\!] \iff t_1 = s_2 \wedge t_2 = s_1$$

Al igual que `FuSe`, esto permite reducir el cálculo del dual de un tipo sesión a la equivalencia entre tipos. También vale para tipos sesión desconocidos o parcialmente instanciados: $[\![A]\!] = \langle \rho_A, \sigma_A \rangle$ y $[\![\overline{A}]\!] = \langle \sigma_A, \rho_A \rangle$.

Habiendo escogido la representación para tipos sesión probabilísticos podemos observar la implementación de la interfaz `OCaml` en la Tabla 4.1. Existe una correspondencia directa entre la firma de las funciones de la Tabla 4.1 y las primitivas introducidas en Tabla 3.1. Queda en evidencia que esta codificación dificulta la lectura de los tipos sesión. Este problema se agudiza a medida que los protocolos implementados son más complejos. Por este motivo extendimos `rosetta`, una herramienta auxiliar que acompaña `FuSe`, que implementa la función inversa al encoding e imprime los tipos utilizando la notación original de tipos probabilísticos (Sección 3.1). Al momento de presentar tipos sesión inferidos por `OCaml` utilizaremos el formato generado por la herramienta para facilitar la lectura (Los detalles de `rosetta` se dan en el Capítulo 7).

```
type 𝟘
type 𝟙
type p₀ (* alias para codificación de probabilidad cero *)
type p₁ (* alias para codificación de probabilidad uno *)
type (ρ,σ) pst (* sintáxis en OCaml para ⟨ρ,σ⟩ *)
val create  : unit → (ρ,σ) pst × (σ,ρ) pst
val close   : (𝟘,𝟘) pst → unit
val idle    : (𝟙,𝟙) pst → unit
val send    : α → (𝟘,(α × (σ,ρ) pst)) pst → (ρ,σ) pst
val receive : ((α × (ρ,σ) pst),𝟘) pst → α × (ρ,σ) pst
val select_true  : (𝟘,[`True of (ρ,σ) pst |
                      `False of (γ,δ) pst] × p₁)
                   → (ρ,σ) pst
val select_false : (𝟘,[`True of (ρ,σ) pst |
                      `False of (γ,δ) pst] × p₀)
                   → (γ,δ) pst
val pick    : p →
            ((𝟘,[`True of (ρ,σ) pst | `False of (γ,δ) pst]
                 × p₀) → α) →
            ((𝟘,[`True of (ρ,σ) pst | `False of (γ,δ) pst]
                 × p₁) → α) →
            (𝟘,[`True of (ρ,σ) pst | `False of (γ,δ) pst]
                × p) → α
val branch : ([`True of (ρ,σ) pst |
                `False of (γ,δ) pst] × p,𝟘)
              → [> `True of (ρ,σ) pst | `False of (γ,δ) pst]
```

Tab. 4.1: Interfaz `OCaml` para tipos sesión probabilísticos.

4.1.1. Codificación de probabilidad

Podemos notar las elecciones llevan en su codificación una probabilidad asociada p que también está presente como argumento en la primitiva `pick`. Comenzamos definiendo p como la probabilidad con la que se envía $_p\oplus$`[True` : A, `False` : B`]` o recibe $_p\&$`[True` : A, `False` : B`]` la elección de continuar por la rama `True` en un tipo sesión.

Normalmente podría utilizarse una representación de coma flotante como `float`, el problema con tal modelo es que no brinda información del valor de p en tiempo de compilación. Disponer del valor a nivel tipo permite la definición de interfaces más fuertes como en `pick`, donde la probabilidad p que se recibe como argumento debe coincidir con la utilizada en el tipo sesión sobre el cual se opera (Tabla 3.1).

Otra ventaja es la de poder utilizar dicho valor en cualquier tipo de análisis estático, requisito para el cálculo de la probabilidad con la que una sesión termina exitosamente.

4.1.2. Representando naturales y racionales

En este trabajo decidimos representar p como un racional cuya construcción queda presente en el tipo. Para ello primero escribimos una definición de tipo para los naturales

que se basa en la construcción mediante la función sucesor.

```
type _z
type _s
type _ nat = Z : _z nat | S : α nat → (_s × α) nat
```

Tab. 4.2: Representación para naturales y el cero

Primero definimos los tipos `_z` y `_s` que serán utilizados como etiquetas para distinguir a nivel tipo entre cero y la función sucesor. Luego definimos `nat` como una estructura de datos algebraica particular denominada GADT (Generalized Algebraic Data Type) presente en `OCaml` [YM15] que permite la implementación de restricciones en los parámetros de sus constructores. Aquí definimos un constructor `Z` que actúa como cero y otro `S` que hace de función sucesor incrementando en uno el natural que toma como argumento. Luego, los naturales se definen de manera obvia como se muestra en Tabla 4.3.

```
let zero : _z nat = Z
let one : (_s × _z) nat = S Z
let two : (_s × (_s × _z)) nat = S (S Z)
```

Tab. 4.3: Construcción de naturales

Teniendo una representación para naturales, definimos `frac` como otro GADT que permite la construcción de números racionales. En este caso se utiliza una tupla donde la primer componente representa al numerador y la segunda el denominador, prohibiendo el cero mediante una restricción de tipo. De esta forma, los racionales se definen como a continuación:

```
type _ frac = Fraction : α nat × (_s × β) nat →
                         (α nat × (_s × β) nat) frac
```

Tab. 4.4: Representación de racionales

A modo de ejemplo damos la construcción para $\frac{1}{2}$.

```
let one_half : ((_s × _z) nat *
                (_s × (_s × _z)) nat) frac
                = Fraction (S Z, S (S Z))
```

Tab. 4.5: Ejemplo de racionales

5. ELECCIONES MULTI-SESIÓN

Hasta ahora estudiamos programas que trabajan con una única sesión. La introducción de múltiples sesiones requiere de algunas consideraciones respecto cómo influyen las probabilidades de unas sobre otras. En esta sección veremos cómo tales interacciones afectan nuestras primitivas y su tipado.

5.1. Limitaciones con `pick` de una sesión

La primitiva de elección probabilística `pick` opera sobre una única sesión, tomando como argumento la misma y el comportamiento sobre cada rama posible. Podría suceder que luego de una elección se interactúe con otra sesión, a continuación estudiaremos en detalle qué ocurre en tal escenario.

Comenzamos con un programa donde ambas ramas de la elección sobre `epX` presentan una comunicación con otra sesión `epY`:

```
let two_sessions_pick_both_branches epX epY =
  Session.pick Rational.one_half
    (fun epX →
      let epX = Session.select_false epX in
      Session.idle epX;
      let epY = Session.send false epY in
      Session.close epY)
    (fun epX →
      let epX = Session.select_true epX in
      Session.close epX;
      let epY = Session.send true epY in
      Session.close epY)
    epX
```

Aquí la sesión `epX` tiene tipo $_{\frac{1}{2}}\oplus$`[True : done, False : idle]` mientras que `epY` lleva el tipo `!bool.done`. Por más que el *valor* comunicado por `epY` dependa de la elección en `epX`, el programa tipa correctamente. Esto sucede porque el tipo de la sesión es idéntico en ambas ramas.

Veamos qué ocurre cuando difiere el valor del tipo a transmitir sobre `epY` según la elección:

```
let two_sessions_invalid_pick epX epY =
  Session.pick Rational.one_half
    (fun epX →
      let epX = Session.select_false epX in
      Session.idle epX;
      let epY = Session.send false epY in
      Session.close epY)
    (fun epX →
      let epX = Session.select_true epX in
      Session.close epX;
```

```
        let epY = Session.send 42 epY in (* error de tipado *)
        Session.close epY)
      epX
```

En este último caso, el sistema de tipo nos indicará que `epY` presenta dos tipos incompatibles entre sí: `!bool.done` y `!int.done`, por lo tanto no tipa. El comportamiento de `epY` difiere según la elección realizada por `epX` y esta dependencia no es capturada por el sistema de tipos.

Para poder describir estas relaciones es necesario introducir algunos aspectos del sistema de tipos que se enfocan en el resultado de combinar sesiones en elecciones.

5.2. Combinador de tipo probabilístico

Como vimos en el último ejemplo surge la necesidad de capturar el tipo de una sesión que cambia de acuerdo con una elección. Para esto nos basamos en el *combinador de tipo probabilístico* presentado en [IMP+20], que permite combinar tipos ponderando las distintas formas en las cuales progresa una sesión bajo una probabilidad determinada.

Definición 1 (combinador de tipo probabilístico): Escribimos $t\ {}_{p}\boxplus\ s$ para la combinación convexa de los tipos t y s, que se encuentra definida por casos sobre la forma de t y s:

$$t\ {}_{p}\boxplus\ s \stackrel{\text{def}}{=} \begin{cases} t & \text{si } t = s \\ {}_{pq+(1-p)r}\oplus[\texttt{True} : T,\ \texttt{False} : S] & \text{si } t = {}_{q}\oplus[\texttt{True} : T,\ \texttt{False} : S] \\ & \text{y } s = {}_{r}\oplus[\texttt{True} : T,\ \texttt{False} : S] \\ \langle pq + (1-p)r \rangle & \text{si } t = \langle q \rangle \text{ y } s = \langle r \rangle \\ \text{indefinido} & \text{caso contrario} \end{cases}$$

Intuitivamente, $t\ {}_{p}\boxplus\ s$ describe un recurso que es utilizado acorde a t con probabilidad p y como s con probabilidad $1-p$. La combinación de t y s se encuentra definida únicamente cuando t y s tienen formas “compatibles”, el caso trivial siendo cuando poseen mismo tipo. Los casos interesantes se dan cuando t y s describen una elección o una sesión cerrada (Capítulo 6). Para el primero, la combinación resulta en una nueva elección que lleva como probabilidad la suma convexa de sus partes. Al combinar sesiones cerradas, se toma la suma convexa sobre la probabilidad de éxito de las sesiones involucradas.

5.3. Reglas de tipado para elecciones

A continuación presentamos las reglas de tipado para elecciones del cálculo de tipos sesión probabilísticos [IMP+20] y cómo se emplea el combinador.

Utilizaremos contextos de tipado para el seguimiento de variables presentes en procesos. Un *contexto de tipado* es un mapa de variables a tipos escrito como $x_1 : t_1, \ldots, x_n : t_n$. Vamos a definir Γ y Δ para representar estos contextos, escribiendo $\emptyset$ para un contexto vacío, $\mathsf{dom}(\Gamma)$ para el dominio de Γ y Γ, Δ como la unión de los contextos Γ y Δ cuando $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. Además, extendemos ${}_{p}\boxplus$ punto a punto sobre contextos.

En Tabla 5.1 adaptamos ligeramente las reglas T-CHOICE y T-BRANCH en [IMP+20] con el equivalente a la sintaxis de nuestra extensión; las primitivas `branch` y `pick`. La regla T-BRANCH tipa un proceso que recibe una elección mediante la sesión x y continua como T con probabilidad p, caso contrario S. El tipo de x debe ser de la forma ${}_{p}\&[\texttt{True} : T,\ \texttt{False} :$

T-BRANCH

$$\frac{\Gamma, x : T \vdash P : \sigma \qquad \Delta, x : S \vdash Q : \sigma}{\Gamma\ {}_p\boxplus\ \Delta, x : {}_p\&[\texttt{True} : T,\ \texttt{False} : S] \vdash \texttt{match}\ (\texttt{branch}\ x)\ \texttt{with}\ \texttt{True} \to Q \mid \texttt{False} \to P : \sigma}$$

T-CHOICE

$$\frac{\emptyset \vdash prob : p \qquad \Gamma \vdash P : \sigma \qquad \Delta \vdash Q : \sigma}{\Gamma\ {}_p\boxplus\ \Delta \vdash \texttt{pick}\ prob\ P\ Q : \sigma}$$

Tab. 5.1: Reglas de tipado.

S], el resto de los recursos son lo que nos compete en esta sección, ya que el proceso deberá comportarse según Γ en caso de ir por `True` o Δ si es `False`. De esta forma el comportamiento del proceso queda descrito por $\Gamma\ {}_p\boxplus\ \Delta$, bajo la definición del combinador esto implica que futuras elecciones en Γ y Δ serán ajustadas como efecto de la elección recibida en x. La regla T-CHOICE permite tipar elecciones probabilísticas combinando la evolución de ambos contextos mediante ${}_p\boxplus$.

Retomando nuestro ejemplo anterior, podemos observar que el tipo de `epY` en cada rama de la elección no era compatible bajo la definición del combinador. Veamos qué sucede al utilizar tipos compatibles:

```
let two_sessions_invalid_pick epX epY =
  Session.pick Rational.one_half
    (fun epX →
      let epX = Session.select_false epX in
      Session.idle epX;
      let epY = Session.select_false epY in
      let epY = Session.send false epY in
      Session.close epY)
    (fun epX →
      let epX = Session.select_true epX in
      Session.close epX;
      let epY = Session.select_true epY in (* error de tipado *)
      let epY = Session.send 42 epY in
      Session.close epY)
    epX
```

Ahora `epY` tiene tipo ${}_0\oplus$`[True` : T, `False` : `!bool.done]` en la rama `False` y ${}_1\oplus$`[True` : `!int.done`, `False` : S`]` cuando se escoge `True`. Estos tipos son compatibles para la combinación, sin embargo el programa no tipa ya que el motor de inferencia no dispone de la información necesaria para unirlos. En la Tabla 5.1 el combinador probabilístico se aplica punto a punto sobre todas los recursos presentes en ambos contextos. Nuestra primitiva `pick` está limitada a combinar los tipos de `epX`.

5.4. Extensión para elecciones multi-sesión

Habiendo presentado las limitaciones del `pick` actual y cómo estas son resueltas en el cálculo probabilístico contamos con el contexto suficiente para introducir nuestra solución. Luego de las últimas modificaciones a nuestro programa, nos falta brindar al motor de inferencia la información necesaria para poder combinar elecciones de más de un tipo sesión.

Idealmente, nos gustaría que la primitiva `pick` permitiera la combinación de cualquier número de sesiones; un desafío que presenta tal objetivo es el de preservar las restricciones de tipo necesarias para evitar combinaciones inválidas y poder calcular la probabilidad de éxito de las sesiones involucradas. Considerando además la potencial complejidad de esta hipotética interfaz, decidimos en cambio agregar primitivas específicas al número de sesiones a combinar.

Para ello extendemos nuestro conjunto de primitivas comenzando con la introducción de la primitiva `pick_2ch` y su tipo en Tabla 5.2. Esta primitiva permite combinar las elecciones de exactamente dos sesiones. Dada una probabilidad p, escoge `True` con probabilidad p y `False` con $p-1$ sobre el endpoint con tipo ${}_{p}\oplus[\texttt{True} : A,\ \texttt{False} : B]$. Además, cada rama presenta una segunda elección cuya combinación se ve reflejada en el endpoint con tipo ${}_{pq+(1-p)r}\oplus[\texttt{True} : C,\ \texttt{False} : D]$. Al igual que con `pick` el comportamiento de la rama seleccionada queda definido por las funciones sobre los tipos sesión correspondientes a cada elección. Por ejemplo, al escoger `True` se llama a la función que toma la sesión con tipo ${}_{1}\oplus[\texttt{True} : A,\ \texttt{False} : B]$ y la sesión secundaria a combinar con probabilidad ${}_{q}\oplus[\texttt{True} : C,\ \texttt{False} : D]$.

$$
\begin{aligned}
&\texttt{val pick_2ch}:\\
&\quad p \to ({}_{0}\oplus[\texttt{True} : A,\ \texttt{False} : B] \to {}_{r}\oplus[\texttt{True} : C,\ \texttt{False} : D] \to \alpha)\\
&\quad\ \to ({}_{1}\oplus[\texttt{True} : A,\ \texttt{False} : B] \to {}_{q}\oplus[\texttt{True} : C,\ \texttt{False} : D] \to \alpha)\\
&\quad\ \to {}_{p}\oplus[\texttt{True} : A,\ \texttt{False} : B] \to {}_{pq+(1-p)r}\oplus[\texttt{True} : C,\ \texttt{False} : D]\\
&\quad\ \to \alpha
\end{aligned}
$$

Tab. 5.2: Interfaz de primitiva `pick_2ch`

Veamos cómo queda nuestro ejemplo original utilizando esta nueva primitiva:

```
let two_sessions_valid_pick2 epX epY =
  Session.pick_2ch Rational.one_half
    (fun epX epY →
      let epX = Session.select_false epX in
      Session.idle epX;
      let epY = Session.select_false epY in
      let epY = Session.send false epY in
      Session.close epY)
    (fun epX epY →
      let epX = Session.select_true epX in
      Session.close epX;
      let epY = Session.select_true epY in
      let epY = Session.send 42 epY in
      Session.close epY)
```

```
epX epY
```

Ya no tenemos más problemas con el tipado de `epY`, en las funciones que definen el comportamiento de cada rama la sesión lleva su tipo como argumento ($_0\oplus$`[True : !int.done, False : !bool.done]` y $_1\oplus$`[True : !int.done, False : !bool.done]`). A su vez, la combinación de ambos tipos queda reflejada en el tipo $_{\frac{1}{2}}\oplus$`[True : !int.done, False : !bool.done]` simulando así la aplicación del combinador probabilístico.

Esta solución puede extenderse a más de dos sesiones siguiendo la misma idea de materializar la combinación probabilística en el tipo de la primitiva.

6. COMPOSICIÓN DE TIPOS SESIÓN DUALES

Nos resta introducir un tipo más a nuestra extensión, la composición de tipos sesión duales representada como $\langle p \rangle$. La misma encapsula la unión de los endpoints de tipo T y $\overline{T}$ ambos con probabilidad de éxito p.

Este tipo se presenta al momento de crear una sesión y su función es puramente informativa, guarda la probabilidad de éxito de la sesión. En nuestra extensión queremos capturar esta información ya que representa uno de los objetivos del cálculo probabilístico, obtener la probabilidad de éxito de una sesión mediante la información de tipo de sus primitivas.

Elegir una representación adecuada para este tipo presenta algunos desafíos. La misma debe capturar la información de tipo de sus endpoints (T o su dual $\overline{T}$) para permitir el cálculo de éxito y a su vez está sujeta a la combinación probabilística vista en Definición 1.

```
type (ρ,σ) cpst (* sintáxis en OCaml para sesión cerrada
                   con endpoint de tipo ⟨ρ,σ⟩ *)
val create   : ?st:(ρ, σ) cpst → unit → (ρ,σ) pst × (σ,ρ) pst
val cst_placeholder : (α, β) cpst
```

Tab. 6.1: Interfaz `OCaml` para tipos sesión probabilísticos.

Nuestra representación toma la forma de un argumento opcional en la primitiva `create`. Dado que la función de este tipo es informativa, el uso de un argumento opcional permite ignorarlo y aún así disponer de las garantías de tipo que provee nuestra extensión. Podemos observar que el tipo unifica con uno de los endpoints de la sesión, $[\![T]\!] = \langle \rho, \sigma \rangle$. También podría haber tomado su dual, $[\![\overline{T}]\!] = \langle \sigma, \rho \rangle$, ya que el propósito es disponer del tipo de la sesión completa para calcular su probabilidad de éxito.

Dado que sólo nos interesa el tipo final de nuestro argumento opcional, el valor a utilizar es indistinto; para ello tenemos la constante `cst_placeholder` que tipa como una sesión cerrada pero su implementación es un valor de tipo `unit`.

6.1. Ejemplos de uso

Veamos qué ocurre al obtener la composición de tipos para los ejemplos vistos en la introducción:

```
let run_echo_client_example ?(st = cst_placeholder) () =
  let ep1, ep2 = Session.create ~st () in
  let _ = Thread.create echo_service ep1 in
  echo_client ep2 42
```

En este programa, primero creamos los endpoints utilizando `st` como argumento para obtener así la información de tipo de la sesión. Luego cada endpdoint es consumido acorde al comportamiento de la sesión, permitiendo así inferir su tipo. Habiendo unificado nuestro argumento opcional `st` con el tipo de la sesión, nuestra herramienta traductora `rosetta`

es capaz de calcular la probabilidad de éxito de la misma. Para este ejemplo, nuestro programa `run_echo_client_example` llevaría el siguiente tipo:

```
val run_echo_client_example : ?st:⟨1⟩ → unit → int
```

La probabilidad de éxito es 1 lo cual no sorprende ya que este ejemplo es completamente determinístico. Contemplemos ahora un escenario con algún tipo de elección probabilística:

```
let run_coin_flip_echo_client_example
                                    ?(st = cst_placeholder) () =
  let ep1, ep2 = Session.create ~st () in
  let _ = Thread.create echo_server ep1 in
  coin_flip_echo_client ep2 42
```

El único cambio con respecto al ejemplo anterior es el uso de `coin_flip_echo_client` que envía con probabilidad $\frac{1}{2}$, caso contrario finaliza con `idle`.

```
val run_coin_flip_echo_client_example : ?st:⟨0,5⟩ → unit → int
```

En este caso, la probabilidad de éxito es de $\frac{1}{2}$ lo cual coincide con el comportamiento de su sesión.

6.2. Interacción con sesiones multi-sesión

Si recordamos la definición del combinador probabilístico, esta incluía la combinación de sesiones cerradas. En particular tenemos que $\langle q \rangle \ {}_p\boxplus \ \langle r \rangle = \langle pq{+}(1{-}p)r \rangle$. A diferencia de la combinación de endpoint donde el tipo combinado lleva información del comportamiento de la sesión, acá se trata nuevamente de un dato informativo; la probabilidad de éxito de una sesión que puede tomar distinta forma según una elección externa.

Veamos un ejemplo de tal combinación:

```
let pick_idle_close_and_run_echo_client
                                    ?(st = cst_placeholder) () ep =
  Session.pick Rational.one_half
    (fun ep →
      let ep = Session.select_false ep in
      Session.idle ep;
      run_echo_client_example ~st ())
    (fun ep →
      let ep = Session.select_true ep in
      Session.close ep;
      run_echo_client_example ~st ())
```

En este caso, `ep` tiene tipo ${}_p\oplus$`[True : done, False : idle]`; esta elección afecta el tipo de `st` ya que vemos está presente en ambas ramas. La sesión cerrada `st` tiene tipo $\langle 1 \rangle$ en cada rama (se trata de una sesión sin elecciones probabilísticas), la combinación no afecta su tipo final.

Para el siguiente ejemplo, `st` captura la probabilidad de éxito de dos sesiones distintas en cada rama:

```
let pick_idle_close_and_run_echo_client_or_coin_flip
                                    ?(st = cst_placeholder) () ep =
  Session.pick Rational.one_half
    (fun ep →
      let ep = Session.select_false ep in
      Session.idle ep;
      run_echo_client_example ~st ())
    (fun ep →
      let ep = Session.select_true ep in
      Session.close ep;
      match run_coin_flip_echo_client_example ~st () with
      (* error de tipado *)
      | Some result → result
      | None → 0)
    ep
```

En este caso podemos observar se obtiene un error de tipado ya que al escoger `True`, `st` toma el tipo $\langle 0, 5\rangle$ mientras que en la rama `False` este es de tipo $\langle 1\rangle$.

Dado que se trata de un tipo informativo, cabe resaltar este ejemplo tipa si quitamos el argumento opcional:

```
let pick_idle_close_and_mix_run_echo_client ep =
  Session.pick Rational.one_half
    (fun ep →
      let ep = Session.select_false ep in
      Session.idle ep;
      run_echo_client_example ())
    (fun ep →
      let ep = Session.select_true ep in
      Session.close ep;
      match run_coin_flip_echo_client_example () with
      | Some result → result
      | None → 0)
    ep
```

El error surge al buscar la aplicación del combinador probabilístico sobre los tipos de las sesiones finalizadas en la firma de nuestro programa.

Nos encontramos con la misma limitación del `pick` solo que aplicado al tipo de sesiones concluidas. Siguiendo la misma motivación que con `pick_2ch`, decidimos agregar una primitiva que capture en su tipo esta nueva combinación. Presentamos a continuación la primitiva `pick_2st` y su tipo en Tabla 6.2. Así como `pick_2ch` permite combinar las elecciones de exactamente dos sesiones, aquí podemos combinar exactamente un endpoint y una sesión cerrada.

El uso de la primitiva es análogo al de `pick_2ch` con la diferencia de que el segundo argumento de la función de cada rama así como el tipo combinado final es sobre sesiones combinadas, no elecciones.

```
let pick_idle_close_and_run_echo_client_or_coin_flip
                                    ?(st = cst_placeholder) () ep =
  Session.pick_2st Rational.one_half
```

```
val pick_2st :
    p → (₀⊕[True : A, False : B] → ⟨r⟩ → α)
      → (₁⊕[True : A, False : B] → ⟨q⟩ → α)
      → ₚ⊕[True : A, False : B] → ⟨pq + (1 − p)r⟩ → α
```

Tab. 6.2: Interfaz de primitiva `pick_2st`

```
(fun ep st →
  let ep = Session.select_false ep in
  Session.idle ep;
  run_echo_client_example ~st ())
(fun ep st →
  let ep = Session.select_true ep in
  Session.close ep;
  match run_coin_flip_echo_client_example ~st () with
  | Some result → result
  | None → 0)
ep st
```

De esta manera, `st` tipa correctamente en cada rama y su información de tipo permite el cálculo de su combinación ponderada:

```
val pick_idle_close_and_run_echo_client_or_coin_flip :
        ?st:⟨0,75⟩ → unit → ½⊕[True : done, False : idle] → int
```

7. PROBABILIDAD DE ÉXITO DE UNA SESIÓN

En secciones anteriores hablamos de la probabilidad de éxito pero aún no dimos una definición formal para la misma. Intuitivamente, la probablidad se computa considerando todos los caminos en la estructura de T que llevan a un estado `done`. A continuación, resumimos la definición dada en [IMP+20]:

Definición 2 (Probabilidad de éxito): La *probabilidad de éxito* de un tipo sesión T, denotada como $\langle\langle T\rangle\rangle$, se encuentra definida por las siguientes ecuaciones:

$$\begin{array}{lll} \langle\langle \texttt{idle}\rangle\rangle = 0 & \langle\langle \texttt{?}t\texttt{.}T\rangle\rangle = \langle\langle T\rangle\rangle & \langle\langle {}_p\texttt{\&[True}:T,\ \texttt{False}:S\texttt{]}\rangle\rangle = p\langle\langle T\rangle\rangle + (1-p)\langle\langle S\rangle\rangle \\ \langle\langle \texttt{done}\rangle\rangle = 1 & \langle\langle \texttt{!}t\texttt{.}T\rangle\rangle = \langle\langle T\rangle\rangle & \langle\langle {}_p\oplus\texttt{[True}:T,\ \texttt{False}:S\texttt{]}\rangle\rangle = p\langle\langle T\rangle\rangle + (1-p)\langle\langle S\rangle\rangle \end{array}$$

Para un tipo sesión T *finito*, la Definición 2 describe un algoritmo recursivo para el cómputo de $\langle\langle T\rangle\rangle$. Cuando T es infinito (un protocolo recursivo) deja de ser tan obvio. Para obtener un algoritmo que funcione en el caso general, interpretamos $\langle\langle T\rangle\rangle$ como una *variable aleatoria*. La Definición 2 nos permite derivar un *sistema finito de ecuaciones* relacionando tales variables. El lado derecho de cada ecuación para $\langle\langle T\rangle\rangle$ está expresada en términos de variables aleatorias que corresponden a los nodos hijos en el árbol descrito por T. Como T tiene una cantidad finita de sub-árboles, obtenemos un número finito de ecuaciones.

Luego, observamos que cada tipo sesión T corresponde a una Cadena de Markov en Tiempo Discreto (Discrete-Time Markov Chain, DTMC) [KS76] cuyo espacio de estados es $\mathcal{T}(T) = \{S_1, \ldots, S_n\}$ y tal que la probabilidad p_{ij} de transicionar de un estado S_i al estado S_j está dada por

$$p_{ij} \stackrel{\text{def}}{=} \begin{cases} p & \text{si } S_i \rightsquigarrow_p S_j \\ 0 & \text{caso contrario} \end{cases}$$

La regularidad y alcanzabilidad implican que la DTMC obtenida por el tipo sesión T tiene un número de estados finito y es absorbente. Esto significa siempre es posible alcanzar un *estado absorbente* (`done` ó `idle`) desde cualquier *estado transitorio* (cualquier otro tipo sesión). En cualquier DTMC absorbente con un número de estados finito, la probabilidad de alcanzar un estado absorbente desde uno transitorio puede ser computado mediante la resolución de un sistema de ecuaciones para el cual se garantiza una solución única [KS76]. La solución que se obtiene para $\langle\langle T\rangle\rangle$ utilizando la Definición 2 es la que determina la probabilidad de llegar a un estado `done` desde T.

7.1. Extendiendo decodificación para tipos sesión probabilísticos

Cuando hablamos sobre la codificación de los tipos sesión, mencionamos brevemente el uso de `rosetta` para decodificar los tipos generados por `OCaml`. Esta herramienta utiliza la gramática de tipos de `OCaml` para parsear las interfaces y aplicar la función inversa $[\![\cdot]\!]^{-1}$ a los tipos sesión codificados. El resultado es un árbol de sintaxis abstracta que puede ser utilizado para procesamientos o simplemente ser impreso bajo la sintaxis que uno desee.

La extensión al cálculo de tipos sesión probabilísticos consistió de dos partes: modificar la función inversa para considerar tipos probabilísticos y constructores de terminación exitosa, y el cálculo de probabilidad de éxito para tipos sesión cerrados $\langle p \rangle$.

Lo primero consistió en modificar la decodificación de forma tal que se contemplen las anotaciones de probabilidad en el árbol sintáctico final. Esto requirió cambios únicamente en el modelado de tipos para recibir o enviar elecciones. Las probabilidades en sí, que como vimos en la sección de codificación representan su valor en el tipo, también debieron ser consideradas al momento de representarlas en el árbol sintáctico. Ya que se asume las expresiones a decodificar tipan en `OCaml`, se transforman para conservar únicamente su valor en un tipo `int` o `float` según el caso. Por último, se creó la distinción entre los nodos `done` y `idle`, indispensable para el cálculo de probabilidad de éxito.

Lo segundo fue agregar el procesamiento necesario para computar la probabilidad de éxito en los tipos sesión cerrados. Para ello, primero tomamos el árbol sintáctico correspondiente al endpoint de tipo T (el cual describe la sesión cerrada) y lo transformamos a una matriz de adyacencia con las probabilidades de transición p_{ij} descritas anteriormente.

Esta matriz es dividida en submatrices que corresponden a las transiciones entre estados transitorios y las que van de estados transitorios a absorbentes. Con esta información es posible computar la matriz de absorción que contiene la probabilidad con la que cada estado es absorbido a `done` o `idle`.

7.2. Calculando probabilidad de absorción

Vamos a seguir el ejemplo de subasta presentado en [IMP+20] para ver cómo se decodifica su tipo y cómo se computa su probabilidad de éxito al combinarlo con su dual.

Presentamos un programa que actúa como comprador y su dual, vendedor.

```
let rec buyer ep offer =
  let ep = Session.send offer ep in
  match Session.branch ep with
  | `True ep →
      Session.close ep;
      offer
  | `False ep →
      let counteroffer, ep = Session.receive ep in
      Session.pick Rational.two_thirds
        (fun ep →
          let ep = Session.select_false ep in
          buyer ep counteroffer)
        (fun ep →
          let ep = Session.select_true ep in
          Session.idle ep;
          -1)
        ep
```

El comprador envía una oferta y espera la decisión del vendedor. En caso de aceptarse la oferta, la sesión termina exitosamente. Si esta es rechazada; el comprador recibe una contra-oferta y debe decidir si aceptarla o no. Si la rechaza, el protocolo termina sin éxito; caso contrario se llama recursivamente para repetir la secuencia.

Mediante `rosetta` obtenemos el siguiente tipo para `ep`:

$$rec\ X.!\texttt{int}.\ {}_{p}\&[\texttt{True}:\texttt{done},\ \texttt{False}:?\texttt{int}.\ {}_{\frac{2}{3}}\oplus[\texttt{True}:\texttt{idle},\ \texttt{False}:X]]$$

Una diferencia con respecto a los ejemplos anteriores es que al presentar una recursión, la sesión hace referencia así misma mediante la variable de tipo X.

A continuación presentamos el programa vendedor:

```
let rec seller ep =
  let bid, ep = Session.receive ep in
  Session.pick Rational.one_quarter
    (fun ep →
      let ep = Session.select_false ep in
      let ep = Session.send (bid + 10) ep in
      match Session.branch ep with
      | `True ep → Session.idle ep
      | `False ep → seller ep)
    (fun ep →
      let ep = Session.select_true ep in
      Session.close ep)
    ep
```

El tipo decodificado de `ep`:

$$rec\ X.?\texttt{int}.\ {}_{\frac{1}{4}}\oplus[\texttt{True}:\texttt{done},\ \texttt{False}:!\texttt{int}.\ {}_{q}\&[\texttt{True}:\texttt{idle},\ \texttt{False}:X]]$$

Además de ser el dual del endpoint utilizado por el comprador, podemos observar tiene instanciada la probabilidad de elección que requiere su contraparte.

Finalmente, unimos ambos endpoints en un programa que ejecuta la subasta:

```
let test_buyer_seller ?(st = cst_placeholder) () =
  let ep1, ep2 = Session.create ~st () in
  let _ = Thread.create seller ep1 in
  buyer ep2 42
```

Al unificar, el tipo de cada endpoint posee las probabilidades concretas para cada elección; como ejemplo tomamos el tipo unificado para `ep2` (el endpoint comprador): $rec\ X.!\texttt{int}.\ {}_{\frac{1}{4}}\&[\texttt{True}:\texttt{done},\ \texttt{False}:?\texttt{int}.\ {}_{\frac{2}{3}}\oplus[\texttt{True}:\texttt{idle},\ \texttt{False}:X]]$.

Lo primero que hace `rosetta` es construir el árbol sintáctico para la expresión de nuestro tipo sesión. A partir de este se generan los siguientes estados correspondientes a cada término alcanzable:

$$s_0 = !\texttt{int}.\ {}_{\frac{1}{4}}\&[\texttt{True}:\texttt{done},\ \texttt{False}:?\texttt{int}.\ {}_{\frac{2}{3}}\oplus[\texttt{True}:\texttt{idle},\ \texttt{False}:X]]$$

$$s_1 = {}_{\frac{1}{4}}\&[\texttt{True}:\texttt{done},\ \texttt{False}:?\texttt{int}.\ {}_{\frac{2}{3}}\oplus[\texttt{True}:\texttt{idle},\ \texttt{False}:X]]$$

$$s_2 = \texttt{done}$$

$$s_3 = ?\texttt{int}.\ {}_{\frac{2}{3}}\oplus[\texttt{True}:\texttt{idle},\ \texttt{False}:X]$$

$$s_4 = {}_{\frac{2}{3}}\oplus[\texttt{True}:\texttt{idle},\ \texttt{False}:X]$$

$$s_5 = \texttt{idle}$$

Agregando las transiciones entre estados, determinadas por la probabilidad de transición entre los mismos, podemos construir la DTMC presente en la Figura 7.2. Para proceder con nuestro algoritmo, obtenemos su representación como matriz de adyacencia, P.

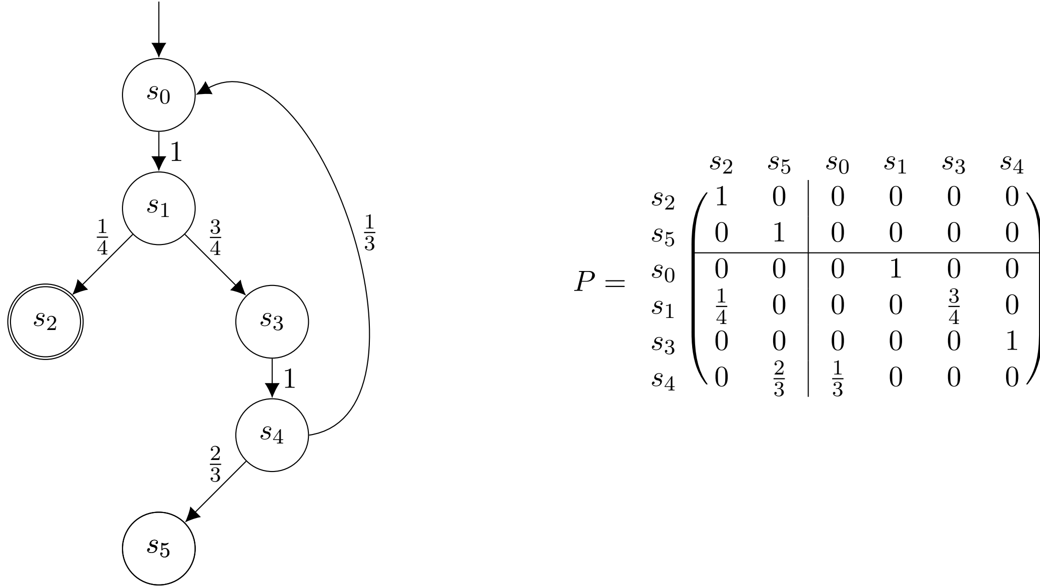

Fig. 7.1: DTMC para T y su matriz de adyacencia P

Vamos a definir las siguientes sub-matrices de P para nuestro cálculo de absorción:

- Q es la matriz de 4×4 con las probabilidades de transición entre estados transitorios.
- R es la matriz de 4×2 con las probabilidades de transición de estados transitorios a estados absorbentes.

Nos resta calcular la matriz absorbente B, para lo cual aplicaremos el teorema 3.3.7 de [KS76]. El mismo enuncia que si b_{ij} es la probabilidad de que un proceso que comienza en un estado transitorio s_i termine en uno abosrbente s_j, entonces:

$$\{b_{ij}\} = B = (I - Q)^{-1} R$$

Comenzando de un estado s_i, podría llegarse a un estado absorbente s_j en uno o más pasos. La probabilidad de llegar en un paso está dada por p_{ij}. Si esto no ocurre, el proceso puede caer en otro estado absorbente o un estado transitorio s_k. En el último caso, existe una probabilidad b_{kj} de ser capturado por el estado absorbente. Por lo tanto tenemos:

$$b_{ij} = p_{ij} + \sum_{s_k \in S} p_{ik} b_{kj}$$

Que en forma matricial puede ser escrito como:

$$\begin{aligned} B &= R + QB \\ B - QB &= R + QB - QB \\ (I - Q)B &= R \\ (I - Q)^{-1}(I - Q)B &= (I - Q)^{-1} R \\ B &= (I - Q)^{-1} R \end{aligned}$$

De esta manera, la probabilidad de que nuestro estado $s_0 = T$ sea absorbido por $s_2 =$ `done` (equivalente a $\langle\langle T \rangle\rangle$) se obtiene calculando:

$$
\begin{aligned}
B &= (I - Q)^{-1} R \\
&= \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & -\frac{3}{4} & 0 \\ 0 & 0 & 1 & -1 \\ -\frac{1}{3} & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 0 \\ \frac{1}{4} & 0 \\ 0 & 0 \\ 0 & \frac{2}{3} \end{bmatrix} = \begin{bmatrix} \frac{4}{3} & \frac{4}{3} & 1 & 1 \\ \frac{1}{3} & \frac{4}{3} & 1 & 1 \\ \frac{4}{9} & \frac{4}{9} & \frac{4}{3} & \frac{4}{3} \\ \frac{4}{9} & \frac{4}{9} & \frac{1}{3} & \frac{4}{3} \end{bmatrix} \begin{bmatrix} 0 & 0 \\ \frac{1}{4} & 0 \\ 0 & 0 \\ 0 & \frac{2}{3} \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ \frac{1}{3} & \frac{2}{3} \\ \frac{1}{9} & \frac{8}{9} \\ \frac{1}{9} & \frac{8}{9} \end{bmatrix}
\end{aligned}
$$

La probabilidad de absorción para s_0 es b_{00}. Por lo tanto $\langle\langle T \rangle\rangle = \frac{1}{3}$. Con este resultado, **`rosetta`** carga la probabilidad de éxito en el árbol sintáctico y este es impreso como:

```
val test_buyer_seller : ?st:⟨0,33⟩ → unit → int
```

8. CONCLUSIÓN

Consideramos que con el auge en el desarrollo de técnicas de descripción de interfaces y soporte a nivel de lenguajes de programación que intenten asegurar la corrección por construcción, esta clase de trabajos enfocados en el desarrollo de interfaces extensibles e implementaciones sencillas pueden contribuir a la popularización y uso de tipos comportamentales como los tipos sesión.

Inspirados por el trabajo de `FuSe` y su codificación de tipos sesión, hemos presentado una extensión que implementa el cálculo de tipos sesión probabilísticos sin perder las garantías del sistema original. Nuestra implementación permite la representación e inferencia de tipos sesión probabilísticos con una interfaz programática enfocada en la usabilidad del sistema. La misma no requiere de funcionalidades avanzadas del lenguaje de implementación como lo pueden ser el uso de mónadas o sistemas de tipo sub-estructurales. El uso de mónadas [FMT18] [PT08] como sesiones de linealidad es más fuerte que nuestra validación mediante tipos y tiempo de ejecución heredada de `FuSe`, sin embargo, consideramos la primera sacrifica expresividad y/o usabilidad del sistema.

Existen varias áreas donde podría continuar esta rama de trabajo. Una de ellas es la implementación de distribuciones de probabilidad para tratar elecciones con más de una rama. La representación de naturales y fracciones como tipos no restringe al uso de valores mayores a 1, esto puede validarse en tiempo de ejecución, al momento de utilizar la herramienta de decodificación o con alguna mejora a la representación actual que sea habilitada por el sistema de tipos. Al momento de calcular la probabilidad de éxito de una sesión, si esta no tiene instanciadas todas las probabilidades involucradas, el decodificador de tipos podría utilizar alguna representación simbólica para variables libres y retornar un resultado en función de las mismas.

Bibliografía

[COE+20] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Trans. Program. Lang. Syst.*, 42(3), November 2020.

[Cou83] Bruno Courcelle. Fundamental properties of infinite trees. In *Theor. Comput. Sci.*, 1983.

[CY20] David Castro-Perez and Nobuko Yoshida. CAMP: cost-aware multiparty session protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):155:1–155:30, 2020.

[DBH+21] A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. Resource-aware session types for digital contracts. In *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 111–126, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.

[DBHP19] Ankush Das, Stephanie Balzer, Jan Hoffmann, and Frank Pfenning. Resource-aware session types for digital contracts. *CoRR*, abs/1902.06056, 2019.

[DGS12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM, 2012.

[DHP18] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 305–314, New York, NY, USA, 2018. Association for Computing Machinery.

[DP20] Ankush Das and Frank Pfenning. Session types with arithmetic refinements. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[DWH20] Ankush Das, Di Wang, and Jan Hoffmann. Probabilistic resource-aware session types, 2020.

[FMT18] Adrian Francalanza, Claudio Antares Mezzina, and Emilio Tuosto. Reversible choreographies via monitoring in erlang. In *Distributed Applications and Interoperable Systems - 18th IFIP WG 6.1 International Conference, DAIS 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings*, pages 75–92, 2018.

[Fow16] Simon Fowler. An erlang implementation of multiparty session actors. *arXiv preprint arXiv:1608.03321*, 2016.

[Fow20] Simon Fowler. Model-view-update-communicate: Session types meet the elm architecture. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[GHK+20] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight go. *Proc. ACM Program. Lang.*, 4(OOPSLA):149:1–149:29, 2020.

[GPP+21] Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021.

[HBK20] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):6:1–6:30, 2020.

[HLV+16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, volume 715, pages 509–523. Springer, 1993.

[Hor20] Ross Horne. Session subtyping and multiparty compatibility using circular sequents. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 12:1–12:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[IMP+20] Omar Inverso, Hernán C. Melgratti, Luca Padovani, Catia Trubiani, and Emilio Tuosto. Probabilistic analysis of binary sessions. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 14:1–14:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[INYY20] Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[JY20] Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In Peter Müller, editor,

Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, volume 12075 of *Lecture Notes in Computer Science*, pages 251–279. Springer, 2020.

[Kok19] Wen Kokke. Rusty variation: Deadlock-free sessions with failure in rust. In Massimo Bartoletti, Ludovic Henrio, Anastasia Mavridou, and Alceste Scalas, editors, *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, volume 304 of *EPTCS*, pages 48–60, 2019.

[KS76] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Springer-Verlag, 1976.

[LM16] Sam Lindley and J. Garrett Morris. Embedding session types in haskell. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 133–145. ACM, 2016.

[LNTY18] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018.

[LNY20] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Implementing multiparty session types in rust. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 127–136. Springer, 2020.

[MFYZ21] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in typescript with routed multiparty session types. In Aaron Smith, Delphine Demange, and Rajiv Gupta, editors, *CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*, pages 94–106. ACM, 2021.

[MP17] Hernán C. Melgratti and Luca Padovani. An ocaml implementation of binary sessions. 2017.

[MYZ20] Rupak Majumdar, Nobuko Yoshida, and Damien Zufferey. Multiparty motion coordination: from choreographies to robotics programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):134:1–134:30, 2020.

[OY17] Dominic Orchard and Nobuko Yoshida. Session types with linearity in haskell. *Behavioural Types: from Theory to Tools*, page 219, 2017.

[Pad17] Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.

[Pie02] Benjamin C. Pierce. *Types and programming languages.* "MIT Press", 2002.

[PT08] Ricardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *HASKELL'08*, pages 25–36. ACM, 2008.

[SYB19] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 502–516. ACM, 2019.

[VCAM20] Vasco T. Vasconcelos, Filipe Casal, Bernardo Almeida, and Andreia Mordido. Mixed sessions. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 715–742. Springer, 2020.

[YM15] Dr Jeremy Yallop and Dr Anil Madhavapeddy. Programming with gadts. `https://www.cl.cam.ac.uk/teaching/1415/L28/gadts.pdf`, 2015.