Universidad ORT Uruguay
Facultad de Ingeniería

Inferencia de Tipos de Sesión

Entregado como requisito para la obtención del título de
Master en Ingeniería

Ernesto Copello - 156597

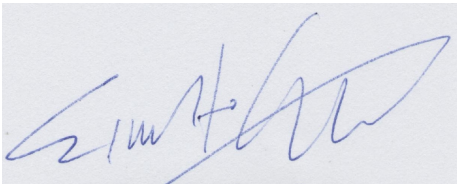
Tutores:
Nora Szasz
Álvaro Tasistro

2012

Declaración de Autoría

Yo, Ernesto Copello, declaro que el trabajo que se presenta en esa obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba mi tesis de Master;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mí;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Ernesto Copello
2 de Marzo del 2012

Agradecimientos

Deseo agradecer a mi padre que me orientó hacia la ingeniería, y a mi madre por motivar continuamente mis estudios, a ambos por brindarme su apoyo incondicional.

Agradezco a Nora Szasz y Álvaro Tasistro la oportunidad y la confianza depositada en mí al brindarme la posibilidad de realizar este trabajo, y por sobre todo la paternal amistad que me brindan. También deseo reconocer que sus enseñanzas, contribuciones y correcciones fueron indispensables para este trabajo.

Finalmente agradezco a la ANII (Asociación Nacional de Investigación e Innovación) por la beca otorgada para la realización de esta Maestría.

Resumen

El cálculo de Tipos de Sesión tal como es introducido en [1] estructura los programas en términos de procesos concurrentes que se comunican estableciendo diálogos. Estos diálogos pueden ser descriptos en forma abstracta mediante secuencias de tipos de mensajes, cada uno de los cuales describe el formato y dirección del mensaje. El sistema resultante impone una disciplina que garantiza la compatibilidad de los diálogos entre procesos de un programa bien tipado. El sistema es polimórfico *à la Curry* [2], pero ningún tratamiento formal de este aspecto ni algoritmo completo de inferencia ha sido publicado aún. En este trabajo presentamos una versión extendida con esquemas de tipos de un fragmento no trivial del cálculo de Tipos de Sesión y un algoritmo que infiere el esquema de tipos principal (*Principal Type Scheme*) para cualquier programa tipable, junto con la prueba de solidez y completitud del mismo. Asimismo presentamos una formalización completa en la Teoría Constructiva de Tipos del cálculo de Tipos de Sesión, y del algoritmo de inferencia y su demostración de solidez, utilizando para esto el asistente de demostraciones Agda.

Palabras Clave

Tipos de Sesión, inferencia de tipos, meta-teoría formal de lenguajes de programación, concurrencia

Índice general

Capítulo 1

Introducción

En sistemas distribuidos es común que la comunicación sincrónica punto a punto por un *canal* entre dos procesos consista en un diálogo estructurado descrito por cierto *protocolo*, el cual especifica el formato, dirección y secuenciación de los mensajes.

En [1] se introdujeron los Tipos de Sesión (*Session Types*), con el fin de controlar estáticamente que la comunicación siga un protocolo preestablecido. Los Tipos de Sesión imponen una disciplina que garantiza la compatibilidad en los patrones de interacción entre los procesos de un programa bien tipado. Los errores capturados por este sistema van desde desacuerdos en el formato o tipo de un mensaje entre el que envía y el que recibe, interferencia de procesos externos en la comunicación punto a punto entre dos procesos, colisiones de mensajes si ambas puntas envían información al mismo tiempo sobre un mismo canal, hasta algunos tipos de *deadlocks* cuando ambos procesos quedan esperando simultáneamente por información sobre un mismo canal.

1.1. Tipos de Sesión

Introducimos los Tipos de Sesión mediante un ejemplo clásico de la literatura [3, 4]. Definimos a continuación el protocolo que describe una interacción simplificada entre un usuario y un cajero automático. Denotamos mediante letras *cursivas* a los datos que viajan, mientras que utilizamos **negrita** para las opciones y procesos, utilizando la primer letra en mayúscula en estos últimos.

1. El **Cliente** comunica su *id* al cajero, donde el *id* es un número natural.
2. El **Cajero** contesta si se tiene **fallo** o **éxito**.
3. Si la respuesta anterior es **fallo** entonces la interacción termina. Pero si en cambio es **éxito** el **Cliente** responde con una de las siguientes dos secuencias de interacciones:
   a) El **Cliente** inicia un **depósito**.
   b) El **Cliente** comunica la *cantidad* a ser depositada, donde *cantidad* es un número natural.
   c) El **Cajero** devuelve el *saldo*, donde *saldo* es un número natural.

O, alternativamente:

a) El **Cliente** inicia un **retiro**.

b) El **Cliente** comunica la *cantidad* a ser retirada.

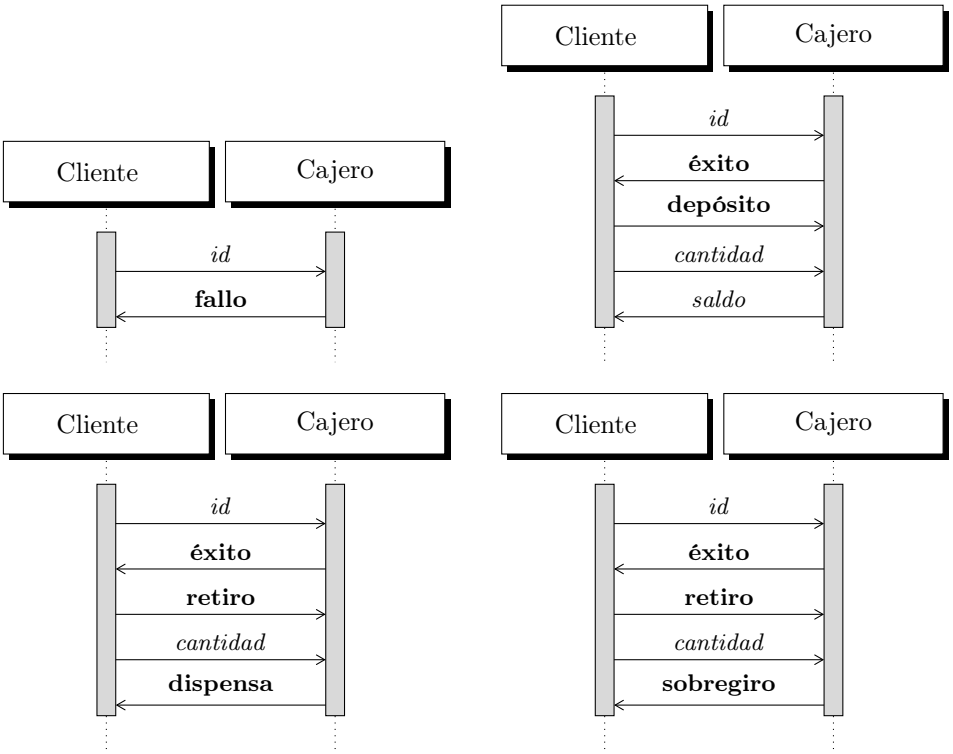
c) Dependiendo de si el saldo actual es mayor o no a la cantidad deseada, el **Cajero** responde con alguna de las siguientes interacciones:

- El **Cajero dispensa** la cantidad pedida.

O, alternativamente:

- El **Cajero** informa un **sobregiro**.

Las siguientes imágenes resumen los escenarios posibles en el protocolo descrito anteriormente:



Desde el punto de vista del **Cliente** el tipo de sesión que modela este protocolo es el siguiente:

$$
\begin{array}{lll}
![\texttt{nat}]; \ \&\{\textbf{fallo}: & \texttt{end} & \\
\quad ,\textbf{éxito}: & \oplus\{\textbf{depósito}: & ![\texttt{nat}]; ?[\texttt{nat}]; \texttt{end} \\
& \quad ,\textbf{retiro}: & ![\texttt{nat}]; \&\{\textbf{dispensa}: \texttt{end} \\
& & \qquad ,\textbf{sobregiro}: \texttt{end}\}\}
\end{array}
\tag{1.1}
$$

En este tipo de sesión se utilizan los operadores ! y ? para indicar el envío y recepción de información de cierto tipo respectivamente. Los operadores & y $\oplus$ indican la posibilidad de aceptar o seleccionar una opción respectivamente.

Así, mediante la expresión ![nat], el tipo comienza describiendo que el **Cliente** envía un natural. Luego con el operador & se especifica que espera una de las opciones: **fallo** o **éxito**. En caso que la opción recibida sea **fallo**, mediante el tipo `end` se especifica que no es posible ya comunicación alguna. En caso que la opción recibida sea **éxito**, mediante $\oplus$ describe que puede seleccionar arbitrariamente la opción **depósito** o **retiro**. En caso de decidir la opción **depósito** se comporta de la siguiente forma: primero envía un natural ![nat], luego espera un natural ?[nat], para finalmente dar por terminada la comunicación, lo cual se representa mediante el tipo `end`. En caso que decida la opción de **retiro**, manda un natural ![nat], y luego mediante & especifica que espera una de las siguientes opciones: **dispensa** o **sobregiro**. En cualquiera de estos dos casos la comunicación termina.

Este mismo protocolo puede ahora ser visto desde el punto de vista del **Cajero**.

$$\begin{array}{lllll} ?[\texttt{nat}]; & \oplus\{\textbf{fallo}: & \texttt{end} & & \\ & \quad,\textbf{éxito}: & \&\{\textbf{depósito}: & ?[\texttt{nat}];![\texttt{nat}];\texttt{end} & \\ & & \quad,\textbf{retiro}: & ?[\texttt{nat}];\oplus\{\textbf{dispensa}:\texttt{end} & \\ & & & \quad,\textbf{sobregiro}:\texttt{end}\}\} & \end{array} \tag{1.2}$$

Observar al principio que cuando el **Cliente** envía un natural el **Cajero** debe ahora estar listo para recibirlo, lo cual se especifica mediante la expresión ?[nat]. Los operadores ? y ! son *duales* o *complementarios* en este sentido ya que para establecer una comunicación entre dos procesos uno debe recibir y el otro enviar información del mismo tipo. Análogamente sucede con los operadores $\oplus$ y &: si un proceso tiene la posibilidad de seleccionar entre varias opciones, especificado mediante $\oplus$, el otro proceso debe ser capaz de aceptar cualquiera de ellas, lo cual es indicado con &, y viceversa.

Esta dualidad se propaga a través de todo el tipo de sesión, dando lugar a la noción de *tipo dual* o *tipo complementario*. Si llamamos α al tipo descrito en 1.1 entonces denotamos por $\overline{\alpha}$ al tipo especificado en 1.2. Más adelante en la sección 2.6 presentamos la definición inductiva del sistema de tipos, y en 2.5 definimos la función que determina el tipo complementario de un tipo de sesión dado.

Mostramos ahora un posible *proceso* para un **Cliente**, donde la variable de canal x tiene el tipo de sesión α descrito anteriormente. En realidad esta variable de canal x representa un extremo del canal, justamente el correspondiente al **Cliente**.

$$\begin{array}{lllll} \textbf{Cliente} = \texttt{accept}\ a(x)\ \texttt{in} & (x![id]; & & & \\ & x \triangleright \{\textbf{fallo} & : \texttt{inact} & & \\ & \quad,\textbf{éxito} & : \texttt{if}\ \ldots & \texttt{then} & x \triangleleft \textbf{depósito}; \\ & & & & x![cantidad]; \\ & & & & x?(saldo)\ \texttt{in inact} \\ & & & \texttt{else} & x \triangleleft \textbf{retiro}; \\ & & & & x![cantidad]; \\ & & & & x \triangleright \{\textbf{dispensa}:\texttt{inact} \\ & & & & \quad,\textbf{sobregiro}:\texttt{inact}\}) \end{array} \tag{1.3}$$

Un posible proceso para un **Cajero** es el siguiente, donde la variable de canal y tiene el tipo de sesión $\overline{\alpha}$.

$$
\begin{array}{rl}
\textbf{Cajero} = & \texttt{request}\ a(y)\ \texttt{in} \\
& (y?(id)\ \texttt{in} \\
& \quad (\texttt{if}\ldots\ \ \texttt{then}\ \ y \triangleleft \textbf{fallo}; \texttt{inact} \\
& \qquad\qquad \texttt{else}\ \ y \triangleleft \textbf{éxito}; \\
& \qquad\qquad\quad y \triangleright \{\textbf{depósito}: \ y?(cantidad)\ \texttt{in} \\
& \qquad\qquad\qquad\qquad\quad (\ldots; \\
& \qquad\qquad\qquad\qquad\quad y![saldo]; \\
& \qquad\qquad\qquad\qquad\quad \texttt{inact}) \\
& \qquad\qquad\quad , \textbf{retiro}: \ \ y?(cantidad)\ \texttt{in} \\
& \qquad\qquad\qquad\qquad\quad (\texttt{if}\ldots\texttt{then}\ y \triangleleft \textbf{dispensa}; \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{inact} \\
& \qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ y \triangleleft \textbf{sobregiro}; \\
& \qquad\qquad\qquad\qquad\qquad\quad \texttt{inact})\}))
\end{array}
\tag{1.4}
$$

El nombre a es un nombre de puerto, y las primitivas `accept` y `request` acuerdan un canal de comunicación sobre este nombre de puerto. Los nombres de variables x e y representan cada extremo de este canal de comunicación creado mediante las primitivas anteriores. La operación ! permite enviar información por un canal, mientras que la operación ? se utiliza para recibir información. Notar que esta sintaxis es la misma que la usada en los Tipos de Sesión. Para el envío y recepción de información (datos) hay variables de tipo `nat` como *id*, *cantidad* y *saldo*. Por otro lado, para seleccionar una opción sobre un canal se utiliza el operador $\triangleleft$, y para aceptar opciones por un canal se utiliza $\triangleright$. La primitiva `inact` determina el fin del proceso. Se dispone también de saltos condicionales a través de la primitiva `if ... then ... else`. En la literatura se los denomina también *ramas internas*, dado que son bifurcaciones debidas a condiciones internas del proceso, en contraposición a la selección de opciones que brindan *ramas externas*, donde la selección de la rama que será ejecutada depende de procesos externos.

Presentamos la sintaxis completa del cálculo de procesos en la sección 2.5. Esta sintaxis facilita la identificación de las variables ligadas; estas son todas aquellas que aparecen dentro de paréntesis curvos y luego seguidas del operador `in`. Siguiendo esta idea vemos en los anteriores ejemplos cómo los canales son ligados por los operadores `accept` y `request`, y las variables de datos aparecen ligadas por el operador ?.

Finalmente, utilizando el operador | podemos ahora componer el proceso **Cliente** y el proceso **Cajero**, para representar así el proceso en el cual estos dos procesos interactúan en paralelo:

Cliente | **Cajero**

Como vimos anteriormente los tipos de sesión de ambos procesos son complementarios. Veremos en el próximo capítulo que por esa razón esta composición paralela estará bien tipada en el sistema.

Existen varias presentaciones de los Tipos de Sesión [5, 6, 7, 8, 1, 4, 3, 9, 10, 11]. En varias de ellas ha habido antecedentes de propiedades erróneamente atribuidas al sistema, con lo cual la formalización de la teoría se hace interesante desde el punto de vista de la confiabilidad de los resultados. El principal objetivo de este trabajo es realizar un experimento inicial con Tipos de Sesión, haciendo uso de la Teoría Constructiva de Tipos como sistema lógico de la formalización. Queremos ver propiedades del sistema de Tipos de Sesión mediante una rigurosa formalización de este sistema de forma de verificar su validez; y en particular, mostraremos cómo esta formalización puede ser utilizada para elaborar un algoritmo de inferencia de Tipos de Sesión y demostrar su corrección. La inferencia de Tipos de Sesión está comenzando a ser activamente estudiada [12, 13, 14, 15, 16, 17], pero por el momento no conocemos trabajos que formulen algoritmos completos de inferencia de Tipos de Sesión ni mucho menos que formalicen su corrección en un asistente de pruebas.

Nótese que el lenguaje de procesos no tiene ninguna clase de anotaciones de tipado. En realidad el sistema de Tipos de Sesión que presentaremos es polimórfico *à la Curry* [2], esto es, existen procesos que admiten varias asignaciones de tipos bajo las reglas de tipado que presentaremos en 2.8. Por ejemplo, en el proceso $x?(y)$ `in` $(x![y].\texttt{end})$ la variable y puede tener cualquier tipo. Esto motiva la introducción de *variables de tipo* que permitan modelar estos tipos polimórficos, pasando así a tener *esquemas de tipos* en vez de tipos. El algoritmo que infiere los tipos para un proceso debe en realidad hallar el esquema de tipos principal (*Principal Type Scheme*) para cualquier programa tipable, esto es, encontrar el esquema de tipos "más general" que tipe un proceso dado. En la sección 3 presentamos un algoritmo de inferencia sólido y completo para un fragmento significativo del cálculo de Tipos de Sesión y demostramos su corrección. También conjeturamos que el algoritmo se puede extender al resto del cálculo. Y en la sección 7 presentamos la formalización de la demostración de solidez de este algoritmo en Teoría Constructiva de Tipos utilizando el asistente de demostraciones Agda.

1.2. Teoría Constructiva de Tipos

En este trabajo utilizamos la Teoría Constructiva de Tipos para realizar una formalización de los Tipos de Sesión. La Teoría Constructiva de Tipos es un sistema fundacional para la matemática constructiva desarrollada por el lógico sueco Per Martin-Löf. Siguiendo el isomorfismo de Curry-Howard [18], un teorema es representado mediante un tipo de datos y una prueba de este teorema es un objeto de ese tipo. Así, una prueba de un teorema es en general una función que, dadas pruebas de las hipótesis, computa una prueba de la tesis del teorema. Este sistema puede también ser usado como un lenguaje de programación, donde las especificaciones son representadas como tipos de datos y los programas como objetos de esos tipos. Gracias a que el control de tipos es decidible tenemos una forma automática de chequear la corrección de las demostraciones y de los programas. Aprovechamos esta característica para desarrollar un algoritmo de inferencia de Tipos de Sesión, para luego, utilizando el poder expresivo de la Teoría Constructiva de Tipos, razonar acerca de propiedades de este algoritmo. Esta ventaja de la Teoría Constructiva de Tipos sobre otros lenguajes estándares

de programación ha sido objeto de diversos estudios [19, 20].

Para codificar nuestro trabajo elegimos Agda [21, 22, 23]. Este es un lenguaje de programación con tipos funcionales dependientes, y un asistente de pruebas a su vez, basado en la Teoría Constructiva de Tipos descrita anteriormente. Tiene familias inductivas, esto es, tipos de datos que dependen de valores. Posee también módulos parametrizables. Este lenguaje tiene similitudes con otros asistentes de pruebas basados en tipos dependientes, como Alf, Coq, Epigram, Matita y NuPRL.

1.3. Descripción del Desarrollo del Trabajo

Este trabajo está organizado de la siguiente forma:

En el capítulo 2, damos una introducción más profunda a los Tipos de Sesión. Retomamos el ejemplo presentado en la sección anterior, introduciendo el sistema de tipos, y cómo aplica a este ejemplo. Agregamos a este ejemplo el concepto de *delegación* en la comunicación, y las reglas de tipado que introduce este nuevo concepto. Luego definimos formalmente el cálculo de procesos de los Tipos de Sesión y presentamos el lema de *weakening* del juicio de tipado de los Tipos de Sesión. Finalmente damos un último ejemplo que utilizamos como caso de prueba en el resto del trabajo.

En el capítulo 3, presentamos el algoritmo de inferencia y demostramos su corrección. Comenzamos por estudiar algunos trabajos relacionados, para luego introducir el fragmento del cálculo de procesos sobre el que definimos el algoritmo de inferencia. Este es presentado como un sistema de reglas dirigido por la sintaxis de los procesos. Finalmente demostramos (en lenguaje natural) los resultados de solidez y completitud para el algoritmo propuesto con respecto al juicio de tipado.

En el capítulo 4, brindamos una breve introducción a la Teoría Constructiva de Tipos y el asistente de pruebas Agda. Presentamos algunos constructores de tipos, junto con alguno de sus operadores y propiedades básicas. También damos una introducción a la inducción y recursión bien fundada, y algunos predicados de accesibilidad propios de Agda.

En el capítulo 5, damos una formalización en Agda del fragmento considerado del cálculo de procesos de Tipos de Sesión y su sistema de tipos tal cual es introducido en el capítulo 3. Finalmente presentamos algunas propiedades formalmente demostradas del sistema de tipos formalizado.

En el capítulo 6, presentamos la codificación en Agda de los algoritmos de unificación para esquemas de tipos. Utilizando predicados de accesibilidad particulares demostramos que estos algoritmos siempre terminan. Para uno de los algoritmos demostramos que el unificador encontrado es el más general, y damos algunas propiedades de la unificación y sustituciones halladas por estos algoritmos.

En el capítulo 7, presentamos la codificación en Agda del algoritmo de inferencia propuesto, y damos la demostración formal del resultado de solidez de este algoritmo.

En el capítulo 8, damos algunas conclusiones y trabajo a futuro.

En el apéndice A, damos una demostración informal y detallada del resultado de weakening introducido en el capítulo 2.

En el apéndice B, brindamos más detalles de la codificación en Agda de los algoritmos de unificación. Damos más propiedades de la unificación y sustituciones. Finalmente presentamos la demostración de la parada de uno de los algoritmos de unificación

Finalmente el apéndice C, contiene la comprobación de tipado completa de un proceso recursivo que calcula el factorial de un número.

Capítulo 2

Tipos de Sesión

El cálculo π [24], CSP (Communicating Sequential Processes [25]) y CCS (Calculus of Communicating Systems [26]) son los principales cálculos de procesos distribuidos, y han sido utilizados como formalismos para investigar sistemas de tipos para lenguajes de programación concurrentes [27, 9, 28, 10, 24, 11]. En estos sistemas el chequeo de tipos de un programa garantiza que ciertas formas de error no ocurran en tiempo de ejecución. Los errores capturados van desde desacuerdos en el formato o tipo de un mensaje entre el que envía y el que recibe, hasta algunos tipos de *deadlocks*.

Los cálculos citados anteriormente están basados en la comunicación punto a punto entre *extremos de canales*, o de aquí en más simplemente *canales*, conocidos mediante un cierto *nombre*. Se suele denominar como *puerto* al propio canal de comunicación visto como un todo, conformado entonces por sus dos extremos. Estos puertos son también identificados mediante cierto nombre. Estos cálculos nos permiten referirnos a un canal mediante una variable de canal, tal y como los lenguajes de programación clásicos nos permiten usar variables con valores enteros además de las propias constantes enteras. No sucede lo mismo para los puertos para los cuales no existen variables de puertos. Más específicamente, los nombres de canales modelan los números de puertos TCP/IP utilizados hoy en día para la comunicación entre computadoras. A su vez las variables de canal pueden ser vistas como variables que almacenan estos números de puerto. Mientras que se podría interpretar el concepto de puertos como una comunicación TCP/IP punto a punto ya instanciada, esto es, con dos números de puertos TCP/IP ya instanciados para cada uno de sus extremos.

La idea central de muchos de los sistemas de tipos desarrollados sobre estos cálculos es que cada uno de estos canales tenga un tipo asociado, el cual describe uniformemente el tipo de los mensajes que viajan por él; por ejemplo `![int]` podría ser el tipo que describe un canal por el cual sólo se envían números enteros.

En sistemas distribuidos es común que la comunicación entre dos procesos consista en un diálogo estructurado descrito por un *protocolo*, el cual especifica secuencialmente el formato y dirección de los mensajes. Esta forma de comunicación estructurada no encaja bien en sistemas de tipos que requieran que cada canal de comunicación tenga mensajes de sólo un formato o tipo. Para tratar este problema Honda *et al.* en [1] introdujeron los *Tipos de Sesión* permitiendo así especificar secuencias estructuradas de mensajes no uniformes en su tipo.

Así por ejemplo, `?[int];![bool];end` es un tipo que describe a un canal que primero recibe un natural, luego envía un booleano y finalmente no puede volver a ser utilizado.

Mediante este sistema de tipos se asegura una *comunicación segura* y *confiabilidad en las sesiones*. Con comunicación segura nos referimos a la extensión a secuencias de comunicaciones de la propiedad de corrección estándar de tipos simples para el cálculo π, esto es, que en un proceso bien tipado bajo este sistema de tipos sólo sea intercambiada información del tipo esperado. Por otro lado la confiabilidad en las sesiones es una característica típica de los Tipos de Sesión, donde nombres especiales, llamados *canales de sesión*, pueden transportar mensajes de distintos tipos pero en una secuencia específica.

En las siguientes secciones retomamos el ejemplo presentado en la introducción [3, 4] para así ir definiendo el sistema de tipos, y las reglas básicas de tipado de los Tipos de Sesión.

2.1. Reglas básicas del Sistema de Tipos de Sesión

En el ejemplo dado en la sección 1.1 introdujimos los elementos básicos del cálculo de Tipos de Sesión. En esta sección daremos las reglas del sistema de tipos para estos elementos. En este cálculo de procesos existen distintos tipos de nombres o variables: de datos (tipos básicos), de canales de sesión, de puertos y finalmente de procesos.

El sistema de Tipos de Sesión asigna tipos básicos S a las variables de tipos de datos, tipos de sesión o de canales α a los canales de sesión, tipos de la forma $\langle \alpha, \overline{\alpha} \rangle$ a los puertos de un proceso P, y tipos de la forma $\tilde{S}\tilde{\alpha}$ a las variables de procesos, mediante el sistema de reglas de asignación de tipos. Los juicios de tipado son de la forma:

$$\Theta; \Gamma \vdash P \triangleright \Delta$$

Este juicio se lee: si los puertos y las variables de datos usados en P tienen los tipos especificados en Γ y los procesos usados en P tienen los tipos dados en Θ, entonces los canales usados en P se comportan de acuerdo a los tipos especificados en Δ.

La expresión $\Delta \cdot x{:}\alpha$ denota la extensión del contexto Δ con la asignación del tipo α a la variable x. Esta expresión exige que $x \notin dom(\Delta)$, esto es, que x no esté declarada en Δ. Esta notación se extiende también a los contextos de procesos, y de puertos y variables.

Las reglas de tipado para la inicialización de sesiones, llamadas [Acc] y [Req], establecen que las variables de canales x ligadas por las primitivas `accept` y `request` sobre un puerto común a, tienen respectivamente los tipos de sesión α y $\overline{\alpha}$ especificados por el tipo de este puerto en el contexto Γ.

$$\frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot x{:}\alpha}{\Theta; \Gamma \vdash \texttt{accept}\ a(x)\ \texttt{in}\ P \triangleright \Delta}\ [\text{Acc}]$$

$$\frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot x{:}\overline{\alpha}}{\Theta; \Gamma \vdash \texttt{request}\ a(x)\ \texttt{in}\ P \triangleright \Delta}\ [\text{Req}]$$

Para ilustrar el uso de estas reglas, volvemos al ejemplo de la sección 1.1. Si abreviamos mediante α al Tipos de Sesión definido en 1.1, llamamos **SubCliente** al proceso resultante de quitar la primitiva `accept` del proceso **Cliente** definido en 1.3, y llamamos **SubCajero** al proceso resultante de quitar la primitiva `request` del proceso **Cajero** presentado en 1.4, por lo explicado informalmente hasta ahora tenemos que se satisfacen los siguientes juicios:

$$\emptyset; \{a{:}\langle \alpha, \overline{\alpha}\rangle\} \vdash \mathbf{SubCliente} \triangleright \{x{:}\alpha\}$$

$$\emptyset; \{a{:}\langle \alpha, \overline{\alpha}\rangle\} \vdash \mathbf{SubCajero} \triangleright \{y{:}\overline{\alpha}\}$$

Aplicando las reglas [Acc] y [Req] a partir de estos juicios se puede verificar que el puerto a tiene el tipo $\langle \alpha, \overline{\alpha}\rangle$ en los procesos **Cliente** y **Cajero**.

$$\frac{\{a{:}\langle \alpha, \overline{\alpha}\rangle\} \vdash a \ \triangleright \langle \alpha, \overline{\alpha}\rangle \qquad \emptyset; \{a{:}\langle \alpha, \overline{\alpha}\rangle\} \vdash \mathbf{SubCliente} \triangleright \{x{:}\alpha\}}{\emptyset; \{a{:}\langle \alpha, \overline{\alpha}\rangle\} \vdash \underbrace{\texttt{accept}\ a(x)\ \texttt{in}\ \mathbf{SubCliente}}_{\mathbf{Cliente}} \triangleright \emptyset}\ [\textsc{Acc}]$$

$$\frac{\{a{:}\langle \alpha, \overline{\alpha}\rangle\} \vdash a \ \triangleright \langle \alpha, \overline{\alpha}\rangle \qquad \emptyset; \{a{:}\langle \alpha, \overline{\alpha}\rangle\} \vdash \mathbf{SubCajero} \triangleright \{y{:}\overline{\alpha}\}}{\emptyset; \{a{:}\langle \alpha, \overline{\alpha}\rangle\} \vdash \underbrace{\texttt{request}\ a(y)\ \texttt{in}\ \mathbf{SubCajero}}_{\mathbf{Cajero}} \triangleright \emptyset}\ [\textsc{Req}]$$

Las reglas [Send] y [Rcv] sirven para ir formando los Tipos de Sesión, tal como se observa en las siguientes reglas.

$$\frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot k{:}\alpha}{\Theta; \Gamma \vdash k!{[\tilde{e}]}; P \ \triangleright \Delta \cdot k{:}!{[\tilde{S}]}; \alpha}\ [\textsc{Send}]$$

$$\frac{\Theta; \Gamma \cdot \tilde{x}{:}\tilde{S} \vdash P \triangleright \Delta \cdot k{:}\alpha}{\Theta; \Gamma \vdash k?(\tilde{x})\ \texttt{in}\ P \ \triangleright \Delta \cdot k{:}?{[\tilde{S}]}; \alpha}\ [\textsc{Rcv}]$$

Si en un proceso P el tipo α especifica el protocolo de comunicación del canal k y el vector de tipos básicos $\tilde{S}$ especifica los tipos del vector de expresiones $\tilde{e}$, entonces la regla [Send] define que k tiene que tener el tipo $![\tilde{S}]; \alpha$ en el proceso $k![\tilde{e}]; P$. La regla complementaria [Rcv] es análoga a la anterior, sólo que, como el operador ? liga las variables que aparecen en el vector $\tilde{x}$ dentro del proceso P, estas variables no tienen alcance fuera de este proceso. Debido a esto la regla [Rcv] quita la información de tipo de las variables $\tilde{x}$ del contexto que tipa el proceso $k?(\tilde{x})$ `in` P.

Por otra parte las reglas [Br] y [Sel] muestran cómo los tipos de sesión α son construidos a partir de la información brindada por los operadores de selección de opciones. Si $\forall i \in 1, \ldots, n$ el tipo de sesión α_i especifica el protocolo de comunicación del canal de sesión k en el proceso P_i, entonces la regla [Br] establece que el tipo $\&\{l_1{:}\alpha_1 \ldots l_n{:}\alpha_n\}$ determina el comportamiento del canal k en el proceso $k \triangleright \{l_1{:}P_1 \ldots l_n{:}P_n\}$. La regla complementaria [Sel] determina que si en un proceso P una variable de canal k se comporta según la especificación dada por el tipo α_j, entonces este canal tiene que tener un tipo $\oplus\{l_1{:}\alpha_1, \ldots, l_j{:}\alpha_j, \ldots, l_n{:}\alpha_n\}$, donde justamente se establece que la etiqueta u opción l_j tiene necesariamente asociado el tipo α_j, en el proceso $k \triangleleft l_j; P$

$$\frac{\Theta;\Gamma \vdash P_1 \triangleright \Delta \cdot k{:}\alpha_1 \qquad \ldots \qquad \Theta;\Gamma \vdash P_n \triangleright \Delta \cdot k{:}\alpha_n}{\Theta;\Gamma \vdash k \triangleright \{l_1{:}P_1 \ldots l_n{:}P_n\} \triangleright \Delta \cdot k{:}\&\{l_1{:}\alpha_1 \ldots l_n{:}\alpha_n\}}\ [\textsc{Br}]$$

$$\frac{\Theta;\Gamma \vdash P \triangleright \Delta \cdot k{:}\alpha_j}{\Theta;\Gamma \vdash k \triangleleft l_j; P \triangleright \Delta \cdot k{:} \oplus \{l_1{:}\alpha_1 \ldots l_n{:}\alpha_n\}}\ (1 \leq j \leq n)[\textsc{Sel}]$$

La regla base para la construcción de los tipos de sesión está dada por la regla [Inact], la cual determina que todos los tipos de sesión que aparezcan en el contexto Δ tienen que tener tipo `end` (que será a su vez el caso base en la definición inductiva de los tipos de sesión introducida en la sección 2.6). Intuitivamente, dado que el proceso `inact` representa el fin del proceso, no es entonces posible comunicación alguna, y por tanto los canales tienen que tener tipo `end`.

$$\frac{\Delta \text{ sólo contiene tipos } \texttt{end}}{\Theta;\Gamma \vdash \texttt{inact} \triangleright \Delta}\ [\textsc{Inact}]$$

Volviendo al ejemplo de la sección anterior, para tipar el proceso **Cliente**|**Cajero** introducimos la regla [Conc]. Esta determina que si un proceso P es tipable bajo los contextos Θ, Γ y Δ, y el proceso Q es tipable bajo los contextos Θ, Γ y Δ', entonces el proceso $P \mid Q$ es tipable bajo los contextos Θ, Γ y $\Delta \otimes \Delta'$. Utilizar el mismo contexto Γ para tipar los procesos que forman la composición paralela garantiza el mismo uso para los puertos declarados en ellos. La operación $\otimes$ "une" la información de dos contextos siempre que éstos sean congruentes. Dos contextos son congruentes si, en el caso que ambos tipen un mismo canal, al menos uno de ellos le asocia el tipo `end` (no permitiendo comunicación alguna). La idea del uso del operador $\otimes$ en esta regla es no permitir interferencias en la comunicación punto a punto, esto es, que sólo dos procesos puedan estar comunicándose a través de un mismo canal en todo momento. Si un mismo extremo de canal se utiliza en los procesos P y Q, entonces su información de tipo va ser distinta de `end` en ambos contextos Δ y Δ', con lo cual estos no serán congruentes.

$$\frac{\Theta;\Gamma \vdash P \triangleright \Delta \qquad \Theta;\Gamma \vdash Q \triangleright \Delta'}{\Theta;\Gamma \vdash P \mid Q \triangleright \Delta \otimes \Delta'}\ [\textsc{Conc}]$$

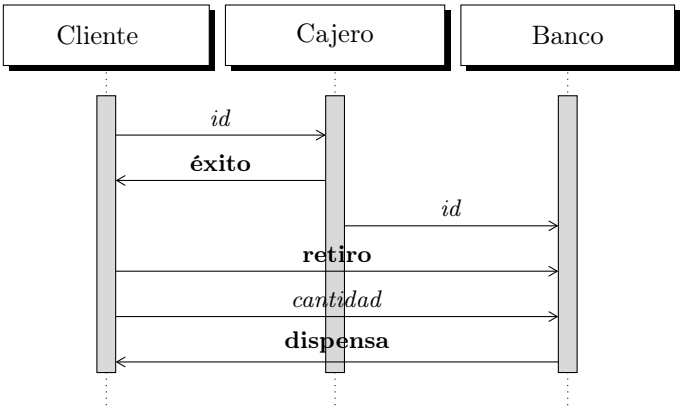
Utilizando la regla [Conc] podemos terminar la asignación de tipos para el ejemplo del proceso **Cliente** | **Cajero**.

$$\frac{\emptyset;\{a{:}\langle\alpha,\overline{\alpha}\rangle\} \vdash \textbf{Cliente} \triangleright \emptyset \qquad \emptyset;\{a{:}\langle\alpha,\overline{\alpha}\rangle\} \vdash \textbf{Cajero} \triangleright \emptyset}{\emptyset;\{a{:}\langle\alpha,\overline{\alpha}\rangle\} \vdash \textbf{Cliente} \mid \textbf{Cajero} \triangleright \emptyset}\ [\textsc{Conc}]$$

2.2. Cajero Automático con Delegación

El poder real del cálculo π, CCS y cálculos de procesos similares proviene de la habilidad de enviar y recibir canales, permitiendo así la *delegación* de canales de comunicación. Para introducir este concepto consideramos ahora una versión más compleja del ejemplo ya presentado, donde para completar el protocolo el **Cajero** delega las acciones sobre la cuenta directamente a otro proceso que representa a un **Banco**.

En la imagen siguiente podemos ver el protocolo en un escenario de retiro exitoso de dinero, donde apreciamos que luego de la autenticación exitosa el **Cajero** envía la identificación al **Banco**, y posteriormente delega todo el trabajo directamente al **Banco** sin que el **Cliente** pueda percibir este cambio. El escenario en el que la autenticación falla es idéntico al ya presentado en la sección anterior, mientras que los escenarios de depósito y sobregiro son análogos al de retiro exitoso que presentamos a continuación.



Se agrega para esto una nueva sesión de comunicación entre el **Cajero** y el **Banco**, sobre la cual inicialmente el **Cajero** envía el *id* del **Cliente**, para luego enviar su extremo del canal de comunicación acordado antes con el **Cliente**. Luego de esto el **Banco** se comporta tal cual lo hacía el **Cajero** en el ejemplo anterior. Así el **Cliente** no tiene cambio alguno en su comportamiento, por lo que tiene el mismo tipo de sesión, y el mismo proceso descritos anteriormente. El tipo de sesión que especifica el protocolo de comunicación del canal entre el **Cliente** y el **Cajero** visto en 1.2 no se altera, simplemente agregamos otro canal de comunicación entre el **Cajero** y **Banco**, sobre el cual delegamos parte del uso del canal acordado entre el **Cliente** y el **Cajero**. En 2.1 se define el tipo de sesión que describe el protocolo de este nuevo canal de comunicación entre el **Cajero** y el **Banco** desde el punto de vista del **Cajero**. Este tipo describe mediante el operador ! que por este canal enviamos ahora un canal de comunicación. Entre los paréntesis rectos que siguen al operador ! tenemos ahora el tipo que describe el uso del canal enviado. Luego de enviar este canal describimos el fin de la comunicación mediante la primitiva **end**. El **Cajero** utiliza el canal de comunicación con el **Cliente** recibiendo primero su *id* y luego enviando la opción **fallo** o **éxito** según corresponda, para finalmente enviar este canal al **Banco** en caso que la autenticación haya sido exitosa. Podemos entonces apreciar que el tipo del canal enviado es la continuación del tipo presentado en 1.2 luego de la selección de la opción **éxito**.

$$
\begin{array}{lll}
![& \&\{\textbf{depósito}: & ?[\texttt{nat}];![\texttt{nat}];\texttt{end} \\
 & ,\textbf{retiro}: & ?[\texttt{nat}];\oplus\{\textbf{dispensa}:\texttt{end} \\
 & & \quad ,\textbf{sobregiro}:\texttt{end}\} \quad];\texttt{end}
\end{array}
\tag{2.1}
$$

El tipo de sesión complementario al anterior, o sea la descripción del canal desde el punto de vista del **Banco**, se obtiene simplemente cambiando la primera aparición del operador ! al comienzo del tipo por su operador complementario ?.

Podemos ver en 2.2 una posible implementación de un nuevo proceso **Cajero′**, donde ahora en la rama exitosa se acuerda un nuevo canal de comunicación representado por la variable z a través del nombre de puerto b, mediante la expresión `accept` $b(z)$ `in` .... Por este canal enviamos luego el extremo del canal de comunicación y utilizando la primitiva `throw`. El protocolo seguido por la variable de canal z es el especificado por el tipo de sesión 2.1 descrito anteriormente.

$$\begin{array}{rl} \textbf{Cajero}' = \texttt{request}\ a(y)\ \texttt{in} & \\ \quad (y?(id)\ \texttt{in} & \\ \quad\quad (\texttt{if}\ldots\ \ \texttt{then}\ \ h \triangleleft \textbf{fallo}; \texttt{inact} & \\ \quad\quad\quad\ \ \texttt{else}\ \ y \triangleleft \textbf{éxito}; & \\ \quad\quad\quad\quad\ \ \texttt{accept}\ b(z)\ \texttt{in}\ (\texttt{throw}\ z[y]; \texttt{inact}))) & \end{array} \tag{2.2}$$

En 2.3 se define un posible proceso dual al anterior correspondiente al **Banco**. En este ejemplo el **Banco** primero acuerda un nuevo canal de comunicación representado por la variable de canal u a través del puerto b. Por este canal posteriormente se recibe el canal de comunicación v mediante la primitiva `catch` (operación dual a `throw`). Luego a través del canal v recibido el **Banco** se comunica con el **Cliente**, . De esta forma el **Banco** puede termina el trabajo hecho por el **Cajero** en el ejemplo anterior. El canal u respeta la especificación de comunicación dada por el tipo dual al presentado en 2.1.

$$\begin{array}{rl} \textbf{Banco} = \texttt{request}\ b(u)\ \texttt{in} & \\ \quad (\texttt{catch}\ u(v)\ \texttt{in}\ \ (v \triangleright \{\textbf{depósito}: & v?(cantidad)\ \texttt{in} \\ & \quad (\ldots; \\ & \quad v![saldo]; \\ & \quad \texttt{inact}) \\ \quad\quad , \textbf{retiro}: & v?(cantidad)\ \texttt{in} \\ & \quad (\texttt{if}\ldots\texttt{then}\ v \triangleleft \textbf{dispensar}; \\ & \qquad\qquad \texttt{inact} \\ & \qquad\quad \texttt{else}\ v \triangleleft \textbf{sobregiro}; \\ & \qquad\qquad \texttt{inact})\})) \end{array} \tag{2.3}$$

Componemos en paralelo ahora estos dos procesos con el proceso **Cliente** definido anteriormente para obtener el proceso que representa el sistema en su totalidad.

$$\textbf{Cliente} \mid \textbf{Cajero}' \mid \textbf{Banco}$$

2.3. Reglas de Delegación de Canales

En el ejemplo anterior introdujimos las primitivas `throw` y `catch`, para las cuales vemos a continuación sus respectivas reglas de tipado. Estas reglas son similares a la de los operadores ! y ?, sólo que actúan sobre tipos de canal pertenecientes al contexto Δ, y no de tipos básicos pertenecientes al contexto Γ. Otra diferencia fundamental de la regla [Thr] es que hace que el nombre del canal enviado desaparezca del contexto, asegurando que éste no se pueda

utilizar más, lo cual tiene el fin de evitar interferencia en la comunicación punto a punto.

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k{:}\beta}{\Theta; \Gamma \vdash \texttt{throw}\ k[k']; P\ \triangleright \Delta \cdot k{:}![\alpha]; \beta \cdot k'{:}\alpha}\ [\textsc{Thr}]$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k{:}\beta \cdot x{:}\alpha}{\Theta; \Gamma \vdash \texttt{catch}\ k(x)\ \texttt{in}\ P\ \triangleright \Delta \cdot k{:}?[\alpha]; \beta}\ [\textsc{Cat}]$$

Usamos la regla [Cat] en la deducción de tipos 2.1 para el proceso visto en 2.3. Denotamos como **SubBanco** al proceso resultante de quitar las primitivas `request` y `catch` al proceso anterior, y como γ al tipo de sesión mostrado en 2.4, el cual especifica el protocolo seguido por el **Banco** con el **Cliente**.

$$\begin{array}{rll} \&\{\textbf{depósito}: & ?[\texttt{nat}]; ![\texttt{nat}]; \texttt{end} & \\ \ ,\textbf{retiro}: & ?[\texttt{nat}]; \oplus\{\textbf{dispensa}: \texttt{end} & (2.4) \\ & \qquad\qquad ,\textbf{sobregiro}: \texttt{end}\} & \end{array}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\emptyset; \{a{:}\langle\alpha,\overline{\alpha}\rangle, b{:}\langle![\gamma]; \texttt{end}, ?[\gamma]; \texttt{end}\rangle\} \vdash \textbf{SubBanco} \triangleright \{u{:}\texttt{end}, v{:}\gamma\}}\ [\textsc{Cat}]}{\emptyset; \{a{:}\langle\alpha,\overline{\alpha}\rangle, b{:}\langle![\gamma]; \texttt{end}, ?[\gamma]; \texttt{end}\rangle\} \vdash \texttt{catch}\ u(v)\ \texttt{in}\ \textbf{SubBanco} \triangleright \{u{:}?[\gamma]; \texttt{end}\}}\ [\textsc{Cat}]}{\emptyset; \{a{:}\langle\alpha,\overline{\alpha}\rangle, b{:}\langle![\gamma]; \texttt{end}, ?[\gamma]; \texttt{end}\rangle\} \vdash \underbrace{\texttt{request}\ b(u)\ \texttt{in}\ (\texttt{catch}\ u(v)\ \texttt{in}\ \textbf{SubBanco})}_{\textbf{Banco}} \triangleright \emptyset}\ [\textsc{Req}]$$

Figura 2.1: Ejemplo de asignación de tipos utilizando la regla [Cat]

Vemos en la figura 2.2 la utilización de la regla [Thr] en la deducción de tipos para el proceso `accept` $b(z)$ `in` (`throw` $z[y]$; `inact`) que forma parte de la rama exitosa del proceso visto en 2.2. Observar que la regla [Thr] quita la información del canal y del contexto, para garantizar que este canal no pueda seguir siendo usado.

$$\dfrac{\dfrac{\dfrac{\{z{:}\texttt{end}\}\text{ sólo contiene tipos }\texttt{end}}{\emptyset; \{a{:}\langle\alpha,\overline{\alpha}\rangle, b{:}\langle![\gamma]; \texttt{end}, ?[\gamma]; \texttt{end}\rangle\} \vdash \texttt{inact} \triangleright \{z{:}\texttt{end}\}}\ [\textsc{Inact}]}{\emptyset; \{a{:}\langle\alpha,\overline{\alpha}\rangle, b{:}\langle![\gamma]; \texttt{end}, ?[\gamma]; \texttt{end}\rangle\} \vdash \texttt{throw}\ z[y]; \texttt{inact} \triangleright \{z{:}![\gamma]; \texttt{end}, y{:}\gamma\}}\ [\textsc{Thr}]}{\emptyset; \{a{:}\langle\alpha,\overline{\alpha}\rangle, b{:}\langle![\gamma]; \texttt{end}, ?[\gamma]; \texttt{end}\rangle\} \vdash \texttt{accept}\ b(z)\ \texttt{in}\ (\texttt{throw}\ z[y]; \texttt{inact}) \triangleright \{y{:}\gamma\}}\ [\textsc{Acc}]$$

Figura 2.2: Ejemplo de asignación de tipos utilizando la regla [Thr]

2.4. Factorial

Introducimos ahora un ejemplo extraído de [3] para ilustrar el uso de procedimientos. En este ejemplo se define un proceso que calcula el factorial de un número recibido a través de un canal.

$$
\begin{array}{l}
\texttt{def}\ \mathrm{Fact}(f) = \quad \texttt{accept}\ f(x)\ \texttt{in}\ x?(y)\ \texttt{in} \\
\qquad\qquad\qquad\quad \texttt{if}\ y = 1\ \texttt{then}\ x![1]; \texttt{inact} \\
\qquad\qquad\qquad\quad \texttt{else}\ (\nu b)(\mathrm{Fact}[b] \mid \texttt{request}\ b(x')\ \texttt{in}\ x'![y-1]; x'?(z)\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad x![y * z]; \texttt{inact}) \\
\texttt{in}\ (\nu f)(\mathrm{Fact}[f] \mid \texttt{request}\ f(x'')\ \texttt{in}\ x''![3]; x''?(y)\ \texttt{in}\ \texttt{inact})
\end{array}
$$

Se define una variable de proceso *Fact* la cual recibe como argumento un nombre de puerto f, sobre el cual primero acuerda un canal de comunicación representado por la variable x, por el cual recibe un entero y. En el caso base cuando $y = 1$ se devuelve 1 también por el canal x, y se termina el proceso que define el procedimiento. En el caso recursivo se declara un puerto fresco b, sobre el cual se componen en paralelo: una llamada al mismo procedimiento con este nuevo puerto como argumento y un proceso que acuerda un canal almacenado en la variable x' sobre este puerto b. Luego por x' manda $y - 1$, y recibe en la variable z el resultado para la invocación anterior (o sea el factorial de $y - 1$). Finalmente devuelve en el canal x la multiplicación este resultado con y, o sea el factorial de y.

En la última línea de la definición se realiza la invocación al procedimiento anteriormente descrito con un argumento concreto, en este caso 3.

En el Apéndice C se muestra la deducción completa de la asignación de tipos para este proceso.

2.5. Cálculo de Procesos

En esta sección introducimos las definiciones completas del cálculo de procesos basados principalmente en [6], que incluye la sintaxis de los procesos, y la semántica operacional del mismo.

La definición inductiva del conjunto de los procesos se presenta en la figura 2.3.

Los nombres de variables, de variables de canal y de puertos se denotan por $a, b, \ldots, x, y, \ldots$. Las *constantes* que incluyen a los conjuntos mencionados antes, y además los enteros y booleanos, son denotadas por $c, c', \ldots$. Las expresiones son denotadas con $e, e', \ldots$ y las *etiquetas* por $l, l', \ldots$. Las *variables de procesos* se denotan por $X, Y, \ldots$ y los procesos mediante $P, Q, \ldots$.

Los nombres de los canales están polarizados de forma de identificar cuál de los extremos del canal k se quiere representar; así luego ambos extremos de un mismo canal pueden coexistir en un mismo contexto de canales, cada uno aportando la información de tipo correspondiente a cada extremo. A las variables de canal o valores de canales polarizados se los representa con k. Los valores de canales o constantes de canal son denotados con κ^p, donde p es la polaridad del canal. Las variables de canal en cambio no precisan polaridades ya que el sistema de reglas asignación de tipos hace que no se pueda dar el caso en que variables de ambos extremos de un mismo canal deban coexistir en un mismo contexto. Más específicamente, las reglas [Acc] y [Req] quitan la información de las variables de canales del contexto de canales, y la asignan al puerto sobre el cual se establece la comunicación en el contexto de puertos. Se puede ver en la regla de reducción [Link], presentada en la figura 2.5, cómo la

P	$::=$	$\texttt{request}\ a(x)\ \texttt{in}\ P$	pedir sesión
	$\mid$	$\texttt{accept}\ a(x)\ \texttt{in}\ P$	aceptar sesión
	$\mid$	$k![\tilde{e}];P$	enviar información
	$\mid$	$k?(\tilde{x})\ \texttt{in}\ P$	recibir información
	$\mid$	$k \triangleleft l_j; P$	seleccionar etiquetas
	$\mid$	$k \triangleright \{l_1{:}P_1 \ldots l_n{:}P_n\}$	bifurcar internamente o sobre etiquetas
	$\mid$	$\texttt{throw}\ k[k'];P$	enviar canal
	$\mid$	$\texttt{catch}\ k(x)\ \texttt{in}\ P$	recibir canal
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	bifurcar condicionalmente
	$\mid$	$P \mid Q$	componer en paralelo
	$\mid$	$\texttt{inact}$	inacción
	$\mid$	$(\nu a)P$	ocultar puerto
	$\mid$	$(\nu\kappa)P$	ocultar canal
	$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	recursión
	$\mid$	$X[\tilde{e}\tilde{k}]$	variables de proceso
e	$::=$	c	constantes
	$\mid$	$e + e \mid e - e \mid e \times e \mid \texttt{not}(e) \mid \ldots$	operadores
D	$::=$	$X_1(\tilde{x_1}\tilde{y_1}) = P_1\ \texttt{and}\ \ldots X_n(\tilde{x_n}\tilde{y_n}) = P_n$	declaración de la recursión
k	$::=$	$x \mid \kappa^p$	variable de canal o valores
p	$::=$	$+ \mid -$	polaridades de canales

Figura 2.3: Sintaxis de los Procesos

comunicación sólo es posible entre canales de polaridad complementaria, donde el complemento de las polaridades se define como $\overline{+} = -$ y $\overline{-} = +$.

Las variables $x, \ldots$ son *ligadas* por las siguientes expresiones: $k?(\tilde{x})$ `in` P, $X(\tilde{x}\tilde{y}) = P$, `request` $a(x)$ `in` P, `accept` $a(x)$ `in` P, y `catch` $k(x)$ `in` P. Los nombres $a, \ldots$ son ligados en la expresión: $(\nu a)P$. Los canales y las variables de procesos en cambio sólo son ligados en las expresiones: $(\nu\kappa)P$ y `def` D `in` P respectivamente. Para un proceso P, fpv(P) denota el conjunto de las variables de procesos libres en P, fn(P) denota los nombres de variables comúnes y de puertos libres, mientras que fc(P) denota los canales libres en P. Se define fu(P) = fn(P) $\cup$ fc(P). También se define dpv($X_1(\tilde{x_1}\tilde{y_1}) = P_1$ `and` $\ldots X_n(\tilde{x_n}\tilde{y_n}) = P_n$) = $\{X_1, \ldots, X_n\}$.

La *congruencia estructural* es la relación más pequeña de congruencia en procesos que incluye las ecuaciones de la figura 2.4.

La semántica operacional de este cálculo se define mediante la relación de *reducción* entre procesos, denotada por $\rightarrow$. Esta relación es definida como la menor relación generada por las reglas definidas en la figura 2.5, donde $e \downarrow c$ significa que la expresión e evalúa a la constante c.

El cálculo de procesos con polaridades fue introducido en [8], cuando originalmente los Tipos de Sesión fueron definidos sin polaridades. En [7] se explica detalladamente el por qué fueron introducidas las polaridades en el cálculo de procesos. También distingue entre el lenguaje de programación, o sea el sub cálculo de procesos a ser utilizado por los programadores, y el lenguaje utilizado en tiempo de ejecución o *runtime*. Establece que los programadores no deben lidiar con los canales directamente sino manipularlos a través de variables de

$$P \equiv Q \text{ if } P \equiv_\alpha Q$$
$$P \mid \texttt{inact} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\nu u)P \mid Q \equiv (\nu u)(P \mid Q) \quad \text{Si } u \notin \text{fu}(Q)$$
$$(\nu u)\texttt{inact} \equiv \texttt{inact}$$
$$\texttt{def } D \texttt{ in inact} \equiv \texttt{inact}$$
$$(\nu u)\texttt{def } D \texttt{ in } P \equiv \texttt{def } D \texttt{ in } (\nu u)P \quad \text{si } u \notin \text{fu}(D)$$
$$(\texttt{def } D \texttt{ in } P) \mid Q \equiv \texttt{def } D \texttt{ in } (P \mid Q) \quad \text{si } \text{dpv}(D) \cap \text{fpv}(Q) = \emptyset$$
$$(\texttt{def } D \texttt{ in } (\texttt{def } D' \texttt{ in } P) \equiv \texttt{def } D \texttt{ and } D' \texttt{ in } P \quad \text{si } \text{dpv}(D) \cap \text{dpv}(D') = \emptyset$$

Figura 2.4: Congruencia estructural

$$(\texttt{accept } a(x) \texttt{ in } P_1) \mid (\texttt{request } a(x) \texttt{ in } P_2) \to (\nu\kappa)(P_1[\kappa^+/x] \mid P_2[\kappa^-/x]) \quad \text{[Link]}$$
$$(\kappa^p![\tilde{e}]; P_1) \mid (\kappa^{\overline{p}}?(\tilde{x}) \texttt{ in } P_2) \to P_1 \mid P_2[\tilde{c}/\tilde{x}] \text{ si } (\tilde{e} \downarrow \tilde{c}) \quad \text{[Com]}$$
$$(\kappa^p \triangleleft l_i; P) \mid (\kappa^{\overline{p}} \triangleright \{l_1{:}P_1, \ldots, l_n{:}P_n\}) \to P \mid P_i \text{ con } (1 \le i \le n) \quad \text{[Label]}$$
$$(\texttt{throw } \kappa^p[\kappa'^q]; P_1) \mid (\texttt{catch } \kappa^{\overline{p}}(x) \texttt{ in } P_2) \to P_1 \mid P_2[k'^q/x] \quad \text{[Pass]}$$
$$\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \to P_1 \text{ si } (e \downarrow \texttt{true}) \quad \text{[If1]}$$
$$\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \to P_2 \text{ si } (e \downarrow \texttt{false}) \quad \text{[If2]}$$
$$P \to P' \Rightarrow (\nu u)P \to (\nu u)P' \quad \text{[Scop]}$$
$$P \to P' \Rightarrow P \mid Q \to P' \mid Q \quad \text{[Par]}$$
$$P \to P' \Rightarrow \texttt{def } D \texttt{ in } P \to \texttt{def } D \texttt{ in } P' \quad \text{[Defin]}$$
$$(P \equiv P') \wedge (P' \to Q') \wedge (Q' \equiv Q) \Rightarrow (P \to Q) \quad \text{[Str]}$$
$$\texttt{def } D \texttt{ in } (X[\tilde{e}\tilde{\kappa^p}] \mid Q) \to \texttt{def } D \texttt{ in } (P[\tilde{c}/\tilde{x}][\tilde{\kappa^p}/\tilde{y}] \mid Q) \text{ si } (\tilde{e} \downarrow \tilde{c}) \wedge (X(\tilde{x}\tilde{y}) = P \in D) \quad \text{[Def]}$$

Figura 2.5: Reducción

programación convencionales. Los canales son entidades que sólo aparecen en tiempo de ejecución (al aplicar reducciones), finalmente sustituyendo a las variables en el código. Esta última afirmación se puede apreciar en los ejemplos introducidos hasta el momento, donde no aparecen canales con polaridades en las distintas definiciones de procesos dadas.

2.6. Sistema de Tipos

Presentamos ahora la definición inductiva del sistema de asignación de tipos a procesos (Sistema de Tipos).

En la figura 2.6 se define la sintaxis de los tipos. Mediante S se denota al conjunto de los tipos comunes (*Sorts*), mientras que α define los Tipos de Sesión o simplemente tipos.

La expresión $\mu t.\alpha$ introduce la recursión en los tipos, y denota que el canal comienza comportándose según α y cuando t es encontrado, recomienza el comportamiento dictado por α.

Definimos a continuación la función complemento que devuelve el tipo de sesión dual o complementario a otro tipo de sesión dado.

$$
\begin{array}{rcl}
S & ::= & \texttt{nat} \mid \texttt{bool} \mid \langle \alpha, \overline{\alpha} \rangle \\
\alpha & ::= & ?[\tilde{S}]; \alpha \mid ?[\tilde{\alpha}]; \alpha \\
& \mid & ![\tilde{S}]; \alpha \mid ![\tilde{\alpha}]; \alpha \\
& \mid & \&\{l_1{:}\alpha_1 \ldots l_n{:}\alpha_n\} \\
& \mid & \oplus\{l_1{:}\alpha_1 \ldots l_n{:}\alpha_n\} \\
& \mid & \mu t.\alpha \\
& \mid & t \\
& \mid & \texttt{end}
\end{array}
$$

Figura 2.6: Sintaxis de los tipos

$$
\begin{array}{lll}
\overline{?[\tilde{S}]; \alpha} = ![\tilde{S}]; \overline{\alpha} & \overline{\oplus\{l_i{:}\alpha_i\}_{i \in I}} = \&\{l_i{:}\overline{\alpha}\}_{i \in I} & \overline{?[\alpha]; \beta} = ![\alpha]; \overline{\beta} \\
\overline{![\tilde{S}]; \alpha} = ?[\tilde{S}]; \overline{\alpha} & \overline{\&\{l_i{:}\alpha_i\}_{i \in I}} = \oplus\{l_i{:}\overline{\alpha}\}_{i \in I} & \overline{![\alpha]; \beta} = ?[\alpha]; \overline{\beta} \\
\overline{\texttt{end}} = \texttt{end} & \overline{\mu t.\alpha} = \mu t.\overline{\alpha} & \overline{t} = t
\end{array}
\tag{2.5}
$$

En la siguiente figura se presenta el sistema completo de asignación de tipos a procesos con polaridades tal cual es presentado en [6][1].

$$
\Gamma \cdot a{:}S \vdash a \triangleright S \ [\textsc{NameI}] \quad \Gamma \vdash 1 \triangleright \texttt{nat} \ [\textsc{Nat}] \quad \Gamma \vdash \texttt{true} \triangleright \texttt{bool} [\textsc{Bool}]
$$

$$
\frac{\Gamma \ \vdash e_i \triangleright \texttt{nat}}{\Gamma \vdash e_1 + e_2 \triangleright \texttt{nat}} \ [\textsc{Sum}]
$$

Figura 2.7: Sistema de asignación de tipos a expresiones

$$
\frac{\Delta \text{ sólo contiene tipos } \texttt{end}}{\Theta; \Gamma \vdash \texttt{inact} \triangleright \Delta} \ [\textsc{Inact}]
$$

$$
\frac{\Gamma \vdash a \ \triangleright \langle \alpha, \overline{\alpha} \rangle \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot x{:}\alpha}{\Theta; \Gamma \vdash \texttt{accept}\ a(x)\ \texttt{in}\ P \triangleright \Delta} \ [\textsc{Acc}]
$$

$$
\frac{\Gamma \vdash a \ \triangleright \langle \alpha, \overline{\alpha} \rangle \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot x{:}\overline{\alpha}}{\Theta; \Gamma \vdash \texttt{request}\ a(x)\ \texttt{in}\ P \triangleright \Delta} \ [\textsc{Req}]
$$

$$
\frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot k{:}\alpha}{\Theta; \Gamma \vdash k![\tilde{e}]; P \ \triangleright \Delta \cdot k{:}![\tilde{S}]; \alpha} \ [\textsc{Send}]
$$

[1] Introducimos una pequeña modificación en la regla [Conc], en donde renombramos la operación . por $\otimes$. Esta operación, al igual que ., exige que los contextos sean disjuntos para poder así unirlos

$$\frac{\Theta;\Gamma\cdot\tilde{x}{:}\tilde{S}\vdash P\triangleright\Delta\cdot k{:}\alpha}{\Theta;\Gamma\vdash k?(\tilde{x})\ \texttt{in}\ P\ \triangleright\Delta\cdot k{:}?[\tilde{S}];\alpha}\ [\textsc{Rcv}]$$

$$\frac{\Theta;\Gamma\vdash P_1\triangleright\Delta\cdot k{:}\alpha_1\qquad\ldots\qquad\Theta;\Gamma\vdash P_n\triangleright\Delta\cdot k{:}\alpha_n}{\Theta;\Gamma\vdash k\triangleright\{l_1{:}P_1\ldots l_n{:}P_n\}\triangleright\Delta\cdot k{:}\&\{l_1{:}\alpha_1\ldots l_n{:}\alpha_n\}}\ [\textsc{Br}]$$

$$\frac{\Theta;\Gamma\vdash P\triangleright\Delta\cdot k{:}\alpha_j}{\Theta;\Gamma\vdash k\triangleleft l_j;P\triangleright\Delta\cdot k{:}\oplus\{l_1{:}\alpha_1\ldots l_n{:}\alpha_n\}}\ (1\leq j\leq n)[\textsc{Sel}]$$

$$\frac{\Theta;\Gamma\vdash P\triangleright\Delta\cdot k{:}\beta}{\Theta;\Gamma\vdash\texttt{throw}\ k[k'];P\ \triangleright\Delta\cdot k{:}![\alpha];\beta\cdot k'{:}\alpha}\ [\textsc{Thr}]$$

$$\frac{\Theta;\Gamma\vdash P\triangleright\Delta\cdot k{:}\beta\cdot x{:}\alpha}{\Theta;\Gamma\vdash\texttt{catch}\ k(x)\ \texttt{in}\ P\ \triangleright\Delta\cdot k{:}?[\alpha];\beta}\ [\textsc{Cat}]$$

$$\frac{\Theta;\Gamma\vdash P\triangleright\Delta\qquad\Theta;\Gamma\vdash Q\triangleright\Delta'}{\Theta;\Gamma\vdash P\mid Q\triangleright\Delta\otimes\Delta'}\ [\textsc{Conc}]$$

$$\frac{\Gamma\vdash e\triangleright\texttt{bool}\qquad\Theta;\Gamma\vdash P\triangleright\Delta\qquad\Theta;\Gamma\vdash Q\triangleright\Delta}{\Theta;\Gamma\vdash\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q\triangleright\Delta}\ [\textsc{If}]$$

$$\frac{\Theta;\Gamma\cdot a{:}S\vdash P\triangleright\Delta}{\Theta;\Gamma\vdash(\nu a)P\ \triangleright\Delta}\ [\textsc{Nres}]$$

$$\frac{\Theta;\Gamma\vdash P\triangleright\Delta\cdot\kappa^{+}{:}\alpha\cdot\kappa^{-}{:}\overline{\alpha}}{\Theta;\Gamma\vdash(\nu\kappa)P\ \triangleright\Delta}\ [\textsc{Cres}]$$

$$\frac{\Theta;\Gamma\vdash P\triangleright\Delta\qquad\kappa^{+}\notin\Delta\wedge\kappa^{\text{-}}\notin\Delta}{\Theta;\Gamma\vdash(\nu\kappa)P\ \triangleright\Delta}\ [\textsc{Cres'}]$$

$$\frac{\Delta\text{ sólo contiene tipos }\texttt{end}\qquad\Gamma\vdash\tilde{e}\triangleright\tilde{S}}{\Theta\cdot X{:}\tilde{S}\tilde{\alpha};\Gamma\vdash X[\tilde{e}\tilde{k}]\triangleright\Delta\cdot\tilde{k}{:}\tilde{\alpha}}\ [\textsc{Var}]$$

$$\frac{\Theta\cdot X{:}\tilde{S}\tilde{\alpha};\Gamma\cdot\tilde{x}{:}\tilde{S}\vdash P\triangleright\Delta\cdot\tilde{k}{:}\tilde{\alpha}\qquad\Theta\cdot X{:}\tilde{S}\tilde{\alpha};\Gamma\vdash Q\triangleright\Delta}{\Theta;\Gamma\vdash\texttt{def}\ X(\tilde{x}\tilde{k})=P\ \texttt{in}\ Q\triangleright\Delta}\ [\textsc{Def}]$$

Figura 2.8: Sistema de asignación de tipos a procesos

En [6] se demuestra que este cálculo tiene la propiedad de *subject reduction*, el cual establece que si un proceso P es tipable en ciertos contextos bajo las reglas de tipado presentada en 2.6, y P reduce a Q, entonces Q es tipable bajo los mismos contextos. A partir de este resultado también se demuestra la propiedad de *type safety*, que establece que si un proceso es tipable entonces nunca reduce a un *error*. Para formalizar la noción de type safety y de error se definen los siguientes conceptos: un $k-$*proceso* es cualquier proceso que en su primera acción involucra al canal k (como por ejemplo $k![\tilde{e}];P$ o $\texttt{catch}\ k(x)\ \texttt{in}\ P$); un proceso $k-$*reducible* es la composición en paralelo de dos $k-$procesos de alguna de las siguientes formas:

- $(\kappa^p![\tilde{e}]; P_1) \mid (\kappa^{\overline{p}}?(\tilde{x})\ \texttt{in}\ P_2)$
- $(\kappa^p \triangleleft l_i; P)) \mid (\kappa^{\overline{p}} \triangleright \{l_1{:}P_1, \ldots, l_n{:}P_n\})$
- $(\texttt{throw}\ \kappa^p[k'^q]; P_1) \mid (\texttt{catch}\ \kappa^{\overline{p}}(x)\ \texttt{in}\ P_2)$

Entonces un proceso P es un error si contiene $k-$procesos compuestos en paralelo que no son $k-$reducibles, para algún k.

2.7. Weakening Lemma

La operación de debilitamiento (*weakening*) de los contextos permite agregar información a los contextos de tipado. Presentamos la propiedad que afirma que esta operación preserva la validez del juicio de tipado, bajo la premisa de no sobreescribir información ya existente en el contexto (*weakening lemma*). En los siguientes capítulos utilizaremos este resultado para justificar la formalización del sistema de tipos, y también la demostración de corrección del algoritmo de inferencia de tipos.

Teorema 2.7.1 (*Weakening Lemma*)**.** *Sean* Θ*,* Γ*,* Δ *y* P *tales que se satisface* $\Theta; \Gamma \vdash P \triangleright \Delta$*.*

Entonces se satisfacen las siguientes afirmaciones:

1. *Si* $X \notin dom(\Theta)$ *, entonces* $\Theta \cdot X{:}\tilde{S}\tilde{\alpha}; \Gamma \vdash P \triangleright \Delta$*.*
2. *Si* $a \notin dom(\Gamma)$ *, entonces* $\Theta; \Gamma \cdot a{:}S \vdash P \triangleright \Delta$*.*
3. *Si* $k \notin dom(\Delta)$ *, entonces* $\Theta; \Gamma \vdash P \triangleright \Delta \cdot k{:}\texttt{end}$*.*

Demostración del Weakening *lema.* La demostración de este lema es por inducción en el árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$, y puede verse su desarrollo completo en el Apéndice A. □

2.8. Ejemplo de Delegación de Canales

Finalmente, presentamos un ejemplo sencillo que utilizaremos a lo largo de este trabajo para ejemplificar las nociones a formalizar. En este ejemplo se delega un canal de comunicación a través de la pareja de primitivas `throw-catch` ya vistas. Presentaremos la verificación completa del tipo del proceso bajo el sistema de reglas ya visto, y en el capítulo 7 comprobaremos que este tipo es efectivamente el mismo que el hallado por nuestro algoritmo de inferencia de tipos. Elegimos este ejemplo debido a que utiliza las principales primitivas del cálculo sin ser demasiado extenso.

Podemos identificar tres procesos concurrentes, los cuales llamamos: P_1, P_2 y P_3. Al comienzo existen solamente dos procesos compuestos que acuerdan canales de comunicación a través del puerto a. El primer proceso identifica este canal de comunicación como x y pasa a ejecutar el proceso P_1, mientras que el segundo lo identifica como z, y luego se subdivide, mediante otra composición paralela, en dos procesos concurrentes, los cuales identificamos como P_2 y P_3. Estos últimos acuerdan otro canal de comunicación a través del puerto b, que es llamado u dentro de P_2 y v dentro de P_3. Luego el proceso P_2 delega el canal u al proceso P_1 a través del canal de comunicación z, al cual P_1 nombra como y. Finalmente el proceso P_1 envía información al proceso P_3 a través de este canal delegado.

$$(\texttt{request}\ a(x)\ \texttt{in}\ \overbrace{(\texttt{catch}\ x(y)\ \texttt{in}\ (y![3,\texttt{true}];\texttt{inact}))}^{P_1})$$
$$\mid (\texttt{accept}\ a(z)\ \texttt{in}\ (\ \overbrace{(\texttt{accept}\ b(u)\ \texttt{in}\ (\texttt{throw}\ z[u];\texttt{inact}))}^{P_2}$$
$$\mid \overbrace{(\texttt{request}\ b(v)\ \texttt{in}\ (v?(w,o)\ \texttt{in}\ \texttt{inact}))}^{P_3}))$$

Figura 2.9: Proceso que delega un canal

Utilizamos el sistema de reglas de asignación de tipos para comprobar que este ejemplo es tipable bajo el contexto $\Gamma = \{a{:}\langle\alpha,\overline{\alpha}\rangle, b{:}\langle\beta,\overline{\beta}\rangle\}$, con $\alpha =![\beta]\,;\texttt{end}$, y $\beta =![\texttt{nat},\texttt{bool}];\texttt{end}$, y los restantes contextos vacíos. Aplicamos las reglas desde la conclusión hacia las premisas, en sentido "de abajo hacia arriba" o *bottom-up*. Podemos ver gráficamente esta deducción en la figura 2.10

$$\dfrac{\dfrac{\vdots\ \text{(figura 2.11)}}{\emptyset,\Gamma\vdash \texttt{request}\ a(x)\ \texttt{in}\ (\texttt{catch}\ x(y)\ \texttt{in}\ (y![3,\texttt{true}];\texttt{inact}))\ \triangleright\emptyset}\,[\textsc{Req}] \qquad \dfrac{\vdots\ \text{(figura 2.13)}}{\emptyset,\Gamma\vdash \texttt{accept}\ a(z)\ \texttt{in}\ ((\texttt{accept}\ b(u)\ \texttt{in}\ (\texttt{throw}\ z[u];\texttt{inact}))\mid (\texttt{request}\ b(v)\ \texttt{in}\ (v?(w,o)\ \texttt{in}\ \texttt{inact})))\ \triangleright\emptyset}\,[\textsc{Acc}]}{\emptyset,\Gamma\vdash \begin{array}{l}\texttt{request}\ a(x)\ \texttt{in}\ (\texttt{catch}\ x(y)\ \texttt{in}\ (y![3,\texttt{true}];\texttt{inact}))\\ \mid \texttt{accept}\ a(z)\ \texttt{in}\ ((\texttt{accept}\ b(u)\ \texttt{in}\ (\texttt{throw}\ z[u];\texttt{inact}))\mid (\texttt{request}\ b(v)\ \texttt{in}\ (v?(w,o)\ \texttt{in}\ \texttt{inact})))\end{array}\ \triangleright\emptyset}\,[\textsc{Conc}]$$

Figura 2.10: Ejemplo de asignación de tipos a un proceso

Comenzamos aplicando la regla [Conc] con los contextos anteriormente descritos, con lo cual debemos tipar los dos subprocesos que componen el proceso completo bajo los mismos contextos.

Estudiamos ahora la rama izquierda de esta derivación (correspondiente al árbol de la figura 2.10). Podemos ver en la figura 2.11 las inferencias de tipado seguidas en esta rama. Comenzamos aplicando la regla [Req], que exige, por un lado que se cumpla el juicio de expresiones $\Gamma\vdash a\triangleright\langle\alpha,\overline{\alpha}\rangle$, el cual se cumple aplicando la regla [NameI] de 2.7. Por otro lado [Req] exige que el proceso que sigue al `request` sea tipable bajo los mismos contextos, solamente agregando la información de que la variable de canal x tiene que tener tipo $\overline{\alpha}$ (pidiendo así que la variable de canal x sea utilizada siguiendo la especificación brindada por el contexto Γ para el puerto a). Aplicando la regla [Cat] se tiene que el tipo α debe ser de la forma $?[\alpha'];\beta'$, lo cual se cumple ya que tal cual definimos $\alpha =?[\beta];\texttt{end}$. Siguiendo esta regla tenemos que el proceso $y![3,\texttt{true}];\texttt{inact}$ tiene que ser tipable bajo los mismos contextos que antes, salvo el de puertos que tiene que tener la continuación del tipo α (o sea `end`) asignada a la variable de canal

x. Esta regla también asigna el tipo de sesión β a la nueva variable y que aparece en el alcance del contexto.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle}\,[\textsc{NameI}] \qquad \dfrac{\dfrac{\vdots \text{ (figura 2.12)}}{\emptyset, \Gamma \vdash \begin{array}{l} y![3, \texttt{true}]; \\ \texttt{inact} \end{array} \triangleright \left\{ \begin{array}{l} x{:}\texttt{end}, \\ y{:}![\texttt{nat}, \texttt{bool}]; \texttt{end} \end{array} \right\}}\,[\textsc{Send}]}{\emptyset, \Gamma \vdash \begin{array}{l} \texttt{catch } x(y) \texttt{ in} \\ \quad (y![3, \texttt{true}]; \\ \quad\ \texttt{inact}) \end{array} \triangleright \{x{:}\overline{\alpha}\}}\,[\textsc{Cat}]}{\emptyset, \Gamma \vdash \begin{array}{l} \texttt{request } a(x) \texttt{ in} \\ (\texttt{catch } x(y) \texttt{ in} \\ \quad (y![3, \texttt{true}]; \texttt{inact})) \end{array} \triangleright \emptyset}\,[\textsc{Req}]$$

Figura 2.11: Ejemplo de asignación de tipos a un proceso

En la figura 2.12 vemos que el proceso $y![3, \texttt{true}]; \texttt{inact}$ tipa correctamente en los contextos descritos anteriormente, finalizando así la derivación de tipos en esta rama.

$$\dfrac{\dfrac{}{\emptyset \vdash 3 \triangleright \texttt{nat}}\,[\textsc{Nat}] \qquad \dfrac{}{\emptyset \vdash \texttt{true} \triangleright \texttt{bool}}\,[\textsc{Bool}] \qquad \dfrac{\left\{ \begin{array}{l} x{:}\texttt{end}, \\ y{:}\texttt{end} \end{array} \right\} \text{ sólo contiene tipos } \texttt{end}}{\emptyset, \Gamma \vdash \texttt{inact} \triangleright \left\{ \begin{array}{l} x{:}\texttt{end}, \\ y{:}\texttt{end} \end{array} \right\}}\,[\textsc{Inact}]}{\emptyset, \Gamma \vdash \begin{array}{l} y![3, \texttt{true}]; \\ \texttt{inact} \end{array} \triangleright \left\{ \begin{array}{l} x{:}\texttt{end}, \\ y{:}![\texttt{nat}, \texttt{bool}]; \texttt{end} \end{array} \right\}}\,[\textsc{Send}]$$

Figura 2.12: Ejemplo de asignación de tipos a un proceso

Retomamos ahora la rama derecha del árbol visto en 2.10. En la figura 2.13 vemos el comienzo de la derivación de tipos para esta rama, la regla [Acc] abre dos ramas. En la rama izquierda se chequea que el contexto Γ brinde información del puerto a, lo cual se satisface y el tipo asignado a este puerto es $\langle \alpha, \overline{\alpha} \rangle$. En la rama derecha se pide que el proceso que sigue al `accept` tipe asignando a la variable de canal z el tipo de sesión α en el contexto de canales.

En 2.14 aplicamos la regla [Conc], la cual pide que los dos procesos que conforman la composición paralela sean tipables bajo los mismos contextos de tipos de variables de procesos y de puertos, pero el contexto de tipos de canales se debe dividir de forma disjunta, dado que un mismo extremo de comunicación no pueden ser usado por dos procesos simultáneamente. Así dejamos la información de la variable de canal z en la rama izquierda, debido a que solamente en esa rama es usado el extremo de canal z.

En la figura 2.15 damos la continuación de la derivación de rama izquierda del árbol anterior, donde se aplica la regla [Acc], la cual pide que exista información para el puerto b en el contexto Γ, lo que se cumple ya que Γ asigna el tipo $\langle \beta, \overline{\beta} \rangle$ a b. Esta regla pide también que el resto del proceso sea tipable agregando la asignación de tipos $u{:}\beta$ al contexto de canales. Luego podemos aplicar la regla

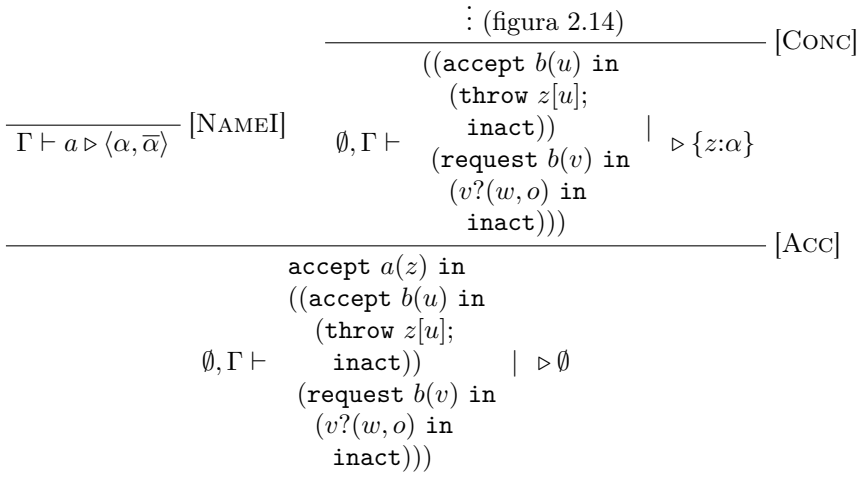


Figura 2.13: Ejemplo de asignación de tipos a un proceso

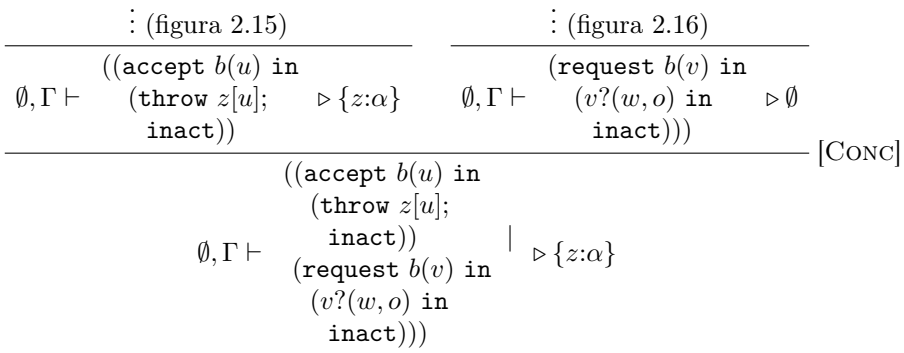


Figura 2.14: Ejemplo de asignación de tipos a un proceso

[Thr] en la rama derecha, debido a que $\alpha = ![\beta]; \texttt{end}$ con lo cual el tipo de la variable de canal z, o sea β, coincide con el tipo especificado como enviado por el tipo α. Terminamos esta rama aplicando la regla [Inact] dado que el contexto de canales contiene solamente tipos `end`.

En la imagen 2.16 vemos la última derivación que resta detallar, correspondiente a rama derecha de la figura 2.14. En ésta comenzamos aplicando la regla [Req] que pide que el puerto b tenga información de tipo en el contexto Γ, lo cual se satisface ya que Γ asigna el tipo $\langle \beta, \overline{\beta} \rangle$ a b. Esta regla también pide que el resto del proceso, sin la primitiva `request`, sea tipable agregando la asignación $v{:}\overline{\beta}$ al contexto de canales. Podemos aplicar la regla [Rcv] dado que $\overline{\beta} = ?[\texttt{nat}, \texttt{bool}]; \texttt{end}$, con lo cual agregamos al contexto Γ la información de tipo de las variables w y o, y dejamos en el contexto de canales la continuación del tipo β, o sea `end`, a la variable de canal v. Quedan así sólo tipos `end` en el contexto de canales lo que permite entonces aplicar finalmente la regla [Inact].

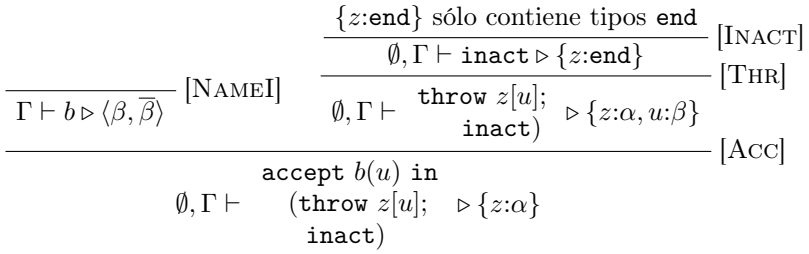


Figura 2.15: Ejemplo de asignación de tipos a un proceso

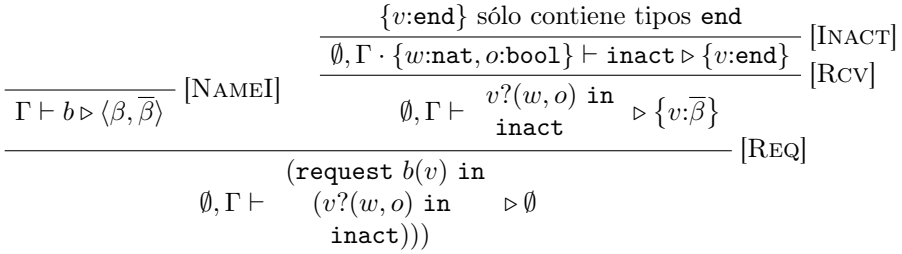


Figura 2.16: Ejemplo de asignación de tipos a un proceso

Capítulo 3

Inferencia de Tipos de Sesión

En este capítulo presentamos el algoritmo de inferencia de Tipos de Sesión propuesto en este trabajo. Dado un proceso sin ninguna clase de declaraciones de tipos el algoritmo infiere, si es que existen, los contextos de tipos bajo los cuales se satisface el juicio de tipado para este proceso.

3.1. Trabajo Relacionado

Los Tipos de Sesión aparecen en [1], mientras que la inferencia de Tipos de Sesión empieza a ser estudiada a partir de 2004 [12], y sigue siendo objeto de estudio en la actualidad [13, 14, 15, 16, 17]. A continuación discutimos los principales trabajos en esta área.

En [15] se hace un estudio de una forma simplificada de Tipos de Sesión, en un cálculo de procesos sintácticamente restringido que no brinda al programador la libertad de utilizar directamente los canales de sesión. En todo momento se pueden utilizar sólo dos canales simultáneamente. El primero es llamado canal *actual* sobre el que se pueden utilizar las primitivas usuales de envío y recepción de datos y de aceptación y selección de opciones, mientras que el otro canal sirve sólo para enviar información a un proceso llamado *padre*. Como consecuencia de no tener libertad en el uso de canales, tampoco se permite enviar y recibir canales, perdiéndose así la funcionalidad de delegación ejemplificada en el capítulo anterior. El cálculo presentado tiene primitivas de aceptación y selección de opciones, y se propone una forma original de inferencia de tipos que permite que procesos que seleccionan menos opciones tipen correctamente con procesos que ofrecen más opciones. En ese trabajo se simplifica el sistema de tipos quitándose los tipos recursivos. Se demuestran la propiedad de *type safety* del cálculo de procesos propuesto. Y también que uno de los sub algoritmos utilizados en la inferencia de tipos termina siempre. Finalmente se muestra la factibilidad del algoritmo propuesto mencionando su implementación en OCaml.

Por otra parte en [17] se estudian formas de inferir Tipos de Sesión mediante técnicas de análisis de código a partir de programas imperativos, los cuales consisten en una secuencia de acciones de comunicación.

Finalmente los demás trabajos relacionados se basan en embeber determinados cálculos de procesos con su respectivo sistema de tipos en lenguajes de programación ya existentes. En [12] se implementan Tipos de Sesión con un

único canal en Haskell, prohibiendo así el uso explícito de los canales. En ese trabajo se prueba la solidez del sistema embebido. También en [14] se implementan Tipos de Sesión en Haskell, pero agregando la posibilidad de usar múltiples canales de sesión. En contrapartida no demuestran ninguna propiedad de solidez del trabajo realizado. En esos dos trabajos se aprovecha el poderoso sistema de tipos polimórfico de Haskell, en particular se utilizan las clases de tipos con dependencias funcionales [29] para modelar así la progresión en el uso de los canales, ganando de esta forma la inferencia en los Tipos de Sesión embebidos a través del sistema de tipos del propio lenguaje anfitrión. En [13] se muestra una técnica un poco más general para codificar los Tipos de Sesión, ganando mayor portabilidad en el lenguaje anfitrión. De esta forma se puede codificar en otros lenguajes además de Haskell, como ML y Java. En particular se establece que la técnica es portable a cualquier lenguaje que posea un sistema de tipos polimórfico. Este trabajo permite la comunicación en múltiples canales, pero solamente demuestra la solidez del trabajo para el caso de un único canal.

3.2. Cálculo de Procesos

A los efectos de acotar el alcance de este trabajo, que implica la formalización rigurosa en Teoría Constructiva de Tipos del cálculo de Tipos de Sesión, hemos quitado algunas primitivas del cálculo original. Presentamos el sistema a formalizar, un cálculo de procesos sin polaridades que es un fragmento del definido en [6]. Quitamos las primitivas de selección ([Sel]) y bifurcación sobre etiquetas ([Br]), bifurcación condicional ([If]) y recursión ([Var] y [Def]).

Removimos las polaridades del cálculo porque éstas sólo aparecen en tiempo de ejecución, como ya fue mencionado en el capítulo anterior.

Las reglas de tipado asociadas a las primitivas eliminadas no son escenciales para el sistema de tipos. Por ejemplo, las bifurcaciones condicionales tal cual son tratadas en [6], donde se exige que las ramas tengan tipos idénticos, pueden introducirse de forma sencilla al sistema de tipos. Si en cambio queremos permitir que las bifurcaciones no tengan que tener tipos idénticos y permitir subtipos, como en [8], entonces se debe introducir una relación de subtipado (orden parcial sobre los tipos) en el sistema de tipos.

También las primitivas de selección y bifurcación sobre etiquetas se pueden agregar fácilmente en el sistema de tipos presentado, tal cual es expuesto en [8]. De la misma forma que antes para la bifurcación condicional, se vuelve deseable un sistema de tipos con una relación de subtipado, que acepte que dos procesos se comuniquen entre sí, si uno ofrece una bifurcación con más etiquetas que las usadas por el otro proceso.

Otra reducción realizada fue quitar los operadores de las expresiones reduciendo las mismas a variables y constantes, pero éstos pueden ser introducidos sin mayor dificultad.

Finalmente el caso de la recursión es distinto ya que introduce más complejidad al sistema de tipos. Esta simplificación reduce considerablemente la complejidad de la formalización presentada. Un buen abordaje a esta problemática se puede ver en [6] y [8].

En la figura 3.1 definimos el cálculo de procesos simplificado según lo descrito anteriormente.

P	::=	$\texttt{request}\ a(x)\ \texttt{in}\ P$	pedir sesión
	|	$\texttt{accept}\ a(x)\ \texttt{in}\ P$	aceptar sesión
	|	$k![\tilde{e}]; P$	enviar información
	|	$k?(\tilde{x})\ \texttt{in}\ P$	recibir información
	|	$\texttt{throw}\ k[k']; P$	enviar canal
	|	$\texttt{catch}\ k(x)\ \texttt{in}\ P$	recibir canal
	|	$P \mid Q$	componer en paralelo
	|	$\texttt{inact}$	inacción
	|	$(\nu a)P$	ocultar puerto
	|	$(\nu \kappa)P$	ocultar canal
e	::=	$a \mid n \mid \texttt{true} \mid \texttt{false}$	variables de nombres, constantes naturales y booleanas
k	::=	$x \mid \kappa$	variable de canal o constante de canal

Figura 3.1: Sintaxis reducida de los procesos

3.3. Sistema de Tipos

En realidad el sistema de tipos presentado en el capítulo anterior es polimórfico *à la Curry* [2], esto es, existen procesos que admiten varias asignaciones de tipos bajo las reglas de tipado introducidas en 2.8. Ejemplos de procesos polimórficos son:

- $\texttt{accept}\ a(x)\ \texttt{in}\ (x?(y)\ \texttt{in}\ (x![y].\texttt{end}))$
- $\texttt{accept}\ a(x)\ \texttt{in}\ (\texttt{accept}\ b(y)\ \texttt{in}\ (\texttt{throw}\ x[y]; \texttt{inact}))$

En estos procesos se le puede asignar cualquier tipo a la variable y. Esto motiva la introducción de *variables de tipo* que permitan modelar los tipos polimórficos, pasando a tener *esquemas de tipos*.

A diferencia de [6] donde se define que el conjunto de los tipos básicos y de puertos `Sorts` como un único conjunto denotado como *Sorts*, nosotros decidimos dividir a este conjunto en dos. De esta forma cortamos la circularidad que existe en la definición de tipos presentada en 2.6, lo cual nos permite definir dos conjuntos inductivos por separado. Por otro lado perdemos la posibilidad de enviar puertos de comunicación a través de los canales. A pesar de esto, un proceso puede seguir comunicándose con procesos no conocidos, más específicamente a los procesos conocidos por un proceso ya conocido que actúe como una especie de "proxy", que delegue todo lo recibido al proceso original. También puede lograrse esto haciendo que el proceso ya conocido delegue un canal de comunicación con el proceso desconocido, mediante las primitivas `throw` y `catch`, de forma similar al ejemplo presentado en 2.9.

Introducimos así en la figura 3.2 la sintaxis de los tipos definida a partir de la ya presentada en la sección 2.6 con el cambio explicado en el párrafo anterior, quitando los tipos recursivos y agregando a su vez variables de tipo.

Utilizamos dos clases de variables de tipos: variables de tipos básicos $\dot{n}$ y variables de tipos de canales $\dot{a}$. Además también introducimos el esquema de tipos $\hat{a}$ que al aplicarle una sustitución $\dot{a} \mapsto \alpha$ se instancia en $\overline{\alpha}$, o sea, el tipo complementario a α.

Sintaxis			Descripción	Contexto
S	::=	$\texttt{nat} \mid \texttt{bool} \mid \dot{n}$	Tipos básicos	Φ
α	::=	$?[\tilde{S}];\alpha$	Tipos de los canales	Δ
	$\mid$	$![\tilde{S}];\alpha$		
	$\mid$	$?[\alpha];\alpha$		
	$\mid$	$![\alpha];\alpha$		
	$\mid$	$\texttt{end}$		
	$\mid$	$\dot{a}$		
	$\mid$	$\hat{a}$		
T	::=	$\langle \alpha, \overline{\alpha} \rangle$	Tipos de los puertos	Γ

Figura 3.2: Sintaxis de los tipos

A partir de esta definición de tipos actualizamos en 3.1 la función que devuelve el tipo dual o complementario de un tipo dado.

$$\begin{gathered}
\overline{?[\tilde{S}];\alpha} = ![\tilde{S}];\overline{\alpha} \quad \overline{?[\alpha];\beta} = ![\alpha];\overline{\beta} \\
\overline{![\tilde{S}];\alpha} = ?[\tilde{S}];\overline{\alpha} \quad \overline{![\alpha];\beta} = ?[\alpha];\overline{\beta} \\
\overline{\dot{a}} = \hat{a} \qquad\qquad \overline{\hat{a}} = \dot{a} \\
\overline{\texttt{end}} = \texttt{end}
\end{gathered} \tag{3.1}$$

El origen del polimorfismo en el sistema de tipos, más específicamente en los tipos de canal, surge a raíz de la regla [Thr], la cual introducimos en la figura 2.8, y repetimos a continuación.

$$\frac{\Theta;\Gamma \vdash P \triangleright \Delta \cdot k : \beta}{\Theta;\Gamma \vdash \texttt{throw}\ k[k'];P \ \triangleright \Delta \cdot k :![\alpha];\beta \cdot k' : \alpha}\ [\textsc{Thr}]$$

Si leemos esta regla de arriba hacia abajo puede observarse que α es un tipo de canal de comunicación cualquiera, y no hay información sobre éste en las premisas; vemos así el no determinismo inherente en esta regla.

Por otro lado el polimorfismo en los tipos básicos y de puertos se debe a las reglas [Rcv] y [Send] dadas en la figura 2.7. Recordamos a continuación la primera de estas reglas.

$$\frac{\Theta;\Gamma \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta;\Gamma \vdash k?(\tilde{x})\ \texttt{in}\ P \ \triangleright \Delta \cdot k :?[\tilde{S}];\alpha}\ [\textsc{Rcv}]$$

Leyendo nuevamente esta regla de arriba hacia abajo, si el proceso P no brinda información para alguna variable x perteneciente al vector $\tilde{x}$, el contexto Γ no va tener información de tipos para x. En este caso estamos en las hipótesis del *weakening lemma* presentado en 2.7 parte 2, y aplicando este resultado podemos entonces asignar un tipo S cualquiera a la variable x.

$$\frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \qquad \Theta;\Gamma \vdash P \triangleright \Delta \cdot k{:}\alpha}{\Theta;\Gamma \vdash k![\tilde{e}];P \ \triangleright \Delta \cdot k{:}![\tilde{S}];\alpha}\ [\textsc{Send}]$$

De forma similar en la regla [Send] si en el proceso P no se tiene información que permita inferir el tipo para algún nombre n perteneciente al vector $\tilde{e}$, el contexto Γ no va contener información de este nombre. Estamos entonces en las hipótesis del weakening lemma parte 2 nuevamente. Aplicando este lema a la segunda premisa de esta regla, obtenemos que podemos asignar un tipos S' cualquiera a n, o sea tenemos que se satisface $\Theta; \Gamma \cdot n{:}S' \vdash P \triangleright \Delta \cdot k{:}\alpha$. Aplicando ahora un resultado de weakening similar al anterior pero para el juicio de expresiones ahora a la primer premisa obtenemos que $\Gamma \cdot n{:}S' \vdash n \triangleright S$, con lo cual finalmente tenemos que $S = S'$, o sea corroboramos que el juicio de tipado permite la asignación de cualquier tipo básico S' a n.

Debido a la división en dos del conjunto de los tipos básicos y los de puertos decidimos dividir también el correspondiente contexto en los juicios de tipado, subdividiendo así el contexto Γ del juicio ya presentado en 2.8 en los contextos: de variables de tipos básicos Φ y de variables de tipos de puertos Γ. Por otra parte quitamos el contexto de variables de procesos, y consistentemente quitamos al Sistema de Tipos las reglas correspondientes a las primitivas del cálculo quitadas.

Dado que el cálculo de procesos definido no tiene canales polarizados la regla [Cres] desaparece, y la condición $\kappa^+ \notin \Delta \wedge \kappa^- \notin \Delta$ de la regla [Cres'] se vuelve trivial, pudiendo así eliminarla de las premisas del juicio.

Definimos a continuación el sistema de asignación de tipos a procesos resultante.

$$\Gamma \cdot a{:}S \vdash a \triangleright S \ [\text{VarS}\in] \quad \Gamma \vdash 1 \triangleright \texttt{nat} \ [\text{Num}] \quad \Gamma \vdash \texttt{true} \triangleright \texttt{bool} [\text{Boolean}]$$

Figura 3.3: Sistema de asignación de tipos a expresiones

$$\frac{\Delta \text{ sólo contiene tipos } \texttt{end}}{\Phi; \Gamma \vdash \texttt{inact} \triangleright \Delta} \ [\text{Inact}]$$

$$\frac{\Gamma \vdash a \ \triangleright \langle \alpha, \overline{\alpha} \rangle \qquad \Phi; \Gamma \vdash P \triangleright \Delta \cdot x{:}\alpha}{\Phi; \Gamma \vdash \texttt{accept}\ a(x)\ \texttt{in}\ P \triangleright \Delta} \ [\text{Acc}]$$

$$\frac{\Gamma \vdash a \ \triangleright \langle \alpha, \overline{\alpha} \rangle \qquad \Phi; \Gamma \vdash P \triangleright \Delta \cdot x{:}\overline{\alpha}}{\Phi; \Gamma \vdash \texttt{request}\ a(x)\ \texttt{in}\ P \triangleright \Delta} \ [\text{Req}]$$

$$\frac{\Phi \vdash \tilde{e} \triangleright \tilde{S} \qquad \Phi; \Gamma \vdash P \triangleright \Delta \cdot k{:}\alpha}{\Phi; \Gamma \vdash k![\tilde{e}]; P \ \triangleright \Delta \cdot k{:}![\tilde{S}]; \alpha} \ [\text{Send}]$$

$$\frac{\Phi \cdot \tilde{x}{:}\tilde{S}; \Gamma \vdash P \triangleright \Delta \cdot k{:}\alpha}{\Phi; \Gamma \vdash k?(\tilde{x})\ \texttt{in}\ P \ \triangleright \Delta \cdot k{:}?[\tilde{S}]; \alpha} \ [\text{Rcv}]$$

$$\frac{\Phi; \Gamma \vdash P \triangleright \Delta \cdot k{:}\beta}{\Phi; \Gamma \vdash \texttt{throw}\ k[k']; P \ \triangleright \Delta \cdot k{:}![\alpha]; \beta \cdot k'{:}\alpha} \ [\text{Thr}]$$

$$\frac{\Phi; \Gamma \vdash P \triangleright \Delta \cdot k{:}\beta \cdot x{:}\alpha}{\Phi; \Gamma \vdash \texttt{catch}\ k(x)\ \texttt{in}\ P \ \triangleright \Delta \cdot k{:}?[\alpha]; \beta} \ [\text{Cat}]$$

$$\frac{\Phi;\Gamma \vdash P \triangleright \Delta \qquad \Phi;\Gamma \vdash Q \triangleright \Delta'}{\Phi;\Gamma \vdash P \mid Q \triangleright \Delta \otimes \Delta'}\ [\textsc{Conc}]$$

$$\frac{\Phi;\Gamma \cdot a{:}T \vdash P \triangleright \Delta}{\Phi;\Gamma \vdash (\nu a)P \ \triangleright \Delta}\ [\textsc{Nres}]$$

$$\frac{\Phi;\Gamma \vdash P \triangleright \Delta}{\Phi;\Gamma \vdash (\nu \kappa)P \ \triangleright \Delta}\ [\textsc{Cres'}]$$

Figura 3.4: Sistema de asignación de tipos a procesos

3.4. Algoritmo de Inferencia de Tipos

Denotamos mediante $\Gamma(x)$ al tipo asignado a la variable x por el contexto Γ, mientras que $\Gamma\theta$ denota la aplicación de la sustitución θ a todos los tipos que aparecen en Γ; notar que las sustituciones θ aplican tanto a variables de tipos básicos, como a variables de tipos de canales. La expresión $\Gamma <+ x{:}\alpha$ denota sobrescribir el contexto Γ con la asignación del tipo α a la variable x, $\Gamma \backslash x$ denota la eliminación de la información de tipos de la variable x en Γ si es que ésta existe, y $\Gamma_1 \cup \Gamma_2$ representa la unión de los contextos la cual está bien definida sólo en caso de ser disjuntos en su dominio. Esta notación se extiende con el mismo significado a los contextos Φ y Δ.

Presentamos en la figura 3.5 el sistema de inferencia de tipos para las expresiones del cálculo. La expresión $\Phi \vdash e \hookrightarrow S, \Phi'$ denota que para la expresión e se infiere el tipo S bajo el contexto de tipos básicos Φ, y a su vez en Φ' retorna la asignación de variables de tipos hecha si es que e es un nombre que no aparecía en el contexto Φ, tal cual se puede observar en la regla [NameInfV].

$$\Phi \vdash \texttt{true}, \texttt{false} \hookrightarrow \texttt{bool}, \emptyset\ [\textsc{BoolInf}] \qquad \Phi \vdash 1 \hookrightarrow \texttt{nat}, \emptyset\ [\textsc{NatInf}]$$

$$\frac{a \in dom(\Phi)}{\Phi \vdash a \hookrightarrow \Phi(a), \emptyset}\ [\textsc{NameInf}]$$

$$\frac{a \notin dom(\Phi) \qquad \dot{n} \text{ variable de tipo básicos fresca}}{\Phi \vdash a \hookrightarrow \dot{n}, \{a{:}\dot{n}\}}\ [\textsc{NameInfV}]$$

Figura 3.5: Sistema de inferencia de tipos en expresiones

Observar que el sistema de reglas para el tipado de expresiones anterior define correctamente un algoritmo, dado que estas reglas son dirigidas por la sintaxis de las expresiones. Dada una expresión cualquiera queda determinado qué regla usar observando la forma de la expresión. Con lo cual es directa la traducción de estas reglas a cualquier lenguaje funcional con concordancia de patrones (*pattern matching*). Extendemos este algoritmo para listas de expresiones, denotamos la aplicación de esta extensión a un vector de expresiones $\tilde{e}$ y un contexto de tipos básicos Φ de la siguiente forma $\Phi \vdash \tilde{e} \hookrightarrow \tilde{S}, \Phi'$, donde se retorna el vector de

tipos básicos $\tilde{S}$ resultado de aplicar el anterior algoritmo de la figura 3.5 a cada expresión de la lista $\tilde{e}$, donde Φ' almacena las asignaciones de variables frescas hechas. Calculamos Φ' llevando un acumulador ac donde se van agregando las asignaciones de variables frescas hechas al ir aplicando el anterior algoritmo a cada expresión con argumento $\Phi \cup ac$. O sea al inferir el tipo de cada expresión usamos además del contexto Φ la información de asignación de tipos recabada hasta el momento en ac. Finalmente cuando se terminan de recorrer el vector de expresiones, devolvemos este acumulador como resultado Φ'. Notar que este procedimiento asegura que el contexto Φ' no tenga nombres repetidos, o sea, más de una asignación de tipo para algún nombre.

Para la implementación del algoritmo de inferencia para procesos tomamos al contexto Δ como una función total, la cual asigna el tipo de canal `end` para canales no definidos en el contexto. Este comportamiento para el contexto Δ no aparece explícitamente en sistema de tipos dado en [6] y presentado en la sección 2.6. Pero se puede comprobar que se cumple implícitamente en la tercera parte del weakening lemma presentado en la sección 2.7, donde se demuestra que si $\Theta; \Gamma \vdash P \triangleright \Delta \ \wedge \ k \notin \mathrm{dom}(\Delta) \ \Rightarrow \ \Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \mathtt{end}$. El asumir que el contexto Δ tiene información de cualquier canal facilita la implementación del algoritmo de inferencia, no necesitando entonces exigir explícitamente que Δ tenga información del canal k para que la aplicación $\Delta(k)$ esté bien definida.

Utilizamos la expresión $\overset{\theta}{\sqcup} lst$ para denotar que la unificación de una lista de pares de tipos de sesión lst tiene como resultado la sustitución θ. Finalmente $\Gamma_1 \bowtie \Gamma_2$ denota la lista de pares ordenados de tipos resultante de emparejar la información común de puertos de Γ_1 y Γ_2.

Finalmente presentamos en la figura 3.6 el algoritmo de inferencia de tipos para procesos, donde la expresión $\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta$ denota que para el proceso P se infieren los contextos de tipado Φ, Γ y Δ. Consideramos a los contextos como funciones parciales de nombres en tipos, salvo el contexto de canales que tomamos como una función total de canales en tipos de canal como ya se explicó.

$$\frac{}{\emptyset, \emptyset \hookleftarrow \mathtt{inact} \hookrightarrow \emptyset} \ [\textsc{InactInf}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta \qquad a \notin dom(\Gamma)}{\Phi, \Gamma <\!\!+ a{:}\langle \Delta(x), \overline{\Delta(x)} \rangle \hookleftarrow \mathtt{accept}\ a(x)\ \mathtt{in}\ P \hookrightarrow \Delta \backslash x} \ [\textsc{AccInf1}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta \qquad a \in dom(\Gamma) \qquad \overset{\theta}{\sqcup} [(fst(\Gamma(a)), \Delta(x)), (snd(\Gamma(a)), \overline{\Delta(x)})]}{\Phi\theta, \Gamma\theta \hookleftarrow \mathtt{accept}\ a(x)\ \mathtt{in}\ P \hookrightarrow (\Delta\theta) \backslash x} \ [\textsc{AccInf2}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta \qquad a \notin dom(\Gamma)}{\Phi, \Gamma <\!\!+ a{:}\langle \overline{\Delta(x)}, \Delta(x) \rangle \hookleftarrow \mathtt{request}\ a(x)\ \mathtt{in}\ P \hookrightarrow \Delta \backslash x} \ [\textsc{ReqInf1}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta \qquad a \in dom(\Gamma) \qquad \overset{\theta}{\sqcup} [(fst(\Gamma(a)), \overline{\Delta(x)}), (snd(\Gamma(a)), \Delta(x))]}{\Phi\theta, \Gamma\theta \hookleftarrow \mathtt{request}\ a(x)\ \mathtt{in}\ P \hookrightarrow (\Delta\theta) \backslash x} \ [\textsc{ReqInf2}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta \qquad \Phi \vdash \tilde{e} \hookrightarrow \tilde{S}, \Phi'}{\Phi \cup \Phi', \Gamma \hookleftarrow k![\tilde{e}]; P \ \hookrightarrow \Delta <\!\!+ k{:}![\tilde{S}]; \Delta(k)} \ [\textsc{SendInf}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta \qquad \Phi \vdash \tilde{x} \hookrightarrow \tilde{S}, \Phi'}{\Phi \backslash \tilde{x}, \Gamma \hookleftarrow k?(\tilde{x})\ \texttt{in}\ P\ \hookrightarrow \Delta <\!\!+ k{:}?[\tilde{S}]; \Delta(k)}\ [\textsc{RcvInf}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta \qquad \Delta(k') = \texttt{end} \quad k \neq k' \qquad \dot{a}\ \text{variable de tipo de canal fresca}}{\Phi, \Gamma \hookleftarrow \texttt{throw}\ k[k']; P\ \hookrightarrow (\Delta <\!\!+ k{:}![\dot{a}]; \Delta(k)) <\!\!+ k'{:}\dot{a}}\ [\textsc{ThrInf}]$$

$$\frac{k \neq x \qquad \Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta}{\Phi, \Gamma \hookleftarrow \texttt{catch}\ k(x)\ \texttt{in}\ P\ \hookrightarrow (\Delta <\!\!+ k{:}?[\Delta(x)]; \Delta(k)) \backslash x}\ [\textsc{CatInf}]$$

$$\frac{\Phi_1, \Gamma_1 \hookleftarrow P \hookrightarrow \Delta_1 \qquad \Phi_2, \Gamma_2 \hookleftarrow Q \hookrightarrow \Delta_2 \qquad \overset{\theta}{\sqcup} ((\Gamma_1 \ltimes \Gamma_2\pi) \cup (\Phi_1 \ltimes \Phi_2\pi))}{\Phi_1\theta \cup \Phi_2\pi\theta, \Gamma_1\theta \cup \Gamma_2\pi\theta \hookleftarrow P \mid Q \hookrightarrow (\Delta_1\theta) \otimes (\Delta_2\pi\theta)}\ [\textsc{ConcInf}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta}{\Phi, \Gamma \backslash a \hookleftarrow (\nu a)P\ \hookrightarrow \Delta}\ [\textsc{NresInf}]$$

$$\frac{\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta}{\Phi, \Gamma \hookleftarrow (\nu\kappa)P\ \hookrightarrow \Delta}\ [\textsc{Cres'Inf}]$$

Figura 3.6: Algoritmo de inferencia de tipos en procesos

Nuevamente el sistema de reglas introducido ahora para el tipado de procesos es dirigido por la sintaxis de los procesos, a excepción de las parejas de reglas [AccInf1] y [AccInf2] por un lado, y [ReqInf1] y [ReqInf2] por otro. En estos casos sólo observar la forma del proceso no permite saber cuál de las reglas en cada pareja usar. Se debe entonces aplicar el algoritmo recursivamente para luego estudiar si $a \in dom(\Gamma)$ para el contexto Γ devuelto.

A continuación comentamos cada regla. Comenzamos detallando la regla [InactInf], la cual devuelve tres contextos vacíos para un proceso de la forma `inact`, dado que este proceso no contiene nombres o variables a ser tipadas.

En el caso de un proceso de la forma `accept` $a(x)$ `in` P, como mencionamos antes, podemos aplicar tanto la regla [AccInf1] como [AccInf2]. Para decidir qué regla usar debemos entonces aplicar el algoritmo al proceso P, si el resultado es que no hay contextos que tipen este proceso entonces el proceso original tampoco es tipable ya que no se cumplen las premisas de ninguna de las anteriores reglas. Si en cambio sí existen contextos Φ, Δ y Γ que tipen P analizamos si el puerto a aparece en el contexto Γ. Cuando aparece aplicamos la regla [AccInf2], y debemos entonces resolver la tercer premisa de esta regla. Esta pide unificar: la primera componente del par (obtenida mediante la función fst) que conforma el tipo de puerto de a en el contexto Γ con el tipo asignado al canal x en Δ, y también la segunda componente (obtenida mediante la función *snd*) en el tipo de a con el complemento del tipo de x en Δ. Si no existe unificación posible entonces el resultado es que el proceso no es tipable, al fallar la tercer premisa de esta regla. Cuando sí existe una sustitucion θ que unifique las dos parejas de tipos descrita entonces devolvemos los contextos resultantes de aplicar las sustituciones a los contextos Φ, Δ y Γ. Quitamos la información de tipo del canal x en Δ, ya que la primitiva de procesos `accept` liga la variable

x en P. Cuando $a \notin \Gamma$ aplicamos la regla [AccInf1] que consistentemente también quita la información del canal x de Δ, y la asigna a la variable de puerto a en su primera componente, y el complemento en su segunda componente en el contexto Γ. El caso inductivo `request` $a(x)$ `in` P es similar.

Para un proceso $k![\tilde{e}]; P$ utilizamos la regla [SendInf] que pide como premisas que el proceso P sea tipable bajo contextos Φ, Δ y Γ, y con el mismo contexto de tipos básicos Φ como argumento, más el vector de expresiones $\tilde{e}$, invocamos a la extensión del algoritmo de inferencia en expresiones a listas de expresiones. Se devuelve finalmente los contextos $\Phi \cup \Phi'$, Γ y Δ agregándole la asignación al canal k del tipo $![\tilde{S}]; \Delta(k)$. El dominio de Φ' sólo contiene nombres que no pertenecen a Φ; esto es debido a que en la definición del algoritmo de inferencia en expresiones sólo se agregan asignaciones de tipos a nombres a en la regla [NameInfV], y esta regla tiene como premisa que $a \notin dom(\Phi)$. Queda entonces bien definida la unión $\Phi \cup \Phi'$.

En el caso en que el proceso sea de la forma $k?(\tilde{x})$ `in` P aplicamos la regla [RcvInf], que tiene como premisa que el proceso P sea tipable bajo contextos Φ, Γ y Δ, y en este caso en la segunda premisa invocamos al algoritmo de inferencia en expresiones con argumentos Φ y $\tilde{x}$. Retornamos el contexto Γ inalterado, a Φ le quitamos la información de tipado para las variables $\tilde{x}$, y a Δ le agregamos la asignación a k del tipo $?[\tilde{S}]; \Delta(k)$, donde $\tilde{S}$ es el vector de tipos básicos resultante de aplicar el algoritmo de inferencia en expresiones.

Cuando el proceso es de la forma `throw` $k[k']; P$, aplicamos la regla [ThrInf] que pide que los canales k y k' sean distintos, y aplicamos el algoritmo al proceso P. Si existen contextos Φ, Δ y Γ que tipen P entonces comprobamos si el contexto Δ contiene información del canal k'; en caso de tener información devolvemos que el proceso original no es tipable, debido a que el canal k' es utilizado dentro de P. Si no aparece información del canal k' en el contexto Δ devolvemos los contextos Φ y Γ inalterados, mientras que al contexto Δ le asignamos a k el tipo $![\dot{a}]; \Delta(k)$, y a k' el tipo $\dot{a}$. Aquí $\dot{a}$ es una variable de tipo de canal *fresca*, esto es, no aparece en los contextos Δ ni Γ.

Si el proceso es de la forma `catch` $k(x)$ `in` P, aplicamos la regla [CatInf] que exige que k no sea una variable con nombre x; en caso de serlo entonces el proceso no es tipable. Si no son idénticos entonces aplicamos el algoritmo al proceso P tal cual exige la segunda premisa de esta regla. Si P es tipable bajo contextos Φ, Δ y Γ, devolvemos los contextos Φ y Γ sin modificaciones, mientras que al contexto Δ le asignamos al canal k el tipo $?[\Delta(x)]; \Delta(k)$. Finalmente, luego de agregar esta información, le quitamos la asignación de tipos para la variable de canal x, debido a que la variable x es ligada por la primitiva `catch` en P.

En el caso de un proceso de la forma $P \mid Q$ utilizamos la regla [ConcInf] cuyas premisas piden que los procesos que componen la composición en paralelo P y Q sean tipables. Si ambos procesos son tipables bajo contextos Φ_1, Δ_1 y Γ_1 , y Φ_2, Δ_2 y Γ_2 para P y Q respectivamente, entonces la tercer premisa pide unificar el conjunto de tuplas de tipos resultante de agrupar en pares la información de tipos de puertos en común en Γ_1 y Γ_2, y de tipos básicos para nombres en común en Φ_1 y Φ_2. Antes de agrupar esta información renombramos las variables de tipos en Γ_2, Φ_2 y Δ_2 con nombres frescos respecto de Γ_1, Φ_1 y Δ_1. Este renombre lo representamos mediante una sustitución π.

Para un proceso de la forma $(\nu a)P$ utilizamos la regla [NresInf] que pide que el proceso P sea tipable también por este algoritmo bajo ciertos contextos. Se retorna entonces estos mismos contextos, salvo que quitamos el puerto a del

contexto de puertos ya que la primitiva νa liga el puerto en el proceso P

Finalmente cuando el proceso es de la forma $(\nu\kappa)P$ devolvemos directamente el resultado de la llamada recursiva para el proceso P.

3.5. Solidez

Presentamos ahora distintos lemas necesarios para demostrar en 3.5.5 la solidez con respecto al juicio de tipos del algoritmo definido en la sección anterior.

Lema 3.5.1 (Solidez de la inferencia para expresiones)**.** $\Phi \vdash \tilde{e} \hookrightarrow \tilde{S}, \Phi'$ *implica:*

1. $\Phi \cup \Phi' \vdash \tilde{e} \triangleright \tilde{S}$
2. $dom(\Phi') \cap dom(\Phi) = \emptyset$
3. *el contexto* Φ' *no contiene nombres repetidos y sus asignaciones son todas a variables de tipos frescas*
4. $dom(\Phi') \subseteq \tilde{e}$

Demostración. La justificación es directa viendo la definición de este juicio. □

Lema 3.5.2 (Weakening Lemma para el contexto Φ)**.** $\Phi, \Gamma \vdash P \triangleright \Delta$ *y* $n \notin dom(\Phi)$ *implica que* $\Phi \cdot n{:}S, \Gamma \vdash P \triangleright \Delta$.

Demostración. La demostración de este resultado es por inducción en las reglas del juicio $\Phi, \Gamma \vdash P \triangleright \Delta$. Y es similar a la detallada en el Apéndice A. □

Lema 3.5.3 (Weakening Lemma para el contexto Γ)**.** $\Phi, \Gamma \vdash P \triangleright \Delta$ *y* $a \notin dom(\Gamma)$ *implica que* $\Phi, \Gamma \cdot a{:}\langle\alpha, \overline{\alpha}\rangle \vdash P \triangleright \Delta$.

Demostración. Al igual que el anterior caso la demostración de este resultado es por inducción en las reglas del juicio $\Phi, \Gamma \vdash P \triangleright \Delta$, y es similar a la detallada en el Apéndice A. □

Lema 3.5.4 (Conservación bajo sustituciones)**.** $\Phi, \Gamma \vdash P \triangleright \Delta$ *implica* $\Phi\theta, \Gamma\theta \vdash P \triangleright \Delta\theta$.

Demostración. Esta demostración es por inducción en el juicio $\Phi, \Gamma \vdash P \triangleright \Delta$. □

Teorema 3.5.5 (Solidez de la inferencia para procesos)**.** $\Phi, \Gamma \hookleftarrow P \hookrightarrow \Delta$ *implica* $\Phi, \Gamma \vdash P \triangleright \Delta$.

Demostración. Hacemos inducción en la definición inductiva del proceso P.

Caso $P = \texttt{inact}$ Aplicando la definición del algoritmo de inferencia a este caso tenemos que aplica la regla [InactInf], entonces necesariamente Φ, Γ y Δ son vacíos. Es trivial que $\emptyset$ tiene sólo información de tipo `end`, aplicando la regla [Inact] obtenemos la prueba.

Caso $P = \texttt{accept}\ a(x)\ \texttt{in}\ P$ Se puede aplicar tanto la regla [AccInf1] como [AccInf2]. En ambas reglas su primer premisa es idéntica, a partir de ésta sabemos que $\exists\Gamma', \Delta'$ tales que $\Phi, \Gamma' \hookleftarrow P \hookrightarrow \Delta'$. Aplicando la hipótesis inductiva a este resultado obtenemos que $\Phi', \Gamma' \vdash P \triangleright \Delta'$. Se abren ahora dos casos:

1. $a \in dom(\Gamma')$ Aplica entonces la regla [AccInf2], tenemos así que se cumplen las siguientes condiciones:
   a) $(\Phi = \Phi'\theta) \wedge (\Gamma = \Gamma'\theta) \wedge (\Delta = (\Delta'\theta)\backslash x)$

b) Como $\overset{\theta}{\sqcup}\ [(fst(\Gamma'(a)), \Delta'(x)), (snd(\Gamma'(a)), \overline{\Delta'(x)})]$ utilizando la corrección de la unificación tenemos entonces que $fst((\Gamma'\theta)(a)) = (\Delta'\theta)(x) \wedge snd((\Gamma'\theta)(a)) = \overline{(\Delta'\theta)(x)}$, lo que equivale a que $(\Gamma'\theta)(a) = \langle(\Delta'\theta)(x), \overline{(\Delta'\theta)(x)}\rangle$

Aplicamos el lema 3.5.4 a la hipótesis inductiva, obteniendo que $\Phi'\theta, \Gamma'\theta \vdash P \triangleright \Delta'\theta$. Podemos entonces construir el siguiente árbol de derivación en el sistema de reglas de asignación de tipos que demuestra finalmente este caso:

$$\frac{\Gamma'\theta \vdash a \triangleright \overbrace{(\Gamma'\theta)(a)}^{\overset{1b}{=}\langle(\Delta'\theta)(x), \overline{(\Delta'\theta)(x)}\rangle} \qquad \Phi'\theta, \Gamma'\theta \vdash P \triangleright \Delta'\theta}{\underbrace{\Phi'\theta}_{\overset{1a}{=}\Phi}, \underbrace{\Gamma'\theta}_{\overset{1a}{=}\Gamma} \vdash \texttt{accept}\ a(x)\ \texttt{in}\ P \triangleright \underbrace{(\Delta'\theta)\backslash x}_{\overset{1a}{=}\Delta}}\ [\textsc{Acc}]$$

2. $\underline{a \notin dom(\Gamma')}$ Aplica entonces la regla [AccInf1], con lo cual tenemos que $(\Gamma = \Gamma' <\!\!+ a{:}\langle\Delta'(x), \overline{\Delta'(x)}\rangle \wedge (\Delta = \Delta'\backslash x)$. Construimos la siguiente derivación que demuestra este último caso:

$$\frac{\Gamma' <\!\!+ a{:}\langle\Delta'(x), \overline{\Delta'(x)}\rangle \vdash a \triangleright \overbrace{(\Gamma' <\!\!+ a{:}\langle\Delta'(x), \overline{\Delta'(x)}\rangle)(a)}^{=\langle\Delta'(x), \overline{\Delta'(x)}\rangle} \qquad \Phi, \Gamma' \vdash P \triangleright \Delta'}{\Phi, \underbrace{\Gamma' <\!\!+ a{:}\langle\Delta'(x), \overline{\Delta'(x)}\rangle}_{\Gamma} \vdash \texttt{accept}\ a(x)\ \texttt{in}\ P \triangleright \underbrace{\Delta'\backslash x}_{\Delta}}\ [\textsc{Acc}]$$

Caso $\underline{P = \texttt{request}\ a(x)\ \texttt{in}\ P}$ Análogo al caso anterior

Caso $\underline{P = k![\tilde{e}]; P}$ Aplica entonces la regla [SendInf] con lo que tenemos que $\exists \Delta', \Phi', \Phi''$ tal que:

1. $\Phi', \Gamma \leftharpoonup P \hookrightarrow \Delta'$
2. $\Phi' \vdash \tilde{e} \hookrightarrow \tilde{S}, \Phi''$
3. $\Delta = \Delta' <\!\!+ k{:}![\tilde{S}]; \Delta'(k)$ y $\Phi = \Phi' \cup \Phi''$

Aplicando la hipótesis inductiva a 1 tenemos:

$$\Phi', \Gamma \vdash P \triangleright \Delta' \tag{3.2}$$

Aplicando 3.5.1 a 2 tenemos que:

$$\begin{aligned} &\Phi' \cup \Phi'' \vdash \tilde{e} \triangleright \tilde{S} \\ &dom(\Phi') \cap dom(\Phi'') = \emptyset \\ &\Phi'' \text{ no contiene asignaciones repetidas} \end{aligned} \tag{3.3}$$

Utilizamos las dos últimas condiciones del resultado anterior para aplicar tantas veces el lema 3.5.2 a 3.2 como asignaciones tenga el contexto Φ'' para obtener así:

$$\Phi' \cup \Phi'', \Gamma \vdash P \triangleright \Delta' \tag{3.4}$$

Podemos ahora mediante la siguiente derivación construir la demostración final de este caso:

$$\dfrac{\dfrac{}{\Phi' \cup \Phi'' \vdash \tilde{e} \triangleright \tilde{S}}\,3.3 \qquad \dfrac{}{\Phi' \cup \Phi'', \Gamma \vdash P \triangleright \Delta'}\,3.4}{\underbrace{\Phi' \cup \Phi''}_{\stackrel{3}{=}\Phi}, \Gamma \vdash k![\tilde{e}]; P \triangleright \underbrace{\Delta' <\!\!+k:![\tilde{S}]; \Delta'(k)}_{\stackrel{3}{=}\Delta}}\ [\textsc{Send}]$$

Caso $P = k?(\tilde{x})$ `in` P Aplica entonces [RcvInf1] teniendo así que $\exists \Delta', \Phi', \Phi'', \tilde{S}$ tal que:

1. $\Phi', \Gamma \leftarrow P \hookrightarrow \Delta'$
2. $\Phi' \vdash \tilde{x} \hookrightarrow \tilde{S}, \Phi''$
3. $\Delta = \Delta' <\!\!+k:?[\tilde{S})]; \Delta'(k)$ y $\Phi = \Phi' \backslash \tilde{x}$

Aplicamos la hipótesis inductiva a 1 obteniendo así:

$$\Phi', \Gamma \vdash P \triangleright \Delta' \tag{3.5}$$

Notar que no necesariamente Φ' tiene asignaciones de tipado definidas para todo nombre en el vector $\tilde{x}$ tal cual pide la regla[Rcv]. Por lo cual utilizamos 3.5.1 con la premisa 2 para obtener los siguientes resultados:

$$\begin{array}{l} \Phi' \cup \Phi'' \vdash \tilde{x} \triangleright \tilde{S} \\ dom(\Phi') \cap dom(\Phi'') = \emptyset \\ \Phi'' \text{ no contiene nombres repetidos y } dom(\Phi'') \subseteq \tilde{x} \end{array} \tag{3.6}$$

Observando la definición de este juicio de tipado de expresiones se puede ver que la primer afirmación en 3.6 equivale a pedir que $\Phi' \cup \Phi''$ tenga asignaciones de tipado definidas para todo nombre en el vector $\tilde{x}$, quedando entonces bien definida la expresión $(\Phi' \cup \Phi'')(\tilde{x})$.

Aplicando el weakening 3.5.2 tantas veces como asignaciones de tipos que existan en Φ'', con premisas la segunda y tercera condición de en 3.6, y la hipótesis inductiva 3.5 obtenemos que se satisface:

$$\Phi' \cup \Phi'', \Gamma \vdash P \triangleright \Delta' \tag{3.7}$$

Por la tercer premisa en 3.6, donde se establece que $dom(\Phi'') \subseteq \tilde{x}$, junto con que $\Phi = \Phi' \backslash \tilde{x}$ se puede inferir que $(\Phi' \cup \Phi'') \backslash \tilde{x} = \Phi$. Podemos ahora construir la siguiente derivación:

$$\dfrac{\dfrac{}{\Phi' \cup \Phi'', \Gamma \vdash P \triangleright \Delta'}\,3.7}{\underbrace{(\Phi' \cup \Phi'') \backslash \tilde{x}}_{=\Phi}, \Gamma \vdash k?(\tilde{x})\ \texttt{in}\ P \triangleright \underbrace{\Delta' <\!\!+k:?[\tilde{S}]; \Delta'(k)}_{\stackrel{3}{=}\Delta}}\ [\textsc{Rcv}]$$

Caso $P = \texttt{throw}\ k[k']; P$ Aplicamos la regla [ThrInf] teniendo así que $\exists \Delta'$ y $\exists \dot{a}$ variable de tipo fresca tal que:

1. $\Phi, \Gamma \leftarrow P \hookrightarrow \Delta'$
2. $\Delta = (\Delta' <\!\!+k:![\dot{a}]; \Delta'(k)) <\!\!+k':\dot{a}$
3. $\Delta'(k') = \texttt{end}$ y $k \neq k'$

Para que la expresión $\Delta' \cdot k{:}![\dot{a}]; \Delta'(k) \cdot k'{:}\dot{a}$ en la conclusión de la regla de tipado [Thr] esté bien definida debe suceder que $k \neq k'$ y $k \notin dom(\Delta')$ lo cual se cumple por 3.

Aplicando la hipótesis inductiva a 1 tenemos $\Phi, \Gamma \vdash P \triangleright \Delta'$. Podemos ahora construir la siguiente derivación que demuestra este caso:

$$\frac{\Phi, \Gamma \vdash P \triangleright \Delta'}{\Phi, \Gamma \vdash \texttt{throw}\ k[k']; P \triangleright \underbrace{(\Delta' <+ k{:}![\dot{a}]; \Delta'(k)) <+ k'{:}\dot{a}}_{\stackrel{2}{=}\Delta}} [\text{Thr}]$$

Caso $P = \texttt{catch}\ k(x)\ \texttt{in}\ P$ Caso similar al anterior

Caso $P = P \mid Q$ Aplicando la definición del algoritmo $\exists \Phi_1, \Phi_2, \Delta_1, \Delta_2, \Gamma_1, \Gamma_2, \theta, \pi$ tal que:

1. $\Phi_1, \Gamma_1 \hookleftarrow P \hookrightarrow \Delta_1$
2. $\Phi_2, \Gamma_2 \hookleftarrow Q \hookrightarrow \Delta_2$
3. $\overset{\theta}{\sqcup} ((\Gamma_1 \bowtie \Gamma_2\pi) \cup (\Phi_1 \bowtie \Phi_2\pi))$
4. $\Gamma = (\Gamma_1\theta) \cup (\Gamma_2\pi\theta)$, $\Delta = (\Delta_1\theta) \otimes (\Delta_2\pi\theta)$ y $\Phi = \Phi_1\theta \cup \Phi_2\pi\theta$

Aplicando la hipótesis inductivas y luego el lema 3.5.4 a 1 y 2, tenemos respectivamente que:

$$\Phi_1\theta, \Gamma_1\theta \vdash P \triangleright \Delta_1\theta \tag{3.8}$$

$$\Phi_2\pi\theta, \Gamma_2\pi\theta \vdash Q \triangleright \Delta_2\pi\theta \tag{3.9}$$

Utilizando la solidez de la unificación hecha en 3 tenemos que la información de tipado de nombres en común en los contextos $\Gamma_1\theta$ y $\Gamma_2\pi\theta$ por un lado, y $\Phi_1\theta$ y $\Phi_2\pi\theta$ por otro, tiene que coincidir. Así para cada par de información de tipos $x{:}\alpha \in \Gamma_2\pi\theta$, o sea que $\Gamma_2\pi\theta$ brinde la información de tipo α explícita para la variable x, analizamos según la variable x aparezca o no en el contexto $\Gamma_1\theta$:

- Si $\exists \beta, x{:}\beta \in \Gamma_1\theta$: entonces por la solidez de la unificación tenemos que necesariamente $\beta \equiv \alpha$, con lo cual agregar la asignación $x{:}\alpha$ a $\Gamma_1\theta$ no tiene efecto alguno.
- Si $\nexists \beta, x{:}\beta \in \Gamma_1\theta$: entonces podemos aplicar el lema 3.5.3 y agregar esta asignación de información al contexto $\Gamma_1\theta$ preservando el juicio de tipado

Obtenemos así que se satisface:

$$\Phi_1\theta, (\Gamma_1\theta) \cup (\Gamma_2\pi\theta) \vdash P \triangleright \Delta_1\theta$$

Mediante un razonamiento similar, sólo que ahora aplicamos ahora el lema 3.5.2 de weakening para contextos de tipos básicos, tenemos a partir del anterior resultado:

$$(\Phi_1\theta \cup \Phi_2\pi\theta), (\Gamma_1\theta) \cup (\Gamma_2\pi\theta) \vdash P \triangleright \Delta_1\theta \tag{3.10}$$

Análogamente aplicamos el mismo razonamiento a 3.9 para obtener que se satisface:

$$(\Phi_2\pi\theta \cup \Phi_1\theta), (\Gamma_2\pi\theta) \cup (\Gamma_1\theta) \vdash Q \triangleright \Delta_2\pi\theta$$

Como el orden en que aparecen las asignaciones de tipos a nombres en los conextos no tiene relevancia, obtenemos el siguiente resultado al reordenar las asignaciones en los contextos del anterior juicio.

$$(\Phi_1\theta \cup \Phi_2\pi\theta), (\Gamma_1\theta) \cup (\Gamma_2\pi\theta) \vdash Q \triangleright \Delta_2\pi\theta \tag{3.11}$$

Podemos ahora, renombrando tal cual vimos en 4, construir la siguiente derivación que prueba este caso:

$$\frac{\dfrac{3.10}{\Phi, \Gamma \vdash P \triangleright \Delta_1\theta} \quad \dfrac{3.11}{\Phi, \Gamma \vdash Q \triangleright \Delta_2\pi\theta}}{\Phi, \Gamma \vdash P \mid Q \triangleright \underbrace{(\Delta_1\theta) \otimes (\Delta_2\pi\theta)}_{\stackrel{4}{=}\Delta}} [\textsc{Conc}]$$

Caso $P = (\nu a)P$ Aplica en este caso la regla [NresInf] con lo cual $\exists\Gamma'$ tal que:

1. $\Phi, \Gamma' \hookleftarrow P \hookrightarrow \Delta$
2. $\Gamma = \Gamma'\backslash a$

Aplicando la hipótesis inductiva a 1 tenemos que $\Phi, \Gamma' \vdash P \triangleright \Delta$

Podemos entonces construir la siguiente derivación que demuestra este caso:

$$\frac{\Phi, \Gamma' \vdash P \triangleright \Delta}{\Phi, \underbrace{(\Gamma'\backslash a)}_{\stackrel{2}{=}\Gamma} \vdash (\nu a)P \triangleright \Delta} [\textsc{Nres}]$$

Casos $P = (\nu\kappa)P$ Es directo

Dados que estos son todos los casos dados por la definición inductiva de los procesos queda demostrado el teorema □

3.6. Completitud

En esta sección probamos que nuestro algoritmo infiere el esquema de tipos principal (*Principal Type Scheme*) para cualquier programa tipable.

Definición Decimos que un contexto Γ está *incluido ampliamente* en otro Γ' ($\Gamma \subseteq \Gamma'$) si y solo si $\forall a{:}S \in \Gamma$ (cada nombre que Γ brinde información de tipado S), $a{:}S \in \Gamma'$.

Definición Decimos que un contexto Δ de canales está *incluido ampliamente* en otro Δ' ($\Delta \subseteq \Delta'$) si y solo si se cumplen las siguientes dos condiciones:

- $\forall x{:}\alpha \in \Delta, x{:}\alpha \in \Delta'$
- $\forall x$ si $\nexists\alpha, x{:}\alpha \in \Delta$ y $\exists\alpha, x{:}\alpha \in \Delta'$ entonces $\alpha \equiv \texttt{end}$.

Teorema 3.6.1 (Completitud de la inferencia para procesos)**.** *Si se satisface* $\Phi, \Gamma \vdash P \triangleright \Delta$*, entonces* $\exists\Phi', \Gamma', \Delta', \theta$ *tales que* $\Phi', \Gamma' \hookleftarrow P \hookrightarrow \Delta'$*,* $\Phi'\theta \subseteq \Phi$*,* $\Gamma'\theta \subseteq \Gamma$ *y* $\Delta'\theta \subseteq \Delta$*.*

Demostración. Hacemos inducción en la derivación del juicio $\Phi, \Gamma \vdash P \triangleright \Delta$.

Caso [Inact] Como en este caso $P \equiv \texttt{inact}$ podemos aplicar la regla [InactInf], obteniendo así que el algoritmo infiere contextos vacíos. Existe entonces una sustitución θ, por ejemplo vacía, que hace que los contextos vacíos hallados al aplicarles esta sustitución estén incluidos en Φ, Γ y Δ. Notar que $\emptyset \subseteq \Delta$ exige

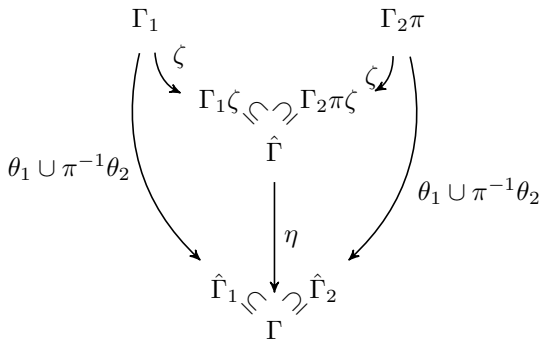
que toda la información de tipos en Δ sean `end`, lo cual se cumple por ser una premisa de regla [Inact] que tenemos por hipótesis.

Caso inductivo [Conc] Entonces $P \equiv P_1 \mid P_2$ y $\exists \Delta', \Delta''$, disjuntos tales que: $\Delta = \Delta' \otimes \Delta''$, $\Phi, \Gamma \vdash P_1 \triangleright \Delta'$ y $\Phi, \Gamma \vdash P_2 \triangleright \Delta''$.

Por hipótesis inductiva tenemos que:

1. $\exists \Phi_1, \Gamma_1, \Delta_1, \theta_1$ tales que $\Phi_1, \Gamma_1 \leftarrow P_1 \hookrightarrow \Delta_1$, $\Phi_1\theta_1 \subseteq \Phi$, $\Gamma_1\theta_1 \subseteq \Gamma$ y $\Delta_1\theta_1 \subseteq \Delta'$
2. $\exists \Phi_2, \Gamma_2, \Delta_2, \theta_2$ tales que $\Phi_2, \Gamma_2 \leftarrow P_2 \hookrightarrow \Delta_2$, $\Phi_2\theta_2 \subseteq \Phi$, $\Gamma_2\theta_2 \subseteq \Gamma$ y $\Delta_2\theta_2 \subseteq \Delta''$

Sea π una sustitución que renombre las variables de tipos que aparecen en los contextos Φ_1, Γ_1 y Δ_1, y que también aparezcan en los contextos Φ_2, Γ_2 y Δ_2. Dado que π es un mero renombre de variables, podemos definir π^{-1} como el renombre inverso, teniendo así que $\pi\pi^{-1}$ es una identidad. Entonces se cumple que las sustituciones θ_1 y $\pi^{-1}\theta_2$ son disjuntas, por lo cual podemos entonces unirlas, lo que denotamos por $\theta_1 \cup \pi^{-1}\theta_2$.



Entonces tenemos que $\theta_1 \cup \pi^{-1}\theta_2$ es un unificador de la información común de los contextos Γ_1 y $\Gamma_2\pi$, y de Φ_1 y $\Phi_2\pi$ también. Dado que existe una instancia común utilizamos la completitud de la operación de unificación ($\sqcup$) para obtener que necesariamente existe entonces la sustitución más general ζ ($\exists \eta, \zeta\eta \equiv (\theta_1 \cup \pi^{-1}\theta_2)$) que unifique estos dos contextos.

Δ' y Δ'' son disjuntos, y como $\Delta_1\theta_1 \subseteq \Delta'$ y $\Delta_2\theta_2 \subseteq \Delta''$, entonces Δ_1 y Δ_2 son disjuntos también. A partir del anterior resultado tenemos entonces que $\Delta_1\zeta$ y $\Delta_2\pi\zeta$ son disjuntos. Queda entonces bien definida la expresión $(\Delta_1\zeta) \otimes (\Delta_2\pi\zeta)$.

Podemos entonces ahora utilizar la regla del algoritmo de inferencia [ConcInf] a las premisas obtenidas anteriormente para demostrar finalmente este caso mediante la siguiente derivación:

$$\frac{\Phi_1, \Gamma_1 \leftarrow P_1 \hookrightarrow \Delta_1 \qquad \Phi_2, \Gamma_2 \leftarrow P_2 \hookrightarrow \Delta_2 \qquad \overset{\zeta}{\sqcup} ((\Gamma_1 \bowtie \Gamma_2\pi) \cup (\Phi_1 \cup \Phi_2\pi))}{\Phi_1\zeta \cup \Phi_2\pi\zeta, \Gamma_1\zeta \cup \Gamma_2\pi\zeta \leftarrow P_1 \mid P_2 \hookrightarrow (\Delta_1\zeta) \otimes (\Delta_2\pi\zeta)}\ [\text{ConcInf}]$$

Vemos ahora que existe la sustitución, la cual es justamente η, que satisface:

1. $(\Phi_1\zeta \cup \Phi_2\pi\zeta)\eta \subseteq \Phi$
2. $(\Gamma_1\zeta \cup \Gamma_2\pi\zeta)\eta \subseteq \Gamma$
3. $((\Delta_1\zeta) \otimes (\Delta_2\pi\zeta))\eta \subseteq \Delta = \Delta'.\Delta''$

Tenemos que $\Gamma_1\theta_1 \subseteq \Gamma$ y $\Gamma_2\pi\pi^{-1}\theta_2 \subseteq \Gamma$, como θ_1 y $\pi^{-1}\theta_2$ son sustituciones disjuntas, se cumple también que $\Gamma_1(\theta_1 \cup \pi^{-1}\theta_1) \subseteq \Gamma$ y $\Gamma_2\pi(\theta_1 \cup \pi^{-1}\theta_1) \subseteq \Gamma$. Como $\zeta\eta \equiv (\theta' \cup \pi^{-1}\theta'')$ tenemos entonces finalmente que $\Gamma_1\zeta\eta \subseteq \Gamma$ y $\Gamma_2\pi\zeta\eta \subseteq \Gamma$, con lo cual tenemos que se cumple 2.

Siguiendo un razonamiento similar tenemos que se satisface la afirmación 1, y la primer condición de la definición de $\subseteq$ para contextos de canales para 3. Ahora vemos que también se satisface la segunda condición de la definición de $\subseteq$. La cual aplica para todo x tal que $\nexists \alpha, x{:}\alpha \in ((\Delta_1\zeta) \otimes (\Delta_2\pi\zeta))\eta$. Como $\otimes$ sólo une los contextos y las sustituciones sólo modifican las segundas coordenadas de los pares, tenemos entonces que $\nexists \alpha, x{:}\alpha \in \Delta_1\theta_1$ y $\nexists \alpha, x{:}\alpha \in \Delta_2\theta_2$. Entonces si $\exists \alpha, x{:}\alpha \in \Delta = \Delta'.\Delta''$ tenemos que $x{:}\alpha \in \Delta'$ o $x{:}\alpha \in \Delta''$. si se cumple la primera parte de la disyunción aplicamos la segunda condición de la definición de $\subseteq$ en $\Delta_1\theta_1 \subseteq \Delta'$ y como tenemos que $x{:}\alpha \notin \Delta_1\theta_1$, entonces necesariamente $\alpha \equiv$ `end`. Análogamente se cumple si se satisface la segunda parte de la disyunción. Quedando entonces demostrado este caso.

Caso inductivo [Acc] Entonces $P =$ `accept` $a(x)$ `in` P y $\Phi, \Gamma \vdash P \triangleright \Delta \cdot x{:}\overline{fst(\Gamma(a))}$.

Por hipótesis inductiva tenemos que $\exists \Phi', \Gamma', \Delta', \theta'$ tales que:

- $\Phi', \Gamma' \hookleftarrow P \hookrightarrow \Delta'$
- $\Phi'\theta' \subseteq \Phi$
- $\Gamma'\theta' \subseteq \Gamma$
- $\Delta'\theta' \subseteq \Delta \cdot x{:}fst(\Gamma(a))$

La prueba se divide según se satisfazca $a \in dom(\Gamma')$:

- $a \notin dom(\Gamma')$ Debemos demostrar que:

$$(\Gamma' <\!\!+ a{:}\langle \Delta'(x), \overline{\Delta'(x)} \rangle)\theta' \subseteq \Gamma \tag{3.12}$$

  Se puede demostrar que vale la propiedad distributiva de la sustitución con respecto al operador $<\!\!+$, y que son intercambiables las operaciones de complemento y sustitución. Otro resultado útil es que $\Gamma(a)\theta' = (\Gamma\theta')(a)$. Finalmente vemos la definición de una sustitución en un tipo de puerto $\langle \alpha, \beta \rangle\theta' = \langle \alpha\theta', \beta\theta' \rangle$. Mediante los anteriores resultados tenemos que el resultado 3.12 es equivalentemente al siguiente::

$$(\Gamma'\theta' <\!\!+ a{:}\langle \Delta'\theta'(x), \overline{\Delta'\theta'(x)} \rangle)\theta' \subseteq \Gamma$$

  Dado que $\Delta'\theta' \subseteq \Delta \cdot x{:}fst(\Gamma(a))$ tenemos entonces que $\Delta'\theta'(x) = fst(\Gamma(a))$, aplicando esto al anterior resultado tenemos:

$$\Gamma'\theta' <\!\!+ a{:}\Gamma(a) \subseteq \Gamma$$

  Lo cual se satisface ya que ya sabíamos que $\Gamma'\theta' \subseteq \Gamma$. Notar también que es directo que $(\Delta' \backslash x)\theta' \subseteq \Delta$ debido a que $\Delta'\theta' \subseteq \Delta \cdot x{:}fst(\Gamma(a))$. Aplicando la regla [AccInf1] a la hipótesis inductiva tenemos entonces el resultado esperado.

- $a \in dom(\Gamma')$
  Necesitamos entonces aplicar ahora la regla [AccInf2], para lo cual debemos probar que existe la sustitución que unifica las parejas $(fst(\Gamma'(a)), \Delta'(x))$ y $(snd(\Gamma'(a)), \overline{\Delta'(x)})$. La demostración se subdivide en los siguientes casos de acuerdo a si Δ' brinda o no información de x:

Como Γ' brinda información de a, o sea, $\exists S, a{:}S \in \Gamma'$, y por otro lado $\Gamma'\theta' \subseteq \Gamma$, entonces tenemos que $(\Gamma'\theta')(a) \equiv \Gamma(a) \equiv S$. Se abren entonces los siguientes casos:

1. Δ' brinda información de x entonces como $\Delta'\theta' \subseteq \Delta \cdot x{:}fst(\Gamma(a))$ se cumple $(\Delta'\theta')(x) \equiv fst(\Gamma(a)) \equiv fst((\Gamma'\theta')(a))$, que equivale a $(\Delta'(x))\theta' \equiv (fst(\Gamma'(a)))\theta'$. Con lo cual existe la sustitución θ' que unifica las expresiones $\Delta'(x)$ y $fst(\Gamma'(a))$. Por tanto también unifica los complementos de las anteriores expresiones, o sea $\overline{\Delta'(x)}$ y $\overline{fst(\Gamma'(a))}$. Con esto vemos que esta sustitución también unifica $\overline{\Delta'(x)}$ y $\overline{snd(\Gamma'(a))}$. Esto es debido a que la regla de tipado [Acc] establece que $\overline{fst(\Gamma(a))} \equiv snd(\Gamma(a))$, como $\overline{(\Gamma'\theta')(a)} \equiv \overline{\Gamma(a)}$ entonces $\overline{fst(\Gamma'(a))} \equiv snd(\Gamma'(a))$, y como θ' unifica $\overline{\Delta'(x)}$ y $\overline{fst(\Gamma'(a))}$ tenemos entonces que también unifica $\overline{\Delta'(x)}$ y $snd(\Gamma'(a))$.
2. Δ' no brinda información de x como $\Delta'\theta' \subseteq \Delta \cdot x{:}fst(\Gamma(a))$, utilizando la segunda condición de la definición de $\subseteq$, se cumple que $(\Delta \cdot x{:}fst(\Gamma(a)))(x) \equiv \texttt{end}$ que reduce a $fst(\Gamma(a)) \equiv \texttt{end}$, como $fst(\Gamma(a)) \equiv fst((\Gamma'\theta')(a))$, se cumple que $fst((\Gamma'\theta')(a)) \equiv \texttt{end}$. Lo que equivale a que $(fst(\Gamma'(a)))\theta' \equiv \texttt{end}$. Como Δ' no brinda información de x se satisface que $\Delta'(x) \equiv \texttt{end}$. Finalmente tenemos entonces que la sustitución θ' unifica las expresiones $\Delta'(x)$ y $fst(\Gamma'(a))$, con lo cual también las expresiones complemento, teniendo así que unifica las expresiones $\overline{\Delta'(x)}$ y $snd(\Gamma'(a))$.

Dado que θ' unifica las parejas $(fst(\Gamma'(a)), \Delta'(x))$ y $(snd(\Gamma'(a)), \overline{\Delta'(x)})$ en todos los casos, entonces existe ζ unificador más general ($\exists \eta, \zeta\eta \equiv \theta'$) de los anteriores pares de expresiones. Podemos entonces aplicar la regla [AccInf]:

$$\frac{\Phi', \Gamma' \hookleftarrow P \hookrightarrow \Delta' \qquad \overset{\zeta}{\sqcup} [(fst(\Gamma'(a)), \Delta'(x)), (snd(\Gamma'(a)), \overline{\Delta'(x)})]}{\Phi'\zeta, \Gamma'\zeta \hookleftarrow \texttt{accept}\ a(x)\ \texttt{in}\ P \hookrightarrow (\Delta'\zeta)\backslash x}\ [\text{AccInf}]$$

Como $\zeta\eta = \theta'$ es directo que se satisfacen las siguientes afirmaciones:

1. $\Phi'\zeta\eta \subseteq \Phi$
2. $\Gamma'\zeta\eta \subseteq \Gamma$
3. $((\Delta'\zeta)\backslash x)\eta \subseteq \Delta$

Quedando así demostrado este caso.

Caso inductivo [Req] Es análogo al anterior

Caso inductivo [Send] Entonces $P = k![\tilde{e}]; P$, y se cumplen los siguientes juicios $\Phi \vdash \tilde{e} \triangleright \tilde{S}$ y $\Phi, \Gamma \vdash P \triangleright \Delta \cdot k{:}![\tilde{S}]; \Delta(k)$.

Por hipótesis inductiva tenemos que $\exists \Phi', \Gamma', \Delta', \theta'$ tales que:

- $\Phi', \Gamma' \hookleftarrow P \hookrightarrow \Delta'$
- $\Phi'\theta' \subseteq \Phi$
- $\Gamma'\theta' \subseteq \Gamma$
- $\Delta'\theta' \subseteq \Delta \cdot k{:}\alpha$.

Dado que $\Phi'\theta' \subseteq \Phi$, podemos ver entonces que $\exists \hat{\Phi}, \Phi = \Phi'\theta \cup \hat{\Phi}$.

Debido a que la inferencia para expresiones es una función total, o sea siempre tiene un resultado exitoso, la segunda premisa de la regla [SendInf] siempre vale. Sean entonces $\tilde{S''}$ y Φ'' los resultados de aplicar la inferencia a expresiones con argumentos Φ' y $\tilde{e}$, esto es, $\Phi' \vdash \tilde{e} \hookrightarrow \tilde{S''}, \Phi''$. Utilizando el resultado de solidez de este algoritmo 3.5.1 tenemos que $\Phi' \cup \Phi'' \vdash \tilde{e} \triangleright \tilde{S''}$, $dom(\Phi') \cap dom(\Phi'') =$

$\emptyset$,Φ'' sólo contiene variables de tipos frescas y $dom(\Phi'') \subseteq \tilde{e}$. Tenemos así todas las premisas necesarias para aplicar la regla [SendInf]:

$$\frac{\Phi', \Gamma' \hookleftarrow P \hookrightarrow \Delta' \qquad \Phi' \vdash \tilde{e} \hookrightarrow \tilde{S''}, \Phi''}{\Phi' \cup \Phi'', \Gamma \hookleftarrow k![\tilde{e}]; P \ \hookrightarrow \Delta' <\!\!+ k:![\tilde{S''}]; \Delta'(k)} \text{ [SendInf]}$$

Por hipótesis tenemos que $\Phi \vdash \tilde{e} \triangleright \tilde{S}$, se puede demostrar entonces a partir de este resultado que los nombres que aparecen en el vector $\tilde{e}$ están contenidos en Φ para que este juicio valga. Dado que $dom(\Phi'') \subseteq \tilde{e}$, tenemos entonces a partir del anterior resultado que $dom(\Phi'') \subseteq dom(\Phi)$. Sabemos que Φ'' sólo contiene variables, con lo cual podemos definir la sustitución más pequeña η tal que $\forall x \in dom(\Phi''), (x \mapsto \Phi(x)) \in \eta$. Tenemos así que $\Phi''\eta \subseteq \Phi$, y además η es disjunta de θ' por cumplirse que $dom(\Phi') \cap dom(\Phi'') = \emptyset$, siendo así válida su unión. Es directo ver entonces que se cumple que $(\Phi' \cup \Phi'')(\theta' \cup \eta) \subseteq \Phi$. De forma similar se puede ver que $\Gamma'\theta' = \Gamma'(\theta' \cup \eta)$, por lo cual $\Gamma'(\theta' \cup \eta) \subseteq \Gamma$. Finalmente observar que esta misma sustitución hace que valga la siguiente igualdad $\tilde{S''}(\theta' \cup \eta) = \tilde{S}$.

Utilizando la propiedad distributiva entre las operaciones: $<\!\!+$ y de sustitución, y la propia definición de la sustitución en un tipo de canal, tenemos que:

$$\begin{aligned}(\Delta' <\!\!+ k:![\tilde{S''}]; \Delta'(k))(\theta' \cup \eta) &= \Delta'(\theta' \cup \eta) <\!\!+ (k:![\tilde{S''}]; \Delta'(k))(\theta' \cup \eta) \\ &= \Delta'(\theta' \cup \eta) <\!\!+ (k:![\tilde{S''}(\theta' \cup \eta)]; \Delta'(k)(\theta' \cup \eta)) \\ &= \Delta'(\theta' \cup \eta) <\!\!+ (k:![\tilde{S}]; \Delta'(k)(\theta' \cup \eta))\end{aligned}$$

Dado que η solo contiene variables frescas, o sea, no ocurren en Δ', $\Delta'\theta' = \Delta'(\theta' \cup \eta)$. Una de las hipótesis inductivas nos dice que $\Delta'\theta' \subseteq \Delta \cdot k::\alpha$, a partir de las anteriores premisas obtenemos que $\Delta'(k)(\theta' \cup \eta) = \alpha$. Así tenemos la anterior expresión equivale a:

$$\Delta'(\theta' \cup \eta) <\!\!+ k:![\tilde{S}]; \alpha$$

Es directo ver que esta expresión está incluida en $\Delta <\!\!+ k:![\tilde{S}]; \alpha$, quedando finalmente demostrado este caso.

Caso inductivo [Rcv] Es similar al caso anterior.

Caso inductivo [Throw] En este caso el proceso es de la forma `throw` $k[k']; P$ y se cumple necesariamente la premisa de esta regla $\Phi, \Gamma \vdash P \triangleright \Delta \cdot x{:}\beta$.

Por hipótesis inductiva tenemos que $\exists \Phi', \Gamma', \Delta', \theta'$ tales que:

- $\Phi', \Gamma' \hookleftarrow P \hookrightarrow \Delta'$
- $\Phi'\theta' \subseteq \Phi$
- $\Gamma'\theta' \subseteq \Gamma$
- $\Delta'\theta' \subseteq \Delta \cdot k{:}\beta$.

Dado que la expresión que ocurre en el juicio $\Delta \cdot k :![\alpha]; \beta \cdot k' : \alpha$ es válida, entonces $k \neq k'$ y $k' \notin dom(\Delta)$, por lo cual aplicando la definición de $\subseteq$ a partir de $\Delta'\theta' \subseteq \Delta \cdot k{:}\beta$ obtenemos que $\Delta'(k') =$ `end`. Sea $\dot{a}$ una variable fresca cualquiera tenemos todas las premisas para aplicar la regla [ThrInf]:

$$\frac{\Phi', \Gamma' \hookleftarrow P \hookrightarrow \Delta' \qquad \Delta(k') = \texttt{end} \quad k \neq k' \qquad \dot{a} \text{ variable de tipo de canal fresca}}{\Phi', \Gamma' \hookleftarrow \texttt{throw}\ k[k']; P\ \hookrightarrow (\Delta' <\!\!+ k{:}![\dot{a}]; \Delta'(k)) <\!\!+ k'{:}\dot{a}} \text{ [ThrInf]}$$

Como la variable $\dot{a}$ es fresca queda bien definida la sustitución $\theta' \cup \{\dot{a} \mapsto \alpha\}$, la cual llamamos η. Siendo directo entonces que $\Phi'\eta \subseteq \Phi$ y $\Gamma'\eta \subseteq \Gamma$.

Dado que por hipótesis inductiva $\Delta'\theta' \subseteq \Delta \cdot k{:}\beta$, tenemos que $(\Delta'\theta')(k) = \beta$. Con lo cual también se cumple que $(\Delta'\eta)(k) = \beta$.

Tenemos ahora aplicando la propiedad distributiva entre el operador $<\!\!+$ y la aplicación de una sustitución, junto con la definición de sustitución, y que $(\Delta'(k))\eta = (\Delta'\eta)(k)$:

$$\begin{aligned}((\Delta' <\!\!+ k{:}![\dot{a}]; \Delta'(k)) <\!\!+ k'{:}![\dot{a}])\eta &= \Delta'\eta <\!\!+ k{:}![\dot{a}\eta]; \Delta'\eta(k) <\!\!+ k'{:}(\dot{a}\eta) \\ &= \Delta'\eta <\!\!+ k{:}![\alpha]; \beta <\!\!+ k'{:}\alpha\end{aligned}$$

Es directo finalmente que esta última expresión está incluida en $\Delta \cdot k{:}![\alpha]; \beta \cdot k'{:}\alpha$, quedando terminada entonces la demostración en este caso.

Caso inductivo [Cat] Es similar al caso anterior.

Casos inductivos [Nres] y [Cres'] Son directos.

□

3.6.1. Corolario de la Completitud

Corolario 3.6.2. *Surge como corolario del anterior resultado que si nuestro algoritmo no encuentra solución alguna, entonces para el proceso argumento no existen contextos bajo los cuales se satisfazca el juicio de tipado.*

Demostración. Si existieran contextos que satisfacen el juicio de tipos para un determinado proceso entonces por el anterior resultado tendrían que existir contextos que nuestro algoritmo lograra inferir, lo cual es absurdo dado que las premisas establecen que esto no sucede. Por absurdo no pueden existir entonces contextos algunos para los cuales se satisfazca el juicio de tipado para el proceso dado. □

Capítulo 4

Teoría Constructiva de Tipos y Agda

Presentamos una breve introducción a la Teoría Constructiva de Tipos de Martin-Löf y el asistente de pruebas Agda, de forma de facilitar la lectura de las siguientes secciones. Para una introducción más completa al tema se puede ver [30, 31, 32]. Definimos algunos tipos, junto a ciertos operadores y propiedades que los caracterizan, que serán usados en el resto de este trabajo. También damos una introducción a la inducción bien fundada, y algunos predicados de accesibilidad propios de Agda.

4.1. Introducción a la Teoría Constructiva de Tipos

La Teoría Constructiva de Tipos de Martin-Löf tiene un tipo básico y dos constructores de tipos. El tipo de todos los conjuntos es el tipo básico. Para cada conjunto S, los elementos de S forman un tipo. Dado un tipo α y una familia β de tipos sobre α, podemos construir el tipo de función de α en β. Escribimos $a{:}\alpha$ para describir que "a es un objeto de tipo α".

Definición Los *conjuntos* son definidos inductivamente, esto es, un conjunto es determinado por las reglas que construyen sus elementos (los constructores del conjunto). Denotamos como **Set** al tipo de los conjuntos.

Definición Los *elementos de un conjunto* S forman un tipo llamado $\mathbf{El}(S)$. Por simplicidad, si a es un elemento del conjunto S, decimos que a tiene tipo S, y entonces escribimos $a{:}S$ en vez de $a{:}\mathbf{El}(S)$.

Definición Una *función dependiente* es una función en la cual el tipo de la salida depende del valor de la entrada. Para formar el tipo de una función dependiente, primero necesitamos un tipo α como dominio y luego una familia de tipos sobre α. Si β es una familia de tipos sobre α, entonces para cada objeto a de tipo α existe un tipo correspondiente $\beta(a)$. Denotamos mediante la expresión $(x{:}\alpha) \to \beta(x)$ el tipo de esta función dependiente.

Si f es una función con tipo $(x{:}\alpha) \to \beta(x)$, entonces aplicando f a un objeto a de tipo α obtenemos un objeto de tipo $\beta(a)$. Denotamos por $f(a)$ a esta aplicación.

Una *función no dependiente* es considerada un caso especial de una función dependiente, donde el tipo β no depende del valor de entrada de tipo α.

Los predicados y relaciones en la Teoría Constructiva de Tipos son vistos como funciones cuya salida son proposiciones. Al igual que los conjuntos, las proposiciones son definidas inductivamente. Esto es, una proposición es determinada por las reglas de construcción de sus pruebas. Si llamaramos **Prop** al tipo de las proposiciones y $\mathbf{Proof}(P)$ al tipo de las pruebas de la proposición P, tendríamos una situación análoga a los conjuntos y sus elementos. Para demostrar una proposición P debemos construir un objeto de tipo $\mathbf{Proof}(P)$. En otras palabras, una proposición es verdadera si podemos construir un objeto de tipo $\mathbf{Proof}(P)$, y es falsa si el tipo $\mathbf{Proof}(P)$ no tiene elementos. Dada la forma en que las proposiciones son introducidas nos permite identificar las proposiciones como conjuntos, con lo que usualmente escribimos **Set** en lugar de **Prop**.

4.2. Introducción a Agda

Agda es el último desarrollo de una serie de implementaciones de la Teoría Constructiva de Tipos que comenzó con ALF [33] en 1990. La versión actual (Agda 2) ha sido implementada por Ulf Norell.

Agda es un asistente de pruebas basado en la Teoría Constructiva de Tipos de Martin-Löf introducida en la sección anterior, y extendido con pattern matching [34]. Es un sistema interactivo para escribir y verificar pruebas, y a su vez, un lenguaje de programación con tipos dependientes. En la Teoría Constructiva de Tipos de Martin-Löf, las demostraciones de los teoremas son representadas como funciones que toman pruebas de las hipótesis y devuelven una prueba de la tesis. Agda asegura que los objetos construidos estén bien formados y bien tipados. Dado que las pruebas son objetos, el chequeo de tipos de objetos asegura así la correctitud de la demostración. Para una introducción más profunda acerca de Agda ver [23, 35, 21, 22].

4.2.1. Ejemplos Básicos

Presentamos a continuación algunos ejemplos básicos de tipos de datos y funciones sobre éstos. Comenzamos definiendo el conjunto de las listas sobre un conjunto genérico `A`.

```
data List (A : Set) : Set where
  []  : List A
  _::_ : (x : A) (xs : List A) → List A
```

En la anterior definición al estar el conjunto `A` a la izquierda de los dos puntos se convierte en un *parámetro* en la definición, convirtiéndose así en un argumento implícito en los constructores. Vemos ahora la definición de la función de largo para una lista formada por elementos de un conjunto `A` cualquiera.

```
length : (A : Set) → List A → ℕ
length A [] = 0
length A (x :: xs) = 1 + (length xs)
```

Notar que el parámetro `A` no tiene utilidad alguna en la implementación de la función, mientras que sí es necesario en la declaración de tipos de esta función. Ésta aprovecha los tipos dependientes para así definir una familia de funciones indexadas en el argumento `A`. Así tenemos una función por cada conjunto posible. Como dijimos este parámetro extra `A` no tiene utilidad en la implementación, por este motivo, Agda define un mecanismo para omitir argumentos de este tipo, siempre y cuando estos puedan ser inferidos durante el chequeo de tipos. Se especifican estos argumentos omitidos mediante el uso de llaves en vez de paréntesis curvos. A continuación reescribimos el anterior ejemplo de forma de aprovechar esta funcionalidad.

```
length : {A : Set} → List A → ℕ
length [] = 0
length (x :: xs) = 1 + (length xs)
```

El siguiente conjunto representa la proposición de igualdad. Su único constructor establece que un elemento es igual a si mismo.

```
data _≡_ {A : Set} (x : A) : A → Set where
   refl : x ≡ x
```

En la anterior definición podemos ver el uso de argumentos implícitos nuevamente, en este caso el conjunto `A`, pero ahora en la definición de un tipo de datos. Ahora presentamos la demostración de propiedad que establece que si dos elementos son iguales, entonces sus imágenes bajo una función cualquiera son iguales también.

```
cong : {A : Set} {B : Set}
       (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

4.2.2. Maybe

El tipo `Maybe` es una familia de tipos indexada en **Set**, la cual para un conjunto `A` cualquiera devuelve el conjunto formado por los elementos de este conjunto, a los que se les yuxtapone el constructor `just`, y el nuevo elemento `nothing`. Este último se utiliza para modelar en un conjunto cualquiera un valor indefinido.

```
data Maybe (A : Set) : Set where
  just    : (x : A) → Maybe A
  nothing : Maybe A
```

4.2.3. Unión disjunta

El tipo ⊎ es una familia de tipos indexada en dos elementos de **Set**, que devuelve el tipo formado por los elementos de estos dos conjuntos. Yuxtaponiendo con el constructor $\texttt{inj}_1$ a los elementos de `A` y con $\texttt{inj}_2$ a los elementos de `B`.

```
data _⊎_ (A : Set) (B : Set) : Set where
  inj₁ : (x : A) → A ⊎ B
  inj₂ : (y : B) → A ⊎ B
```

4.2.4. Predicado Any

Este predicado se utiliza para demostrar que existe al menos un elemento que satisface un predicado en una lista. El constructor `here` establece que la demostración de que `x` satisface la propiedad basta para demostrar esta propiedad en la lista `x :: xs`. Finalmente el constructor `there` determina que una demostración de la propiedad para una lista `xs` es suficiente para demostrar esta propiedad en la lista `x :: xs`.

```
data Any {A} (P : A → Set) : List A → Set where
  here  : ∀ {x xs} (px  : P x)      → Any P (x :: xs)
  there : ∀ {x xs} (pxs : Any P xs) → Any P (x :: xs)
```

A partir de una relación binaria ≈ de equivalencia sobre elementos de tipo `A` cualquiera, podemos definir los siguientes operadores.

```
  _∈_ : A → List A → Set
  x ∈ xs = Any (_≈_ x) xs

  _⊆_ : List A → List A → Set
  xs ⊆ ys = ∀ {x} → x ∈ xs → x ∈ ys
```

Si instanciamos la relación binaria ≈ con la relación ≡ vista anteriormente, entonces la expresión `x ∈ xs` denota la prueba de que el elemento `x` ocurre dentro la lista `xs`. De forma similar ⊆ representa la inclusión de una lista en otra.

4.2.5. Predicado All

El predicado `All` se cumple cuando un predicado se satisface para todos los elementos de una lista. El constructor `[]` establece la condición trivial de que cualquier predicado se cumple para todos los elementos de una lista vacía. Si tenemos que la propiedad se cumple para un elemento `x` y que la propiedad también se cumple para una lista `xs`, entonces el constructor :: construye una prueba de que la propiedad se cumple para la lista `x :: xs`.

```
data All {A} (P : A → Set) : List A → Set where
  []  : All P []
  _::_ : ∀ {x xs} (px : P x) (pxs : All P xs) → All P (x :: xs)
```

El siguiente resultado demuestra la corrección de nuestra interpretación de este tipo, esto es, si tengo un elemento de tipo `All P xs` entonces también tengo una demostración de que para cualquier `x` perteneciente a la lista `xs` se satisface el predicado `P`, mediante la aplicación de este resultado.

```
lookup : ∀ {A} {P : A → Set} {xs} → All P xs →
         (∀ {x} → x ∈ xs → P x)
lookup []         ()
lookup (px :: pxs) (here refl)  = px
lookup (px :: pxs) (there x∈xs) = lookup pxs x∈xs
```

4.2.6. Decidibilidad

Podemos interpretar este predicado como el conjunto de las proposiciones decidibles. Este conjunto tiene dos constructores, dependiendo de si la proposición o su negación puede ser demostrada.

```
data Dec (P : Set) : Set where
  yes : ( p :   P) → Dec P
  no  : (¬p : ¬ P) → Dec P
```

Observar que otra posible forma de definir este conjunto es la siguiente:

```
Dec : Set → Set
Dec = λ p → p ⊎ (¬ p)
```

4.2.7. Predicado Inspect

Este tipo puede ser usado para hacer pattern matching en el resultado `r` de una expresión `e`, y también recordar que `r ≡ e`.

```
data Inspect {A : Set} (x : A) : Set where
  _with-≡_ : (y : A) (eq : y ≡ x) → Inspect x

inspect : {A : Set} (x : A) → Inspect x
inspect x = x with-≡ refl
```

Un ejemplo de uso podría ser el siguiente, donde la expresión `z≡gx` tiene tipo `z ≡ g x`.

```
f x y with inspect (g x)
f x y | z with-≡ z≡gx = ...
```

4.2.8. Inducción Bien Fundada

Se utiliza el predicado de *accesibilidad* generalmente [36, 20] para manejar la recursión general en la Teoría Constructiva de Tipos. Dados un conjunto A,

una relación binaria $<$ en A y un elemento x en A podemos formar el conjunto $Acc\ x$. Este conjunto tiene elementos si y solo si, toda secuencia descendiente de elementos $\ldots < a_2 < a_1 < x$ en A es finita. En este caso decimos que x está en la *parte bien fundada* de $<$ en A. Constructivamente decimos que un elemento x en A es *accesible* si todos los elementos menores que x son *accesibles*, esto se puede ver en la figura 4.1.

```
WfRec : RecStruct A
WfRec P x = ∀ y → y < x → P y

data Acc (x : A) : Set a where
  acc : (rs : WfRec Acc x) → Acc x
```

Figura 4.1: Predicado de accesibilidad

Inducción Bien Fundada en Naturales

Extraemos de la biblioteca de Agda un ejemplo que utiliza esta teoría, formalizado también en Alf en [20]. Éste implementa la división entera por dos para los naturales. Una posible implementación tentativa se puede ver en la figura 4.2. El compilador sabe que $\mathbb{N}$ es bien fundado bajo la relación $<$. Para que no pueda usar esta información utilizamos la función identidad `id` para ocultar que la llamada recursiva es con un elemento más pequeño. De aquí en más siempre ocultaremos la variable n con la aplicación de la identidad para así asegurarnos que el compilador esté efectivamente utilizando la teoría que queremos presentar.

```
halfNoHalt : ℕ → ℕ
halfNoHalt zero                = zero
halfNoHalt (suc zero)          = zero
halfNoHalt (suc (suc n))       = suc (halfNoHalt  (id n))
```

Figura 4.2: Función que calcula la división entera por dos

Utilizamos el predicado de accesibilidad ahora para redefinir esta función de forma que pase la comprobación de terminación de Agda. Para esto vamos a usar que los naturales son todos accesibles bajo la relación `<'` (figura 4.3).

```
data _≤′_ : Rel ℕ zero where
  ≤′-refl : ∀ {n}                    → n ≤′ n
  ≤′-step : ∀ {m n} (m≤′n : m ≤′ n) → m ≤′ suc n

_<′_ : Rel ℕ zero
m <′ n = suc m ≤′ n
```

Figura 4.3: Relación de orden sobre $\mathbb{N}$

Demostramos que los naturales son todos *accesibles* utilizando la función `helper`. Esta función dado un natural n cualquiera devuelve una demostración

de que n es accesible, o sea que $\forall m, m <' n \Rightarrow Acc\ m$. En caso de que n sea cero la demostración es vacía ya que para ningún elemento m es posible demostrar que $m <' 0$. Cuando $n > 0$ lo podemos reescribir entonces como $suc\ n$, y hacer concordancia de patrones sobre la demostración de que $(suc\ m) \leq' (suc\ n)$ (figura 4.3). En caso que la demostración sea $\leq' -refl$, entonces necesariamente $m \equiv n$, con lo cual la demostración de que m es accesible reduce a demostrar que n es accesible, llamando para esto recursivamente a la función `helper` con argumento n, haciendo así una recursión primitiva sobre n. Si en cambio la demostración es de la forma $\leq' -step\ m{<}n$, donde $m{<}n$ identifica una demostración de que $m < n$, entonces ya que `helper` aplicado en n es una demostración de que n es accesible, entonces aplicando esta demostración al elemento m y la demostración de que $m < n$ devuelve una demostración de que m es accesible de tipo $Acc\ m$ tal cual buscamos.

```
<-allAcc : ∀ n → Acc n
<-allAcc n = acc (helper n)
  where
  helper : ∀ n m → m <′ n → Acc m
  helper zero     _ ()
  helper (suc n) .n ≤′-refl       = acc (helper n)
  helper (suc n)  m (≤′-step m<n) = helper n m m<n
```

Figura 4.4: Demostración de que los naturales son todos accesibles

Podemos ahora definir la función $\texttt{half}_2$ que devuelve la división entera por dos. Definiendo para esto la función auxiliar `half-aux`, que recibe como argumento extra la demostración de que n es accesible. Sobre este parámetro adicional hacemos la recursión. En la figura 4.5 vemos como en la llamada recursiva debemos dar una demostración de que el argumento sobre el que hacemos la recursión es efectivamente menor, de forma de obtener así una demostración de que este argumento es también accesible.

```
half-aux : (n : ℕ) → Acc n → ℕ
half-aux zero          _       = zero
half-aux (suc zero)    _       = zero
half-aux (suc (suc n)) (acc f) =
  suc (half-aux (id n) (f (id n) (≤′-step ≤′-refl)))

half₂ : ℕ → ℕ
half₂ n = half-aux n (<-allAcc n)
```

Figura 4.5: Función $\texttt{half}_2$

También podemos redefinir la función $\texttt{half}_2$ utilizando ahora los constructores de recursión bien fundada de Agda, quedando así una definición similar a la que ya presentamos, mostramos esta implementación en la figura 4.6.

Finalmente mostramos una última forma de encarar el problema la cual es construir un predicado de accesibilidad particular para este problema, llamado `DivAcc` (figura 4.7), para luego demostrar que los naturales satisfacen este pre-

```
open WF.All _<′_ <-allAcc public
  renaming ( wfRec-builder to <-rec-builder
           ; wfRec to <-rec
           )

HalfPred : ℕ → Set
HalfPred _ = ℕ

half₂-agda : ℕ → ℕ
half₂-agda = <-rec HalfPred half₂'
  where
  half₂' : ∀ n → WfRec HalfPred n → HalfPred n
  half₂' zero          _   = zero
  half₂' (suc zero)    _   = zero
  half₂' (suc (suc n)) rec = suc (rec (id n) (≤′-step ≤′-refl))
```

Figura 4.6: Función `half₂-agda`

dicado particular de accesibilidad, y así finalmente definir `half₂` por recursión primitiva en la anterior demostración.

```
data DivAcc : (n : ℕ) → Set where
 zero : DivAcc zero
 one  : DivAcc (suc zero)
 sucsuc : (n : ℕ) → DivAcc n → DivAcc (suc (suc n))
```

Figura 4.7: Predicado de accesibilidad particular para la división entera por dos

En la demostración de que todos los naturales cumplen el predicado `DivAcc`, presentado en la figura 4.8, utilizamos el lema que establece que todos los naturales son accesibles demostrado anteriormente.

Otra forma de pensar esta última forma de encarar el problema es ver la familia inductiva de conjuntos que define `DivAcc` como una definición equivalente de los naturales cuya estructura se adapta al algoritmo de división entera por dos. La demostración de que todos los naturales cumplen este predicado de accesibilidad se puede ver entonces como la demostración que los naturales están incluidos en esta familia inductiva. Es trivial ver que la familia de conjuntos está a su vez incluida en los naturales por estar indexada en estos justamente, verificándose de esta forma que existe un mapeo directo entre los naturales y esta familia de conjuntos. Finalmente definimos el algoritmo por recursión primitiva en esta definición equivalente de los naturales (figura 4.9).

Inducción Lexicográfica

Dados distintos conjuntos $A_1, A_2, \ldots, A_n$ sobre los cuales existen distintos ordenes bien fundados $<_1, <_2, \ldots, <_n$ podemos construir un orden bien fundado para el conjunto $A_1 \times A_2 \times \cdots \times A_n$ basándonos en el orden lexicográfico de las n-tuplas estándar que se presenta en (4.1).

```
divacc-aux : (n : ℕ) → <-Acc n → DivAcc n
divacc-aux zero           _             = zero
divacc-aux (suc zero)     _             = one
divacc-aux (suc (suc n)) (acc fsucsucn)
          = sucsuc (id n)
                   (divacc-aux (id n)
                   (fsucsucn (id n)
                             (≤′-step ≤′-refl)))

divacc : (n : ℕ) → DivAcc n
divacc n = divacc-aux n (<-allAcc n)
```

Figura 4.8: Demostración de que todos los naturales cumplen el predicado `DivAcc`

```
half₂-another-aux : (n : ℕ) → DivAcc n → ℕ
half₂-another-aux .zero       zero                  = zero
half₂-another-aux .(suc zero) one                   = zero
half₂-another-aux .(suc (suc n)) (sucsuc n divAccn)
                        = half₂-another-aux (id n) divAccn

half₂-another : (n : ℕ) → ℕ
half₂-another n = half₂-another-aux n (divacc n)
```

Figura 4.9: Implementación de la división entera por dos utilizando el predicado particular de accesibilidad `DivAcc`

$$\langle a_1, a_2, \ldots, a_n\rangle < \langle b_1, b_2, \ldots, b_n\rangle \Leftrightarrow \begin{cases} (a_1 <_1 b_1) \vee \\ (a_1 = b_1 \wedge a_2 <_2 b_2) \vee \\ \ldots \\ (a_1 = b_1 \wedge a_2 = b_2 \wedge \ldots \\ \qquad a_{n-1} = b_{n-1} \wedge a_n <_n b_n) \end{cases} \tag{4.1}$$

Extraemos de la biblioteca de Agda una codificación para la función de *Ackermann* que se basa en inducción lexicográfica. Primero presentamos la definición à la Rózsa Péter de la función de *Ackermann* en la ecuación 4.2.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases} \tag{4.2}$$

No es evidente la terminación de este algoritmo, sin embargo la recursión es bien fundada ya que en cada llamada recursivas o bien m decrece, o m permanece igual y n decrece. Por esto podemos definir esta función utilizando que la recursión primitiva en los naturales es bien fundada, y luego $\mathbb{N} \times \mathbb{N}$ es un orden bien fundado bajo una relación de orden lexicográfico presentada anteriormente.

Podemos ver en la figura 4.10 la codificación en Agda de la función de *Ackermann* utilizando la recursión primitiva de los naturales `N.rec-builder`. Vemos como la función auxiliar `ack` tiene como parámetros auxiliares `ack[1+m]n` y

`ackm•`, donde el primer parámetro es la llamada recursiva cuando m se mantiene constante y n decrece en una unidad, y el segundo parámetro es una función que representa Ackermann cuando m decrece una unidad y n puede ser cualquier natural. Justamente estos dos casos representan todas las tuplas lexicográficamente menores a la tupla $\langle m, n\rangle$ para las cuales es válida la recursión.

```
ackermann : ℕ → ℕ → ℕ
ackermann m n = build [ N.rec-builder ⊗ N.rec-builder ]
                      AckPred ack (m , n)
  where
    AckPred : ℕ × ℕ → Set
    AckPred _ = ℕ

    ack : ∀ p → (N.Rec ⊗ N.Rec) AckPred p → AckPred p
    ack (zero  , n)     _                    = 1 + n
    ack (suc m , zero)  (_          , ackm•) = ackm• 1
    ack (suc m , suc n) (ack[1+m]n , ackm•) = ackm• ack[1+m]n
```

Figura 4.10: Función de Ackermann

Presentamos otra posible posible forma de definir la función de Ackermann en la figura 4.11, en esta utilizamos recursión completa en el argumento n, o sea que los naturales son un orden bien fundado bajo la relación de orden $<$. Así podemos redefinir esta función utilizando ahora la recursión completa `N.<-Rec` en los naturales para el argumento n, quedando entonces una definición similar a la ya vista salvo que en la última regla de la función ahora el primer argumento extra `ack[1+m]•` pasa a ser una función que espera un natural cualquiera y, una demostración de que $y < suc\ n$ para finalmente devolver la función de Ackermann aplicada en la tupla $\langle suc\ m, y\rangle$. En este ejemplo aplicamos `ack[1+m]•` a n y la demostración de que $n < suc\ n$ para así obtener el resultado de Ackerman en la tupla $\langle suc\ m, n\rangle$.

```
ackermann2 : ℕ → ℕ → ℕ
ackermann2 m n = build [ N.rec-builder ⊗ N.<-rec-builder ]
                       AckPred ack (m , n)
  where
    AckPred : ℕ × ℕ → Set
    AckPred _ = ℕ

    ack : ∀ p → (N.Rec ⊗ N.<-Rec) AckPred p → AckPred p
    ack (zero  , n)     _                    = 1 + n
    ack (suc m , zero)  (_          , ackm•) = ackm• 1
    ack (suc m , suc n) (ack[1+m]• , ackm•) = ack[1+m]• n ≤′-refl
```

Figura 4.11: Función de Ackermann utilizando recursión completa en el argumento n

4.2.9. Listas sin elementos repetidos

Para este trabajo definimos el siguente predicado que especifica cuando una lista no contiene elementos repetidos. El constructor `[]` establece que una lista vacía no tiene repetido. Y el constructor :: define una demostración de que una lista `x :: xs` con al menos un elemento no tiene repetidos está formada por: una demostración de que el elemento `x` no pertenezca a `xs`, y que el resto de la lista tampoco tenga repetidos.

```
data NoRep : List A → Set where
  [] : NoRep []
  _::_ : {x : A}{xs : List A}(x∉xs : x ∉ xs) →
        NoRep xs → NoRep (x :: xs)
```

4.2.10. Permutación de Listas

Finalmente damos la definición del predicado que determina si una lista sin elementos repetidos es permutación de otra.

```
permutation : {A : Set} → List A → List A → Set
permutation e1 e2 = (e1 ⊆ e2) × (e2 ⊆ e1)
```

Capítulo 5

Formalización de los Tipos de Sesión

En este capítulo presentamos la formalización en Agda del cálculo de procesos y el sistema de tipos para el Cálculo de Tipos de Sesión. Nos basamos principalmente en [6] presentado en la sección 2. Para la implementación utilizamos la versión 2.2.6 del compilador y la versión 0.3 de la biblioteca estándar de Agda.

5.1. Cálculo de Procesos

Formalizamos un cálculo de procesos con polaridades, el cual es un fragmento del definido en [6]. Quitamos las primitivas de selección ([Sel]) y bifurcación sobre etiquetas ([Br]), bifurcación condicional ([If]) y recursión ([Var] y [Def]).

Se podría incluir la bifurcación condicional de forma sencilla en un sistema de tipos sin subtipado como el que desarrollamos en este trabajo. Si en cambio queremos permitir que las bifurcaciones no tengan que tener tipos idénticos, y habilitar la existencia de subtipos mediante una relación de subtipado (orden parcial sobre los tipos), como en [8], entonces el trabajo se vuelve más complejo al necesitar introducir una relación de subtipado en el sistema de tipos.

Las primitivas de selección y bifurcación sobre etiquetas también se pueden agregar fácilmente en el sistema de tipos presentado, tal cual es expuesto en [8]. De la misma forma que antes para la bifurcación condicional, se vuelve deseable un sistema de tipos con subtipado, que acepte que dos procesos se comuniquen entre sí, si uno ofrece una bifurcación con más etiquetas que las utilizadas por el otro proceso.

Debido a lo explicado anteriormente decidimos recortar estas primitivas al cálculo, sobretodo pensando en reducir el tamaño de la formalización en Agda sin alterar así la esencia del problema objetivo.

Distinto es la recursión que sí introduce más complejidad al sistema de tipos. Un buen abordaje a esta problemática puede encontrarse en [6] y [8], donde se introducen los tipos recursivos mediante la expresión $\mu t.\alpha$, la cual denota que el canal comienza comportándose como especifica α y cuando t es encontrado, repite el comportamiento dictado por α. Esta vez, para disminuir la compleji-

dad del cálculo a formalizar decidimos recortar la recursión del alcance de este trabajo.

Finalmente, para las expresiones se recortaron los operadores, pero éstos pueden ser introducidos fácilmente en la forma usual.

En la figura figura 5.1 se introducen un conjunto de definiciones de tipos inductivos en Agda necesarios para definir el conjunto inductivo de los procesos válidos del cálculo, llamado `Process`.

```
data Polarity : Set where
  pos : Polarity
  neg : Polarity

Name = ℕ
ChannelVar = ℕ
ChannelName = ℕ

data ChannelVar&Value : Set where
  var : ChannelVar → ChannelVar&Value
  ch : ChannelName → Polarity → ChannelVar&Value

data Expression : Set where
  num : ℕ → Expression
  bool : Bool → Expression
  name : Name → Expression

data Process : Set where
  inact : Process
  pipe  : Process → Process → Process
  request : Name → ChannelVar → Process → Process
  accept  : Name → ChannelVar → Process → Process
  catch : ChannelVar&Value → ChannelVar → Process →
          Process
  throw : ChannelVar&Value → ChannelVar&Value → Process →
          Process
  receive : ChannelVar&Value → List Name → Process →
            Process
  send    : ChannelVar&Value → List Expression → Process →
            Process
  vc : ChannelName → Process → Process
  vn : Name → Process → Process
```

Figura 5.1: Procesos

Si bien definimos el cálculo de procesos con polaridades, a la hora de implementar el algoritmo de inferencia de tipos no tendremos en cuenta las polaridades de los canales en el proceso a ser tipado. Esto es debido a que, como se menciona en [7], las polaridades de los canales son definidas únicamente por su utilidad técnica en la demostración de *subject reduction* [6], y no necesita ser expuesta al programador, ni en la semántica operacional, ni en el sistema de tipos. Nosotros decidimos dejar las polaridades en el cálculo de procesos de forma de facilitar posibles trabajos a futuro, como la formalización en Agda del resultado de *subject reduction*.

En la siguiente figura presentamos la codificación en Agda del proceso de la figura 2.9 con las primitivas definidas en esta sección.

```
(pipe
     (request 1 2
              (catch (var 2) 3
                     (send (var 3) ((num 3) ::  (bool true) :: [])
                           inact)))
     (accept  1 4 (pipe
                       (accept 5 6
                               (throw (var 4) (var 6)
                                      inact))
                       (request 5 7
                               (receive (var 7) (8 :: 9 :: [])
                                        inact)))))
```

Figura 5.2: Codificación en Agda del proceso que delega un canal presentado en la figura 2.9

5.2. Sistema de Tipos

Presentamos ahora la codificación del sistema de tipos tal cual introducimos en la sección 3.3. Comenzamos por el conjunto de los tipos básicos $\texttt{Sort} = \{\texttt{nat}, \texttt{boolean}, \texttt{varS}\ n\}$.

```
data Sort : Set where
  nat : Sort
  boolean : Sort
  varS : ℕ → Sort
```

Figura 5.3: Tipos básicos

Definimos ahora el conjunto inductivo `SType` que representa el tipo de los canales de comunicación.

```
data SType : Set where
  receiveS : List Sort → SType → SType
  sendS    : List Sort → SType → SType
  receiveST : SType → SType → SType
  sendST    : SType → SType → SType
  end : SType
  varST : ℕ → SType
  varSTCompl : ℕ → SType
```

Figura 5.4: Tipo de los canales de comunicación

Sobre el conjunto `SType` definimos la función que devuelve el tipo complementario al dado como argumento, esto es el tipo que debe tener un canal para

comunicarse con un canal con el tipo dado como argumento.

```
complST : SType → SType
complST (receiveS ss st) = sendS ss (complST st)
complST (sendS ss st) = receiveS ss (complST st)
complST (receiveST st1 st2) = sendST st1 (complST st2)
complST (sendST st1 st2) = receiveST st1 (complST st2)
complST end = end
complST (varST n) = varSTCompl n
complST (varSTCompl n) = varST n
```

Figura 5.5: Tipo complemento de un canal

Finalmente el conjunto `Type` define el tipo de los puertos de comunicación (figura 5.6). Para este tipo se implementan las funciones selectoras `fstPar` y `sndPar`, que devuelven el primero y segundo argumento respectivamente del único constructor `par` de elementos en este conjunto.

```
data Type : Set where
  par : SType → SType → Type
```

Figura 5.6: Tipo de los puertos de comunicación

Definimos en la figura 5.7 los contextos de tipos como listas de pares ordenados. De aquí en adelante se denota en general al contexto de los tipos básicos `Sorts` mediante la letra griega Φ, a los tipos de canales `STypes` con Δ, y los puertos `Types` con Γ.

```
Sorts  = List (Name × Sort)
STypes = List (ChannelVar&Value × SType)
Types  = List (Name × Type)
```

Figura 5.7: Contextos de tipos

Necesitamos que la relación de igualdad sobre la primera coordenada de los pares ordenados sea decidible y demostrable para luego poder formalizar las búsquedas por la primera coordenada en listas de pares. Entonces para los canales introducimos la función infija $\stackrel{?}{=}_c$, la cual dados dos nombres de canales devuelve si son iguales o no, junto con una demostración de este hecho. Al ser esta una función total se demuestra la decidibilidad de la relación de igualdad sobre los canales, por esto el conjunto de los canales pasa a ser un *setoide decidible*. En Agda esto se establece mediante la definición del tipo `decSetoidC`, se puede encontrar una introducción al predicado `Decidable` de la biblioteca estándar de Agda en la sección 4.2.6.

Finalmente notar que el conjunto `Name` definido en la figura 5.1, por ser un alias de $\mathbb{N}$ hereda la decidibilidad de la relación de igualdad de la biblioteca de $\mathbb{N}$ en Agda.

Al igual que para los procesos se definen funciones que deciden la relación

```
 _≟c_ : Decidable {A = ChannelVar&Value} _≡_

decSetoidC : DecSetoid _ _
decSetoidC = record
  { _≈_               = _≡_
  ; isDecEquivalence = record
    { isEquivalence = PropEq.isEquivalence
    ; _≟_           = _≟c_
    }
  }
```

Figura 5.8: Decidibilidad de la igualdad sobre canales

de igualdad para `Sort` y `SType`. Utilizaremos estas funciones en los algoritmos de unificación para tipos básicos y de canal respectivamente.

```
  _≟S_ : Decidable {A = Sort} _≡_
  _≟ST_ : Decidable {A = SType} _≡_
```

Figura 5.9: Relación de igualdad para `Sort` y `SType`

5.3. Sustituciones de Variables de Tipo

En esta sección definimos las distintas funciones necesarias para instanciar las variables de tipo.

Modelamos una sustitución para una variable como un par donde la primer componente es el nombre de la variable, y la segunda componente es el tipo por el cual ésta es sustituida. Como se puede apreciar en las figuras 5.3 y 5.4 identificamos las variables mediante números naturales.

A continuación definimos las sustituciones de una variable de tipos básicos en tipos básicos (`subSS`), de una variable de tipos básicos en tipos de canal (`subSST`), y de una variable de tipos de canal en tipos de canal (`subSTST`).

```
subSS : ℕ × Sort → Sort → Sort
subSS (n , s) (varS m) with n ≟ m
subSS (n , s) (varS .n) | (yes refl) = s
subSS (n , s) (varS m)  | (no  n≠m)  = varS m
subSS (n , s) nat = nat
subSS (n , s) boolean = boolean

subSST : ℕ × Sort → SType → SType
subSST s (receiveS ss t) = receiveS (map (subSS s) ss)
                                    (subSST s t)
subSST s (sendS ss t)    = sendS    (map (subSS s) ss)
                                    (subSST s t)
subSST s (receiveST t1 t2) = receiveST  (subSST s t1) (subSST s t2)
```

```
subSST s (sendST t1 t2)    = sendST     (subSST s t1) (subSST s t2)
subSST s end = end
subSST s (varST m)      = varST m
subSST s (varSTCompl m) = varSTCompl m

subSTST : ℕ × SType → SType → SType
subSTST s (receiveS ss t) = receiveS ss (subSTST s t)
subSTST s (sendS ss t)    = sendS    ss (subSTST s t)
subSTST s (receiveST t1 t2) = receiveST (subSTST s t1)
                                        (subSTST s t2)
subSTST s (sendST t1 t2)    = sendST    (subSTST s t1)
                                        (subSTST s t2)
subSTST (n , s) (varST m) with n ≟ m
subSTST (n , s) (varST .n) | yes refl = s
subSTST (n , s) (varST m)  | no n≢m   = varST m
subSTST (n , s) (varSTCompl m) with n ≟ m
subSTST (n , s) (varSTCompl .n) | yes refl = complST s
subSTST (n , s) (varSTCompl m)  | no n≢m   = varSTCompl m
subSTST s end = end
```

A continuación implementamos las sustituciones como listas de sustituciones de una variable, o sea listas de pares. En la siguiente figura definimos varias funciones que aplican a sustituciones, de las más importantes son las que permiten aplicar una sustitución a las tres clases de contextos. Usamos la función `map×` definida en la biblioteca de Agda, que devuelve una función que aplica dos funciones argumentos a un par argumento, la primer función a la primer componente y la segunda función a la segunda componente del par respectivamente. Las funciones `foldr` y `map` tienen la definición usual y están definidas en la biblioteca de Agda.

```
lsubS : List (ℕ × Sort) → Sort → Sort
lsubS ls s = foldr subSS s ls

subSLS : ℕ × Sort → List (Sort × Sort) → List (Sort × Sort)
subSLS u ss = map (λ ps → (subSS u (proj₁ ps) ,
                           subSS u (proj₂ ps)))
              ss

lsubSSS : List (ℕ × Sort) → Sorts → Sorts
lsubSSS ls ss = map (map× id (lsubS ls)) ss

lsubSST : List (ℕ × Sort) → SType → SType
lsubSST ss st = foldr subSST st ss

lsubSLPCST : List (ℕ × Sort) → STypes → STypes
lsubSLPCST ss sts = map (map× id (lsubSST ss)) sts

lsubSSTPCST : List (ℕ × SType) → SType → SType
lsubSSTPCST sst st = foldr subSTST st sst

lsubSTLPCST : List (ℕ × SType) → STypes → STypes
lsubSTLPCST sst sts = map (map× id (lsubSSTPCST sst))
                          sts
```

```
subST : List (ℕ × Sort) → Type → Type
subST ss (par t1 t2) = par (lsubSST ss t1)
                           (lsubSST ss t2)
```

```
lsubSLPNT :  List (ℕ × Sort) → Types → Types
lsubSLPNT ss ts = map (map× id (subST ss)) ts

subSTT : List (ℕ × SType) → Type → Type
subSTT ss (par t1 t2) = par (lsubSSTPCST ss t1)
                            (lsubSSTPCST ss t2)

lsubSTLPNT :  List (ℕ × SType) → Types → Types
lsubSTLPNT sst ts = map (map× id (subSTT sst)) ts
```

Figura 5.10: Funciones de sustitución auxiliares

5.4. Contextos

A continuación introducimos la codificación de los contextos en Agda. Para el manejo de los contextos se implementaron distintos módulos, los cuales proveen varias funcionalidades, entre estas están los siguientes operadores que son utilizados en la codificación del juicio de tipos presentado en la próxima sección:

- $n \in_1 L$: es el tipo de las demostraciones de que existe algún t tal que $\langle n, t\rangle \in L$, donde n es un elemento y L una lista de pares. Las primeras proyecciones de la lista L son elementos del mismo tipo que n.
- $L <+ \langle n, t\rangle$: dadas una lista de pares L y un par $\langle n, t\rangle$ agrega el par a la lista, si existe t_2 tal que $\langle n, t_2\rangle \in L$ entonces antes quita la primera ocurrencia (de izquierda a derecha) de un par $\langle n, t_3\rangle$ que aparezca en L para cualquier t_3
- $L \bullet n$: función que dados un elemento n y una lista de pares L, si existe t tal que $\langle n, t\rangle \in L$ devuelve t, si no existe devuelve un elemento por defecto que es elegido al instanciar el módulo
- $L \bullet\!\in d$: es una función que dada una lista L y una demostración d de que existe t tal que $\langle n, t\rangle \in L$, devuelve t
- $L \setminus\setminus n$: dada una lista de pares L y un elemento n remueve de la lista la primera ocurrencia de un par $\langle n, t\rangle$ para cualquier t
- $L \setminus\setminus [] \tilde{n}$: dada una lista de pares L y una lista de elementos $\tilde{n}$ de igual tipo que la primeras proyecciones de la lista L, para cada elemento $n \in \tilde{n}$ aplica la función $\setminus\setminus$ definida antes, para remover este elemento de a la lista resultado que inicialmente es igual a L
- $L_1 \cap L_2$: dadas dos listas de pares L_1 y L_2 devuelve la lista de pares perteneciente al conjunto $\{\langle n, t_1\rangle : \langle n, t_1\rangle \in L_1 \wedge (\exists t_2)(\langle n, t_2\rangle \in L_2)\}$ preservando el orden relativo que tienen en L_1

Uno de los principales módulos para el manejo de los contextos es `Environment`, a continuación mostramos la definición de éste módulo de forma de ejemplificar como se generalizó el manejo del los contextos, y así reutilizar la implementación para los tres tipos de contextos usados.

```
module Environment {K : Set}{D : Set}
                   (_≟_ : Decidable (_≡_ {A = K}); d : D) where
```

La definición anterior define el módulo `Environment` para el manejo de lista de pares genéricas parametrizadas por:

- `K`, el tipo de las claves sobre la que se hace las búsquedas, que corresponde a la primera proyección de los pares que conforman los contextos
- `D`, el tipo de los datos almacenados, correspondiente a la segunda proyección de los pares de los contextos
- $\stackrel{?}{=}$ es la función que demuestra la decidibilidad de la relación de igualdad sobre el conjunto de las claves
- `d`, un elemento particular de tipo `D`, el cual es devuelto por defecto por la operación $\bullet$ en caso de que la clave buscada no exista en la lista de pares

En la figura 5.11 se muestran las instanciaciones para cada contexto usadas más adelante en la definición del sistema de tipos. Aquí vemos como la operación $\_ \bullet c \_$ aplicada a un contexto de tipos de canal devuelve por defecto `end` en el caso de que el contexto no provea información para un canal. También apreciamos como los nombres por defecto de las distintas funcionalidades dentro del módulo genéricos son renombradas para así no superponer los nombres para las distintas instanciaciones de los contextos.

```
import Environment
private open module EST = Environment _≟c_ end using (permutation)
             renaming (_•_ to _•c_ ;
                       _\\_ to _\\c_ ;
                       _<+_ to _<+c_;
                       permutation-refl to permutation-refl-st)
private open module ES = Environment _≟N_ nat using ()
             renaming (_\\_ to _\\s_ ;
                       _\\[]_ to _\\[]s_ ;
                       permutation-refl to permutation-refl-s ;
                       _<+_ to _<+s_)
private open module ET  = Environment _≟N_ (par end end) using ()
            renaming (_•p_ to _•pn_ ;
                      _•∈_ to _•∈n_ ;
                      _<+_ to _<+n_ ;
                      _\\_ to _\\n_ ;
                      permutation-refl to permutation-refl-t ;
                      permutation-++ to permutation-++-t)
```

Figura 5.11: Instanciaciones del módulo Environment

La generalización de los módulos realizada no solo permite la reutilización de funciones en la implementación, sino que también permite reutilizar las distintas

demostraciones hechas sobre propiedades de los contextos. Vemos un ejemplo de esto en el lema `permutation-++` (figura 5.12), el cual prueba que el orden en el cual se concatenan contextos devuelve permutaciones de los mismos pares. En la sección 4.2.10 introdujimos la formalización del concepto de permutaciones de listas.

```
permutation-++ : {A : Set}{e1 e2 : List A} →
                 permutation (e1 ++ e2) (e2 ++ e1)
```

Figura 5.12: Lema de permutación de la concatencación de listas

5.5. Sistema de Tipos

Definimos en la figura 5.13 la codificación del juicio de tipado para expresiones y también su extensión a listas de expresiones, tal cual fueron introducidos en 3.3.

```
data _⊢_▷_ : Sorts → Expression → Sort → Set where
  Num : (Φ : Sorts)(n : ℕ) → Φ ⊢ (num  n) ▷ nat
  Boolean : (Φ : Sorts)(b : Bool) → Φ ⊢ (bool  b) ▷ boolean
  VarS∈ : (Φ : Sorts)(n : Name)(n∈Φ : n ∈₁s Φ) →
          Φ ⊢ (name n) ▷ (Φ •∈s n∈Φ)

  _⊢_▶_ : (Φ : Sorts)(le : List Expression)(ls : List Sort) → Set
  Φ ⊢ le ▶ ls = All  (λ p → Φ ⊢ proj₁ p ▷ proj₂ p) (zip le ls)
```

Figura 5.13: Juicio de tipos para las expresiones

Para representar el juicio de tipado para los procesos, introducido en la sección 3.3, se define el tipo inductivo de la figura 5.14.

```
  data _⊢_▷_ : Sorts × Types → Process → STypes → Set where
  ...
```

Figura 5.14: Sistema de asignación de tipos a procesos

En la próximas secciones damos la formalización de las distintas reglas del sistema de tipos, comentando la reescritura de las reglas de forma de hacer explícitos los constructores, y así facilitar la demostración de corrección del algoritmo de inferencia.

Inact

La regla [Inact] tal cual presentamos en la figura 3.4 tiene como premisa que todos los tipos del contexto Δ sean `end`. Codificamos en Agda esta propiedad mediante el tipo predefinido `All` de la biblioteca de Agda introducido en la

sección 4.2.5. Además utilizamos el predicado `NoRep` (sección 4.2.9) instanciado para cada uno de los contextos como precondición en esta regla. Este predicado sirve para afirmar que no haya pares con el mismo primer componente repetido en una lista. Exigimos esta última condición en todos los contextos para poder introducir de forma segura la regla [PERMUTATION-ENV] más adelante.

```
Inact : (Δ : STypes)(Γ : Types)(Φ : Sorts) →
        NRLst.NoRep Δ → NRLt.NoRep Γ → NRLs.NoRep Φ →
        All (λ p → proj₂ p ≡ end) Δ →
        (Φ , Γ) ⊢ inact ▷ Δ
```

Accept

Recordamos la regla ya presentada.

$$\frac{\Gamma \vdash a \ \triangleright \langle \alpha, \overline{\alpha} \rangle \qquad \Phi; \Gamma \vdash P \triangleright \Delta \cdot x{:}\alpha}{\Phi; \Gamma \vdash \texttt{accept}\ a(x)\ \texttt{in}\ P \triangleright \Delta} \ [\textsc{Acc}]$$

Reescribimos en la regla anterior $\Delta \cdot x{:}\alpha$ por Δ', con lo cual $\Delta \equiv \Delta' \backslash\ x$, y como $\Delta' \bullet x \equiv \alpha$ podemos también intercambiar α por $\Delta' \bullet x$.

$$\frac{\Gamma \vdash a \ \triangleright \langle \Delta' \bullet x, \overline{\Delta' \bullet x} \rangle \qquad \Phi; \Gamma \vdash P \triangleright \Delta'}{\Phi; \Gamma \vdash \texttt{accept}\ a(x)\ \texttt{in}\ P \triangleright \Delta' \backslash\ x} \ [\textsc{Acc}]$$

Formalizamos el juicio $\Gamma \vdash a \ \triangleright \langle \Delta' \bullet x, \overline{\Delta' \bullet x} \rangle$ exigiendo que el par $\langle a, \langle \Delta' \bullet x, \overline{\Delta' \bullet x} \rangle \rangle$ pertenezca a la lista de pares Γ. Codificamos esto al pedir una prueba de que existe t tal que $\langle n, t \rangle \in \Gamma$ de tipo $\texttt{n} \in_1 \texttt{n}\ \Gamma$, tal que se cumplan las siguientes afirmaciones: $\texttt{fstPar}(t) \equiv \Delta' \bullet x$ y $\texttt{sndPar}(t) \equiv \overline{\Delta' \bullet x}$.

```
Accept : (Δ : STypes)(Γ : Types)(Φ : Sorts)(p : Process)
         (c : ChannelVar)(n : Name)
         (n∈Γ : n ∈₁n Γ) →
         fstPar (Γ •∈n n∈Γ) ≡ (Δ •c (var c)) →
         sndPar (Γ •∈n n∈Γ) ≡ complST (Δ •c (var c)) →
         (Φ , Γ) ⊢ p ▷ Δ →
         (Φ , Γ) ⊢ (accept n c p) ▷ (Δ \\c (var c))
```

Tal cual explicamos en 3.4 el contexto Δ se utiliza de forma que si para algún canal k este no tiene información de tipo, o sea no existe t tal que $\langle k, t \rangle \in \Delta$, entonces se asume que el tipo de k es `end`. Esto se logra en la implementación al instanciar el módulo `Environment` para los tipos de canal con `end` entre sus parámetros (figura 5.11). De esta forma sabemos que siempre está bien definida la aplicación Δ `•c k` para cualquier canal `k` y contexto Δ.

Request

Esta regla es análoga a la anterior.

```
Request : (Δ : STypes)(Γ : Types)(Φ : Sorts)(p : Process)
          (c : ChannelVar)(n : Name) →
          (n∈Γ : n ∈₁n Γ) →
          sndPar (Γ •∈n n∈Γ) ≡ ((Δ •c (var c))) →
          fstPar (Γ •∈n n∈Γ) ≡ complST (Δ •c (var c)) →
          (Φ , Γ) ⊢ p ▷ Δ →
          (Φ , Γ) ⊢ (request n c p) ▷ (Δ  \\c (var c))
```

Send

Recordamos la definición presentada para esta regla.

$$\frac{\Phi \vdash \tilde{e} \triangleright \tilde{S} \qquad \Phi; \Gamma \vdash P \triangleright \Delta \cdot k{:}\alpha}{\Phi; \Gamma \vdash k![\tilde{e}]; P \ \triangleright \Delta \cdot k{:}![\tilde{S}]; \alpha} \ [\textsc{Send}]$$

Reescribmos [Send] renombrando $\Delta \cdot k{:}\alpha$ por Δ', con lo cual $\Delta \equiv \Delta' \backslash\ k$ y $\Delta' \bullet k \equiv \alpha$, pudiendo entonces también intercambiar α por $\Delta' \bullet k$. Además como la operación $<+$ quita del contexto el canal a ser agregado, la expresión $\Delta' \backslash\backslash\ k \cdot k{:}![\tilde{S}]; \Delta' \bullet k$ equivale a $\Delta' <+ k{:}![\tilde{S}]; \Delta' \bullet k$.

$$\frac{\Phi \vdash \tilde{e} \triangleright \tilde{S} \qquad \Phi; \Gamma \vdash P \triangleright \Delta'}{\Phi; \Gamma \vdash k![\tilde{e}]; P \ \triangleright \Delta' <+ k{:}![\tilde{S}]; \Delta' \bullet k} \ [\textsc{Send}]$$

```
Send : (Δ : STypes)(Γ : Types)(Φ : Sorts)(p : Process)
       (k : ChannelVar&Value)
       (le : List Expression)(ls : List Sort) →
       Φ ⊢ le ▶ ls →
       (Φ , Γ) ⊢ p ▷ Δ →
       (Φ , Γ) ⊢ send k le p ▷ (Δ <+c (k , sendS ls (Δ •c k)))
```

A partir de la reescritura la codificación de esta regla es directa. En la primer premisa de esta regla utilizamos la codificación del juicio de tipado para listas de expresiones visto en la figura 5.13.

Receive

Recordamos la definición ya presentada para esta regla.

$$\frac{\Phi \cdot \tilde{x}{:}\tilde{S}; \Gamma \vdash P \triangleright \Delta \cdot k{:}\alpha}{\Phi; \Gamma \vdash k?(\tilde{x}) \ \texttt{in}\ P \ \triangleright \Delta \cdot k{:}?[\tilde{S}]; \alpha} \ [\textsc{Rcv}]$$

Renombramos en la regla [Rcv] $\Phi \cdot \tilde{x}{:}\tilde{S}$ por Φ', con lo cual $\Phi \equiv \Phi' \backslash[]\ \tilde{x}$ y $\Phi'(\tilde{x}) \equiv \tilde{S}$. También renombramos entonces $\tilde{S}$ por $\Phi'(\tilde{x})$, y $\Delta \cdot k{:}\alpha$ por Δ', así $\Delta \equiv \Delta' \backslash\backslash\ k$ y $\alpha \equiv \Delta' \bullet k$. Dado que la operación $<+$ quita del contexto el canal a ser agregado, la expresión $(\Delta' \setminus k) \cdot k{:}?[\tilde{S}]; \Delta' \bullet k$ equivale a $\Delta' <+ k{:}?[\tilde{S}]; \Delta' \bullet k$.

$$\frac{\Phi'; \Gamma \vdash P \triangleright \Delta'}{\Phi' \backslash[]\ \tilde{x}; \Gamma \ \vdash k?(\tilde{x}) \ \texttt{in}\ P \ \triangleright \Delta' <+ k{:}?[\Phi'(\tilde{x})]; \Delta' \bullet k} \ [\textsc{Rcv}]$$

En esta reescritura $\Phi'(\tilde{x})$ puede no estar bien definida si existen elementos del vector argumento que no estén en el dominio de Φ'. Por este motivo utilizamos la función `extractSorts`, definimos esta función a continuación, la cual básicamente extrae del contexto Φ' la lista $\tilde{S}$ de tipos básicos de los nombres que aparecen en $\tilde{x}$. Esta función tiene como argumentos: una lista de variables `ln`, el contexto Φ y una lista de tipos básicos `ss`. Este último argumento es utilizado cuando una variable no pertenece al dominio del contexto Φ, en este caso la función toma un tipo de la lista de tipos básicos dada como argumento, y lo asigna a esta variable. Esta forma de codificar el juicio, o sea asignar un tipo básico cualquiera (cualquiera que esté en la lista de tipos básicos dada como argumento) cuando el contexto no posee información alguna, es válido por la parte 2 del resultado de *weakening lemma* 3.5.3, donde se demuestra que si $a \notin \mathrm{dom}(\Gamma)$ y $\Phi;\Gamma \vdash P \triangleright \Delta$ entonces $\Phi;\Gamma \cdot a{:}S \vdash P \triangleright \Delta$, para cualquier tipo S. Recordamos que el contexto Γ, tal cual es usado en el resultado anterior, aquí se parte en dos: una de estas partes es Φ, por tanto este resultado también vale para el contexto Φ. De todas formas más adelante en la subsección 5.6.1 formalizamos este resultado particular. Esta función devuelve también un contexto con las asignaciones de variables de tipos a nombres hecha cuando Φ no tiene información de tipado para estos nombres. Observar que esta función termina codificando la extensión a listas de expresiones del algoritmo de inferencia de tipos en expresiones visto en la figura 3.5, demostramos esta afirmación más adelante en la sección 7.3.4.

```
extractSorts : List Expression → Sorts → List Sort →
               List Sort × Sorts
extractSorts [] Φ ss = [] , []
extractSorts ((num n) :: ns)  Φ ss = map× (λ xs → nat :: xs) id
                                     (extractSorts ns Φ ss)
extractSorts ((bool n) :: ns) Φ ss = map× (λ xs → boolean :: xs) id
                                     (extractSorts ns Φ ss)
extractSorts ((name n) :: ns) Φ (s :: ss)
  with Any.any (MS._≟×1_ n) Φ
... | yes n∈Φ = map× (λ xs → (Φ •∈s n∈Φ) :: xs) id
                    (extractSorts ns Φ ss)
extractSorts ((name n) :: ns) Φ (s :: ss)
    | no n∉Φ with Any.any (MS._≟×1_ n) (proj₂ (extractSorts ns Φ ss))
...   | yes n∈ext
  = map× (λ xs → ((proj₂ (extractSorts ns Φ ss)) •∈s n∈ext) :: xs) id
         (extractSorts ns Φ ss)
...   | no  _
  = map× (λ xs → s :: xs) (λ xs → (n , s) :: xs)
         (extractSorts ns Φ ss)
extractSorts ((name n) :: ns) Φ [] with Any.any (MS._≟×1_ n) Φ
... | yes n∈Φ = map× (λ xs → (Φ •∈s n∈Φ) :: xs) id
                    (extractSorts ns Φ [])
extractSorts ((name n) :: ns) Φ []
    | no n∉Φ with Any.any (MS._≟×1_ n) (proj₂ (extractSorts ns Φ []))
...   | yes n∈ext
  = map× (λ xs → ((proj₂ (extractSorts ns Φ [])) •∈s n∈ext) :: xs) id
         (extractSorts ns Φ [])
...   | no  _
  = map× (λ xs → nat :: xs) (λ xs → (n , nat) :: xs)
```

```
        (extractSorts ns Φ [])
```

```
Receive : (Δ : STypes)(Γ : Types)(Φ : Sorts)(p : Process)
        (k : ChannelVar&Value)(ln : List Name)(ls : List Sort) →
        (Φ , Γ) ⊢ p ▷ Δ →
        (Φ \\[]s ln , Γ) ⊢ receive k ln p ▷
            (Δ <+c (k , receiveS
                          (proj₁ (extractSorts (map name ln) Φ ls))
                          (Δ •c k)))
```

Throw

$$\frac{\Phi;\Gamma \vdash P \triangleright \Delta \cdot k{:}\beta}{\Phi;\Gamma \vdash \texttt{throw}\ k[k'];P\ \triangleright \Delta \cdot k{:}![\alpha];\beta \cdot k'{:}\alpha}\ [\textsc{Thr}]$$

Para la codificación de esta regla reescribimos como en la regla anterior $\Delta \cdot k{:}\beta$ por Δ', entonces $\Delta \equiv \Delta' \backslash\backslash\ k$ y $\beta \equiv \Delta' \bullet k$, pudiendo así también intercambiar β por $\Delta' \bullet x$. Además dado que la operación $<+$ quita del contexto el canal a ser agregado, la expresión $(\Delta' \backslash\backslash\ k) \cdot k{:}![\alpha];\Delta' \bullet k \cdot k'{:}\alpha$ equivale a $\Delta' <+ k{:}![\alpha];\Delta' \bullet k <+ k'{:}\alpha$.

$$\frac{\Phi;\Gamma \vdash P \triangleright \Delta'}{\Phi;\Gamma \vdash \texttt{throw}\ k[k'];P\ \triangleright \Delta' <+ k{:}![\alpha];\Delta' \bullet k <+ k'{:}\alpha}\ [\textsc{Thr}]$$

En la codificación de esta regla que exigimos que $k \neq k'$, y que no exista t tal que $\langle k', t\rangle \in \Delta$ explícitamente en la expresión de tipo `k' ∉₁st Δ`. En el juicio presentado en 2.6 esto también se pide implícitamente, debido a que la sintaxis $\Delta \cdot k{:}\alpha$ pide que $k \notin dom(\Delta)$. Entonces tenemos que para que la expresión $\Delta \cdot k{:}![\alpha];\beta \cdot k'{:}\alpha$ esté bien definida, necesariamente se tiene que cumplir que $k' \notin dom(\Delta \cdot k{:}![\alpha];\beta)$. Como se satisface que $dom(\Delta \cdot k{:}![\alpha];\beta) \equiv dom(\Delta) \cup \{k\}$ entonces se debe cumplir lo inicialmente pedido.

```
Throw : (Δ : STypes)(Γ : Types)(Φ : Sorts)
     (p : Process)(st : SType)
     (k k' : ChannelVar&Value) →
     (k≠k' : k ≢ k') →
     k' ∉₁st Δ →
     (Φ , Γ) ⊢ p ▷ Δ →
     (Φ , Γ) ⊢ throw k k' p ▷ (Δ <+c (k , sendST st (Δ •c k))
                                 <+c (k' , st))
```

Catch

$$\frac{\Phi;\Gamma \vdash P \triangleright \Delta \cdot k{:}\beta \cdot x{:}\alpha}{\Phi;\Gamma \vdash \texttt{catch}\ k(x)\ \texttt{in}\ P\ \triangleright \Delta \cdot k{:}?[\alpha];\beta}\ [\textsc{Cat}]$$

Reescribimos en la regla anterior $\Delta \cdot k{:}\beta \cdot x{:}\alpha$ por Δ', por tanto $\Delta \equiv \Delta' \backslash\ k \backslash\ x$, entonces $\alpha \equiv \Delta' \cdot x$ y $\beta \equiv \Delta' \cdot k$. Dado que la operación $<+$ quita del contexto el canal a ser agregado, la expresión $(\Delta' \backslash\ k \backslash\ x) \cdot k{:}?[\Delta' \cdot x]; \Delta' \cdot k$ equivale a $(\Delta' \backslash\ x) <+\ k{:}?[\Delta' \cdot x]; \Delta' \cdot k$, y ya que $k \neq x$ (por estar bien definida la expresión $\Delta \cdot k{:}\beta \cdot x{:}\alpha$) podemos intercambiar la operaciones inserción y remoción del contexto quedando así $(\Delta' <+\ k{:}?[\Delta' \cdot x]; \Delta' \cdot k) \backslash\ x$.

$$\frac{\Phi;\Gamma \vdash P \triangleright \Delta'}{\Phi;\Gamma \vdash \texttt{catch}\ k(x)\ \texttt{in}\ P\ \triangleright (\Delta' <+\ k{:}?[\Delta' \cdot x]; \Delta' \cdot k) \backslash\ x}\ [\textsc{Cat}]$$

```
Catch : (Δ : STypes)(Γ : Types)(Φ : Sorts)(p : Process)
  (k : ChannelVar&Value)(x : ChannelVar) →
  (k≢varx : k ≢ (var x)) →
  (Φ , Γ) ⊢ p ▷ Δ →
  (Φ , Γ) ⊢ catch k x p ▷
  ((Δ <+c (k , receiveST (Δ •c (var x)) (Δ •c k))) \\c (var x))
```

Conc

La codificación de esta regla es directa, notar que la premisa $\Delta\texttt{'} \otimes \Delta\texttt{''} \equiv \texttt{just}\ \Delta$ impone que $\Delta' \otimes \Delta''$ esté definido.

```
Conc :   (Δ Δ' Δ'' : STypes)(Γ : Types)(Φ : Sorts)
         (p q : Process) →
         Δ' ⊗ Δ'' ≡ just Δ →
         (Φ , Γ) ⊢ p ▷ Δ' →
         (Φ , Γ) ⊢ q ▷ Δ'' →
         (Φ , Γ) ⊢ pipe p q ▷ Δ
```

Nres

Para la codificación de esta regla reescribimos en [Nres] $\Gamma \cdot a{:}S$ por Γ', quedando así $\Gamma \equiv \Gamma' \backslash\ a$.

```
Nres : (Δ : STypes)(Γ : Types)(Φ : Sorts)
       (p : Process)(n : Name) →
       (Φ , Γ) ⊢ p ▷ Δ →
       (Φ , Γ \\n n) ⊢ vn n p ▷ Δ
```

Cres'

Dado que asumimos no existen canales con polaridades en los contextos la condición $\kappa^+ \notin \Delta \wedge \kappa^- \notin \Delta$ se vuelve trivial con lo cual la eliminamos de las premisas del juicio.

```
Cres' : (Δ : STypes)(Γ : Types)(Φ : Sorts)
       (p : Process)(κ : ChannelName) →
       (Φ , Γ) ⊢ p ▷ Δ →
       (Φ , Γ) ⊢ vc κ p ▷ Δ
```

Permutation-env

Esta última regla no aparece entre las reglas originales introducidas en la sección 2.6, pero es necesario agregarla en nuestro sistema de tipos para introducir el concepto de que el orden de los pares en los contextos no es relevante. En [6] esto se impone al definir que los contextos son funciones parciales. En nuestra formalización utilizamos listas para codificar los contextos, por lo que debemos agregar esta regla explícitamente de forma de establecer así que el orden de los pares no tiene significado alguno en nuestro juicio.

Recordamos que utilizamos la función $\cdot$ en la expresión $\Gamma \cdot n$ para obtener el tipo asignado a un nombre n en un contexto Γ. Nuestra implementación de esta funcionalidad retorna la segunda componente del par más a la izquierda en la lista de pares con que codificamos el contexto, tal que su primera componente es n. Por este motivo el orden de los pares tiene relevancia para esta función en el caso de que haya pares con su primer componente repetida, con lo cual tenemos cuidado de no permitir pares con su primer componente repetida en este juicio. Observar que en la regla base [INACT] de nuestro sistema de tipos agregamos como premisa que los contextos no tengan información repetida. A partir de esta regla base, y dado que las demás reglas del juicio no introducen componentes repetidas a los contextos, puede demostrarse fácilmente que en todos los juicios derivables no hay componentes repetidas en los contextos, validando así la siguiente regla.

```
Permutation-env : {p : Process}{Φ Φ' : Sorts}
    {Δ Δ' : STypes}{Γ Γ' : Types} →
    permutation Φ Φ' → permutation Γ Γ' → permutation Δ Δ' →
    (Φ , Γ) ⊢ p ▷ Δ →
    (Φ' , Γ') ⊢ p ▷ Δ'
```

5.6. Propiedades del Sistema de Tipos

Veremos algunas propiedades importantes que satisface el sistema de reglas visto en la sección anterior, las cuales usaremos posteriormente para demostrar la corrección del algoritmo de inferencia de tipos.

Las dos primeras propiedades son las codificaciones directas del resultado *weakening lemma* parte 2 demostrado informalmente en el apéndice A. Este resultado se codifica en dos partes como consecuencia de que al contexto Γ se le quitan los tipos básicos que pasan al contexto Φ.

Finalmente el último resultado valida que el sistema de tipos presentado es coherente con el sistema sin variables de tipo, esto es, si el sistema de asignación de esquemas de tipos valida el tipo de un proceso, entonces para cualquier instancia de las variables de tipo, la asignación se sigue satisfaciendo. En particular bajo una sustitución *ground*, esto es que instancie todas las variables de tipo en tipos concretos, el juicio seguirá valiendo sin variables de tipo alguna, tal cual el introducido en la sección 2.6.

5.6.1. Weakening Lemma para los Contextos de Tipos Básicos

En la siguiente figura mostramos la codificación del weakening lema presentado en la sección 2.7.

```
weakening-envΦ : {p : Process}{Φ : Sorts}{Δ : STypes}
          {Γ : Types}{n : Name}{s : Sort} →
          n ∉₁s Φ →
          (Φ , Γ) ⊢ p ▷ Δ →
          (Φ <+s (n , s) , Γ) ⊢ p ▷ Δ
```

La prueba de este lema se realizó por inducción en el juicio de tipado. Para el caso de inductivo de la regla *Receive* presentada en la subsección 5.5, es necesario asumir que para el proceso `receive k ln p`, $n \notin ln$. De esta forma se cumple $((\Phi <+ (n,t)) \setminus\setminus [] s \quad ln) \equiv ((\Phi \setminus\setminus [] s \quad ln) <+ (n,t))$. Esta suposición es correcta ya que el juicio presentado en la sección 2.6 asume que los procesos son α-convertibles, esto es, podemos cambiar el nombre de cualquier variable ligada a nuestra discreción en los procesos. Justamente en el proceso `receive k ln p` las variables que aparecen en `ln` están ligadas, por tanto las podemos renombrar (renombrándolas también en `p`) de forma de que se cumpla $n \notin ln$. En nuestra formalización no codificamos la α-conversión, por lo que no pudo ser completada la codificación de este caso. De todas, formas si asumimos $n \notin ln$ la demostración puede ser completada.

5.6.2. Weakening Lemma Extendido a Listas de Contextos de Tipos Básicos

Esta propiedad extiende el resultado anterior pudiendo así concatenar contextos al contexto original bajo el cual tipaba el proceso, siempre y cuando se cumpla que todo elemento de este contexto a ser agregado no pertenezca al contexto de tipos básicos original. La demostración de este lema es directa a partir

del anterior resultado y algunos lemas de la concatenación de listas.

```
weakening-fold-l-Φ : {p : Process}{Φ1 Φ2 : Sorts}{Δ : STypes}
           {Γ : Types} → (∀ n → (n ∈₁s Φ2 → n ∉₁s Φ1)) →
           NRLs.NoRep Φ2 →
           (Φ1 , Γ) ⊢ p ▷ Δ →
           (Φ2 ++ Φ1 , Γ) ⊢ p ▷ Δ
```

El siguiente resultado es un corolario directo de la propiedad anterior utilizando la regla `Permutation-env` del sistema de tipos.

```
weakening-fold-r-Φ : {p : Process}{Φ1 Φ2 : Sorts}{Δ : STypes}
          {Γ : Types} → (∀ n → (n ∈₁s Φ2 → n ∉₁s Φ1)) →
          NRLs.NoRep Φ2 →
          (Φ1 , Γ) ⊢ p ▷ Δ →
          (Φ1 ++ Φ2 , Γ) ⊢ p ▷ Δ
```

5.6.3. Weakening Lemma para los Contextos de Tipos de Puertos

Codificamos el weakening lemma para los contextos de tipos de puertos en la siguiente figura.

```
weakening-envΓ : {p : Process}{Φ : Sorts}{Δ : STypes}
          {Γ : Types}{n : Name}{t : Type} →
          n ∉₁n Γ →
          (Φ , Γ) ⊢ p ▷ Δ →
          (Φ , Γ <+n (n , t)) ⊢ p ▷ Δ
```

La prueba de este resultado también se realizó por inducción en el juicio de tipado. Aquí faltó completar el caso inductivo de la regla *Nres* introducida en la subsección 5.5. En esta regla necesitamos suponer que para el proceso `vn n' p` se cumple $n \neq n'$, de esta forma $((\Gamma <+ (n,t)) \setminus\setminus n\ n') \equiv ((\Gamma \setminus\setminus n\ n') <+ (n,t))$. Dado que la primitiva `vn` liga la variable `n`, nuevamente esta suposición es correcta por ser los procesos α-convertibles, pudiéndose así renombrar la variable n de forma que se cumpla $n \neq n'$. Asumiendo esta última condición se puede completar la demostración del caso inductivo de la regla *Nres*.

5.6.4. Weakening Lemma Extendido a Listas de Contextos de Tipos de Puertos

Esta propiedad extiende el resultado anterior pudiendo así concatenar contextos al contexto original bajo el cual tipaba el proceso, siempre y cuando se cumpla que todo elemento de este contexto a ser agregado no pertenezca al contexto de tipos de puertos original. La demostración de este lema es directa a partir del anterior resultado.

```
weakening-fold-l-Γ : {p : Process}{Φ : Sorts}{Δ : STypes}
           {Γ Γ2 : Types} → (∀ n → (n ∈₁n Γ2 → n ∉₁n Γ)) →
           NRLt.NoRep Γ2 →
           (Φ , Γ) ⊢ p ▷ Δ →
           (Φ , Γ2 ++ Γ) ⊢ p ▷ Δ
```

El siguiente resultado es un corolario directo de la propiedad anterior utilizando la regla `Permutation-env` del sistema de tipos.

```
weakening-fold-r-Γ : {p : Process}{Φ : Sorts}{Δ : STypes}
           {Γ Γ2 : Types} → (∀ n → (n ∈₁n Γ2 → n ∉₁n Γ)) →
           NRLt.NoRep Γ2 →
           (Φ , Γ) ⊢ p ▷ Δ →
           (Φ , Γ ++ Γ2) ⊢ p ▷ Δ
```

5.6.5. Cerradura del Juicio Bajo las Operaciones de Sustituciones de Variables de Tipo en los Contextos

Este resultado determina que para todo proceso `p`, contextos Φ , Γ , Δ cualesquiera , y toda sustitución de las variables de tipo básico `ss` y de las variables de tipo de canal `sst`, si se cumple el juicio $(\Phi , \Gamma) \vdash p \triangleright \Delta$ entonces también se cumple el juicio aplicando las sustituciones a los contextos. Demostramos en su totalidad este resultado haciendo inducción en el juicio de tipado.

```
conservation-env : {p : Process}{Φ : Sorts}
         {Δ : STypes}{Γ : Types}
         (ss : List (ℕ × Sort))(sst : List (ℕ × SType)) →
         (Φ , Γ) ⊢ p ▷ Δ →
         (lsubSSS ss Φ ,
          lsubSTLPNT sst (lsubSLPNT ss Γ)) ⊢ p ▷
                         lsubSTLPCST sst (lsubSLPCST ss Δ)
```

Capítulo 6

Formalización de los Algoritmos de Unificación

Definimos el problema de unificación como el de encontrar una sustitución que haga iguales a dos términos, si es que es posible. Una sustitución es una función que asigna términos a variables. El problema de unificación puede generalizarse para una lista de pares de términos a unificar, en cuyo caso la sustitución resultado iguala a todos los pares de esta lista.

En este capítulo presentamos la formalización en Agda de los algoritmos de unificación para los tipos básicos y para los Tipos de Sesión. Se presentará el código Agda y una explicación precisa de las principales definiciones y pruebas dejando algunos resultados auxiliares en el apéndice B.

Una introducción detallada al problema de la unificación de términos y su formalización en Teoría Constructiva de Tipos (utilizando Alf) se puede encontrar en [20], y nuestra formalización en Agda sigue los lineamientos de este trabajo. Utilizamos también la formalización de inducción bien fundada de la biblioteca estándar de Agda, presentada en la sección 4.2.8. Este resultado permite definir funciones por recursión general, siempre y cuando los argumentos *decrezcan* de alguna forma y la relación de orden sobre la cual decrecen sea bien fundada.

6.1. Unificación de Tipos Básicos

En esta sección presentamos el algoritmo de unificación en tipos básicos, para lo cual comenzamos con la unificación de dos tipos básicos para luego generalizarlo a una lista. Finalmente damos la prueba de corrección del algoritmo de unificación.

6.1.1. Unificación de dos Tipos Básicos

Los tipos básicos (`Sorts`) son `nat`, `boolean` o `varS n` donde `n`:$\mathbb{N}$, con lo cual unificar dos tipos básicos t_1 y t_2 se reduce a encontrar un par `(n,s)` de tipo $\mathbb{N}\times$`Sort` que los iguale. Para considerar el caso en que la sustitución es

trivial por ser t_1 idéntico a t_2 utilizamos el tipo `Maybe` presentado en la sección 4.2.2, indicando con `nothing` que la sustitución es vacía. A su vez como la unificación de dos términos puede fallar volvemos a utilizar este tipo indicando con `nothing` la imposibilidad de unificarlos. Así la unificación de una pareja de tipos básicos definida en la figura 6.1 devuelve un elemento de tipo `Maybe (Maybe (ℕ×Sort))`:

- `nothing` en caso de no existir sustitución que los unifique
- `just nothing` en caso de que la sustitución sea vacía
- `just (just (n , s))` en caso de que la sustitución sea el par `(n , s)`

```
unifyS : Sort → Sort → Maybe (Maybe (ℕ × Sort))
unifyS nat nat = just nothing
unifyS boolean boolean = just nothing
unifyS (varS n) nat = just (just (n , nat))
unifyS (varS n) boolean = just (just (n , boolean))
unifyS (varS n) (varS m) with n ≟ m
unifyS (varS n) (varS .n) | yes refl = just nothing
unifyS (varS n) (varS m)  | no n≢m   = just (just (n , varS m))
unifyS nat     (varS n) = just (just (n , nat))
unifyS boolean (varS n) = just (just (n , boolean))
unifyS s1 s2 = nothing -- casos nat boolean y boolean nat
```

Figura 6.1: Unificación de dos tipos básicos

6.1.2. Unificación de Listas de Pares de Tipos Básicos

Para unificar una lista de pares se la recorre llevando un acumulador de sustituciones. Para cada par de la lista, aplicamos la función presentada en la figura 6.1. Si no existe sustitución el algoritmo para determinando que no existe solución, si en cambio existe sustitución la aplicamos al acumulador y al resto de las pares que quedan por recorrer, agregando la nueva sustitución al acumulador de sustituciones. Cuando el par de listas de tipos básicos se vacía devolvemos el acumulador. Se puede ver en la figura 6.2 la implementación tentativa de este algoritmo.

El problema de esta definición es que no es trivial comprobar que la recursión es hecha en listas decrecientes, ya que en el último caso la llamada recursiva se hace sobre la aplicación de las sustituciones al resto de la lista y no directamente al resto de la lista. El compilador de Agda no reconoce que la lista está decreciendo efectivamente, debido a lo cual falla la comprobación de terminación.

Utilizamos la teoría de recursión bien fundada para demostrar la parada de este algoritmo. En la sección 4.2.8 damos una breve presentación de la recursión bien fundada en Agda. Para utilizar esta teoría necesitamos demostrar que las llamadas recursivas de este algoritmo se realizan sobre listas que decrecen según un orden bien fundado. Usamos entonces el largo de las listas y la relación $<$ entre naturales como un orden bien fundado ya conocido.

Comenzamos definiendo en la figura 6.3 el predicado de accesibilidad particular `UniLS-Acc` para el conjunto de las listas sobre las cuales trabaja este

```
unifyLSac : List (Sort × Sort) → List (ℕ × Sort) →
            Maybe (List (ℕ × Sort))
unifyLSac [] ac = just ac
unifyLSac ((s1 , s2) :: l) ac with unifyS s1 s2
... | nothing = nothing
... | just nothing = unifyLSac l ac
... | just (just u) = unifyLSac (subSLS u l) (u :: (subSSS u ac))

unifyLS : List (Sort × Sort) → Maybe (List (ℕ × Sort))
unifyLS ss = unifyLSac ss []
```

Figura 6.2: Codificación tentativa para la unificación de listas de pares de tipos básicos

algoritmo. Podemos apreciar como las reglas de los constructores del conjunto se basan en las reglas de la función `unifyLSac`. Demostramos ahora que todas las listas satisfacen el predicado `UniLS-Acc`. Los casos base son directos: la lista vacía es accesible aplicando el constructor `uniLS[]`, mientras que para una lista con al menos una par, si no encontramos sustitución que unifique este par entonces aplicando el constructor `uniLSnoth` probamos que es accesible. En el caso de la regla `uniLSjustnoth` es inmediato que ver que la lista construida es más larga que la lista argumento. Finalmente para la regla `uniLSjustjust` utilizamos un argumento similar además del hecho de que una sustitución en una lista no altera su largo.

```
data UniLS-Acc :  List (Sort × Sort) → Set where
  uniLS[] : UniLS-Acc []
  uniLSnoth  : (s1 s2 : Sort)(l : List (Sort × Sort)) →
               unifyS s1 s2 ≡ nothing →
               UniLS-Acc ((s1 , s2) :: l)
  uniLSjustnoth  : (s1 s2 : Sort)(l : List (Sort × Sort)) →
                   unifyS s1 s2 ≡ just nothing → UniLS-Acc l →
                   UniLS-Acc ((s1 , s2) :: l)
  uniLSjustjust  : (s1 s2 : Sort)(l : List (Sort × Sort)) →
                   ∃ (λ u → ((unifyS s1 s2 ≡ just (just u)) ×
                              UniLS-Acc (subSLS u l)))    →
                   UniLS-Acc ((s1 , s2) :: l)
```

Figura 6.3: Listas sobre las cuales trabaja el algoritmo `unifyLsac`

Tal cual mencionamos en 4.2.8, dada una lista l cualquiera si podemos construir un elemento de tipo $UniLS\text{–}Acc\ l$, entonces no existe una secuencia infinita descendiente $\ldots l_2 < l_1 < l$, donde la relación $<$ entre listas es menor largo. En la figura 6.4 demostramos el lema `uniLsAc-Acc`, que establece que para cualquier natural n y sustitución l, si $length\ l \equiv n$ entonces l es accesible, o sea podemos construir un elemento de tipo $UniLS\text{–}Acc\ l$. Utilizamos en esta demostración los constructores de recursión bien fundada de los naturales bajo la relación $<$ que provee la biblioteca estándar de Agda. También utilizamos el predicado `Inspect` presentado en la sección 4.2.7, el cual brinda una demostración de equivalencia entre una expresión y su resultado. Para el caso en que la

lista l sea vacía, el constructor `uniLS[]` es la demostración de que es accesible. Si la lista l no es vacía, o sea es de la forma $(s_1, s_2) :: l$, entonces aplicamos el predicado `Inspect` sobre la unificación de la par de tipos básicos s_1 y s_2. Si no existe sustitución que unifique s_1 y s_2, el resultado es `nothing`, y en ese caso utilizamos la regla `uniLSnoth` con la demostración de que la unificación es equivalente a `nothing` brindada por el uso de `Inspect`. Si el resultado de la unificación es vacía (`just nothing`), utilizamos la regla `uniLSjustnoth` con la demostración de esto y una demostración de que el resto de la lista es accesible. Para esto usamos el operador de recursión bien fundada con argumentos: el largo de la lista l, una demostración de que $length\ l < (length\ ((s_1, s_2) :: l))$ que reduce a $length\ l < suc\ (length\ l)$, la lista l y una demostración de que $length\ l \equiv length\ l$. En el último caso cuando el resultado de la unificación es una sustitución u, utilizamos la regla `uniLSjustjust`, con la demostración de que la aplicación de la sustitución u a l (denotada $l\ u$), es accesible. Utilizamos nuevamente el recursor con la demostración de que el largo de la lista $l\ u < (s_1, s_2) :: l$, usando el lema `length-map` de la biblioteca estándar de Agda, que provee una demostración de que la función `map` no altera el largo de una lista (observar en la figura 5.10 que la definición de la función `subSLS` se basa en la aplicación de un `map`).

```
allListUniLS-AccPred : ℕ → Set
allListUniLS-AccPred n = (l : List (Sort × Sort)) →
                         length l ≡ n →
                         UniLS-Acc l

n≡m→n≤′m : {n m : ℕ} → n ≡ m → n ≤′ m
n≡m→n≤′m refl =  ≤′-refl

allℕ-Acc : (n : ℕ) → allListUniLS-AccPred n
allℕ-Acc = <-rec allListUniLS-AccPred allℕ-Acc'
  where allℕ-Acc' : ∀ n → <-Rec allListUniLS-AccPred n →
                                allListUniLS-AccPred n
        allℕ-Acc' ._ rec [] refl = uniLS[]
        allℕ-Acc' ._ rec ((s1 , s2) :: l) refl
          with inspect (unifyS s1 s2)
        allℕ-Acc' ._ rec ((s1 , s2) :: l) refl
          | nothing with-≡ p≡
          = uniLSnoth s1 s2 l (sym p≡)
        allℕ-Acc' ._ rec ((s1 , s2) :: l) refl
          | (just nothing)  with-≡ p≡
          =  uniLSjustnoth s1 s2 l (sym p≡)
                           (rec (length l) ≤′-refl  l refl)
        allℕ-Acc' ._ rec ((s1 , s2) :: l) refl
          | (just (just u)) with-≡ p≡
          with cong suc (length-map
                           {A = Sort × Sort}
                           {B = Sort × Sort}
                           (λ ps → (subSS u (proj₁ ps) ,
                                    subSS u (proj₂ ps)))
                                 l)
        allℕ-Acc' ._ rec ((s1 , s2) :: l) refl
          | (just (just u)) with-≡ p≡
          | prf
```

```
          = uniLSjustjust s1 s2 l
                          (u , (sym p≡) ,
                                        rec (length (subSLS u l))
                                            (n≡m→n≤′m prf)
                                            (subSLS u l) refl)

allListUniLS-Acc : (l : List (Sort × Sort)) → UniLS-Acc l
allListUniLS-Acc l = allℕ-Acc (length l) l refl
```

Figura 6.4: Demostración de la accesibilidad del conjunto `UniLS-Acc`

Utilizando este resultado podemos ahora redefinir la función de unificación de listas de tipos básicos por recursión primitiva en la demostración de accesibilidad de la lista argumento, quedando la definición presentada en la figura 6.5. Observar que esta forma de introducir la recursión bien fundada para garantizar la terminación, hace que no se altere la estructura original de la función de la figura 6.2.

```
unifyLS-ac : (l : List (Sort × Sort)) → UniLS-Acc l →
             List (ℕ × Sort) → Maybe (List (ℕ × Sort))
unifyLS-ac .[] uniLS[] ac = just ac
unifyLS-ac .((s1 , s2) :: l)
            (uniLSnoth s1 s2 l punifs1s2≡noth)
            ac = nothing
unifyLS-ac .((s1 , s2) :: l)
            (uniLSjustnoth s1 s2 l
                           punifs1s2≡justnoth punifyLS-acl)
            ac = unifyLS-ac l punifyLS-acl ac
unifyLS-ac .((s1 , s2) :: l)
            (uniLSjustjust s1 s2 l
                           (u , (punifs1s2≡justu ,
                                 punifLS-acsubSLSul)))
             ac = unifyLS-ac (subSLS u l)
                             punifLS-acsubSLSul
                             (u :: (subSSS u ac))

unifyLS : List (Sort × Sort) → Maybe (List (ℕ × Sort))
unifyLS ss = unifyLS-ac ss (allListUniLS-Acc ss) []
```

Figura 6.5: Codificación final de la unificación de listas de pares de tipos básicos

En las siguientes secciones presentamos los pasos principales para definir y demostrar la corrección del algoritmo de unificación siguiendo las ideas de [20]. Más detalles se encuentran en el apéndice B.

6.1.3. Unificador más General

Presentamos en esta sección la formalización del concepto de unificador más general. Comenzamos definiendo cuando una lista de pares de tipos básicos unifica. A diferencia de [20] donde se da una definición inductiva particular, nosotros

utilizamos el predicado `All` presentado en la sección 4.2.5. Esta formaliza la idea de que todos los elementos de una lista cumplan determinada propiedad. En este caso la propiedad es que los pares de la lista sean idénticos.

```
unifiesLPS : List (ℕ × Sort) → List (Sort × Sort) → Set
unifiesLPS ss lps = All (uncurry _≡_) (lsubSLS ss lps)
```

De manera similar definimos cuando una sustitución unifica otra sustitución.

```
unifiesSS : List (ℕ × Sort) → List (ℕ × Sort) → Set
unifiesSS ss' ss = All (λ p → (lsubS ss' (varS (proj₁ p))) ≡
                              lsubS ss' (proj₂ p))
                       ss
```

Queremos probar que si la sustitución ss es el resultado de unificar una lista pares de tipos básicos lp, entonces ss y lp son equivalentes en el sentido que ambas tienen el mismo conjunto de unificadores, independientemente del orden de los pares. Como el algoritmo de unificación `unifyLS` es definido a partir de la función auxiliar `unifyLS-ac`, la cual tiene un acumulador de sustituciones como parámetro adicional, necesitamos dar una noción más general de equivalencia. Formalizamos entonces esta noción mediante la definición de ≡`LpsSS-SS`.

```
≡LpsSS-SS : List (Sort × Sort) → List (ℕ × Sort) →
            List (ℕ × Sort) →
            Set
≡LpsSS-SS lp ss ss' = (ss1 : List (ℕ × Sort)) →
              (((unifiesLPS ss1 lp) × (unifiesSS ss1 ss)) ⇔
               unifiesSS ss1 ss')
```

Ahora definimos noción de equivalencia entre listas de pares de tipos y sustituciones como un caso particular de la definición anterior:

```
_≡Lps-SS_ :  List (Sort × Sort) → List (ℕ × Sort) → Set
lp ≡Lps-SS  ss = ≡LpsSS-SS lp [] ss
```

Figura 6.6: Equivalencia entre listas de pares de tipos y sustituciones

Finalmente introducimos la noción de unificador más general en la figura 6.8. Una sustitución ss es el unificador más general de una lista de pares de tipos básicos lps; si ss unifica lps, y además cualquier otra sustitución ss' que también unifiqué lps cumple que ss es al menos tan general como ss'. La relación "tan general como" es definida tal que ss es tan general como ss' si existe otra sustitución ss'' tal que $\forall t \in Sort,\ t\ ss' \equiv (t\ ss)\ ss''$.

```
_≤S_ : List (ℕ × Sort) → List (ℕ × Sort) → Set
ss1 ≤S ss2 = ∃ (λ ss3 →
                  ((s : Sort) →
                  (lsubS ss2 s) ≡ (lsubS ss3 (lsubS ss1 s))))
```

Figura 6.7: Relación "tan general como"

```
mguS : List (ℕ × Sort) → List (Sort × Sort) → Set
mguS ss lps = (unifiesLPS ss lps) ×
              ((ss2 : List (ℕ × Sort)) → unifiesLPS ss2 lps →
                                         (ss ≤S ss2))
```

Figura 6.8: Unificador más general

6.1.4. Propiedades de la Sustitución

En esta sección presentamos algunas propiedades de la sustitución que usaremos en las próximas secciones. Todas las propiedades presentadas fueron demostradas en Agda y para aliviar la lectura sólo mencionamos aquí los principales resultados. Las pruebas completas se encuentran en el Apéndice B.

Primero definimos el siguiente concepto:

Definición Una sustitución s es *idempotente* si para cualquier tipo básico t se cumple que $t\ s \equiv (t\ s)\ s$

En la figura 6.9 podemos ver la definición en Agda de este concepto.

```
IdempotentS : (ss : List (ℕ × Sort)) → Set
IdempotentS ss = (s : Sort) →
                 lsubS ss (lsubS ss s) ≡ lsubS ss s
```

Figura 6.9: Sustitución idempotente

Una condición suficiente para que una sustitución sea idempotente es que las variables de tipo que aparecen en el dominio y el codominio de la sustitución sean disjuntos. Esto se define en la siguiente figura.

```
  data IdemS : List (ℕ × Sort) → Set where
  []idem : IdemS []
  ::idem : (ss : List (ℕ × Sort))(n : ℕ)(s : Sort) →
           IdemS ss →
           disjoin (varsS s) (dom ((n , s) :: ss)) →
           n ∉ dom ss → n ∉ varsSS ss →
           IdemS ((n , s) :: ss)
```

La prueba detallada de que esta propiedad implica la idempotencia se detalla en B.1.1.

Finalmente presentamos el resultado que establece que si una variable n pertenece al conjunto de variables que aparece en un tipo básico s y $n \neq s$ entonces no existe sustitución alguna que unifique n y s. Este resultado se demuestra fácilmente por inducción en el tipo básico s

```
∈∧≢∧unify→⊥ : (ss : List (ℕ × Sort))(n : ℕ)(s : Sort) →
              n ∈ (varsS s) → varS n ≢ s →
              lsubS ss (varS n) ≢ lsubS ss s
```

6.1.5. Corrección del Algoritmo de Unificación

Las siguientes subsecciones introducen algunas propiedades necesarias para probar la corrección del algoritmo de unificación.

Existencia

Los dos siguientes resultados prueban que el resultado del algoritmo de unificación no dice que no existe sustitución si y sólo si existe una sustitución que unifique la lista de pares dada como argumento.

```
unifiesS→¬error : (lps : List (Sort × Sort)) →
                  ∃ (λ ss → unifiesLPS ss lps) →
                  unifyLS lps ≢ nothing
unifiesS→¬error lps (ss , unifiesLPSsslps) unifyLSlps≡noth
  = error∧unifies→⊥ lps (unifyLSlps≡noth ,
                         (ss , unifiesLPSsslps))
```

Figura 6.10: Propiedad de existencia 1

```
error→¬unifiesS : (lps : List (Sort × Sort)) →
                  unifyLS lps ≡ nothing  →
                  ¬ ∃ (λ ss → (unifiesLPS ss lps))
error→¬unifiesS lps unifyLSlps≡noth (ss , unifiesLPSsslps)
   = error∧unifies→⊥ lps (unifyLSlps≡noth ,
                          (ss , unifiesLPSsslps))
```

Figura 6.11: Propiedad de existencia 2

La prueba detallada de estos resultados pueden verse en B.1.2.

Propiedades de las Variables

Ahora enunciamos el resultado de la figura 6.12, el cual determina que el conjunto de variables que ocurren en la sustitución que halla el algoritmo de unificación está incluido en de las variables que ocurren en la lista de pares que unificados. La prueba se encuentra en B.1.3.

```
unifyVarsS⊆lps : (ss : List (ℕ × Sort))
                 (lps : List (Sort × Sort)) →
                 unifyLS lps ≡ just ss →
                 varsSS ss ++ dom ss ⊆ varsSLPT lps
```

Figura 6.12: Propiedad de las variables de la sustitución resultado

Propiedad de Idempotencia

La siguiente propiedad establece que si el algoritmo de unificación resulta en una sustitución, esta es necesariamente idempotente. La prueba de la misma se encuentra en B.1.4.

```
idempotentUnifyS : (lps : List (Sort × Sort))
                   (ss : List (ℕ × Sort)) →
                   unifyLS lps ≡ just ss → IdempotentS ss
```

Unificador más General

Finalmente demostramos que si el algoritmo de unificación resulta en una sustitución para una lista de pares de tipos básicos dada como entrada, entonces el resultado es el unificador más general de la lista de entrada:

```
mguSprop :  (lps : List (Sort × Sort))
            (ss : List (ℕ × Sort)) →
            unifyLS lps ≡ just ss →
            mguS ss lps
```

Figura 6.13: Corrección del algoritmo de unificación para tipos básicos

La demostración completa de esta propiedad se encuentra en B.1.5

6.2. Unificación de Tipos de Canal

Al igual que en el comienzo de esta sección volvemos a definir el problema de unificación ahora para tipos de canales, esto es, encontrar la sustitución que haga iguales a todos los pares de una lista de pares de tipos de canal, si es que existe esta sustitución.

Seguimos un orden similar que cuando presentamos la unificación de tipos básicos, dejando algunos detalles en el apéndice B.2. Comenzamos así definiendo la unificación para dos tipos de canal.

6.2.1. Unificación de Dos Tipos de Canal

En la figura 6.15 presentamos la unificación de dos tipos de canal, esto es dados dos tipos de canal devuelve si es que existe la sustitución que los iguala. Los tipos de canal contienen tipos básicos en su estructura, en la figura 5.4 se ve como los constructores `receiveS` y `sendS` reciben listas de tipos básicos como argumentos. Por tanto en estos dos casos de tipos de canal la unificación significa también unificar las listas de tipos básicos tal cual fue vista en la sección 6.1.2. En las dos reglas que definen la unificación de dos tipos de canal de la forma `receiveS` o `sendS` utilizamos la función `unifyLS` vista en la anterior sección para unificar listas de tipos básicos. En el caso que exista unificación el resultado de esta puede ser de dos clases:

- una lista de pares de tipos de canal a ser unificadas y una sustitución de tipos básicos

- una sustitución de tipos de canal

Modelamos estas dos posibilidades mediante la unión disjunta, denotada por ⊎. Este tipo de datos está implementado en la biblioteca estándar de Agda, y fue presentado en la sección 4.2.3.

También utilizamos el tipo de datos `Maybe` para especificar si es que existe unificación posible.

Definimos la función por pattern matching en el par de tipos de canal de entrada. En los casos que ambos tipos tienen como constructor principal un `receiveS` o `sendS` verificamos que los largos de las listas de tipos básicos sean iguales, para luego unificarlas. En caso que exista una unificación la devolvemos junto con una lista formada por un único par de tipos de canal correspondientes a los tipos de canal argumento de los constructores principales.

En caso que ambos tipos tienen como constructores principales `receiveST` o `sendST`, devolvemos una unificación vacía de tipos básicos, y una lista con dos pares de tipos de canal, la primera formada con los primeros y segundos argumentos de los constructores respectivamente.

Cuando los tipos son ambos variables de tipo comunes o ambos complementadas, si las variables son idénticas entonces devolvemos una unificación vacía de tipos básicos, junto con una lista vacía de tupas de tipos de canal. Si en cambio las variables son distintas, devolvemos una sustitución que corresponde a sustituir una variable por la otra.

Utilizamos la función `ocurrST` (figura 6.14) para determinar si una variable de tipo de canal aparece dentro de otro tipo de canal dado. La utilizamos en los casos que uno de los tipos de canal a unificar es una variable de tipo de canal n, y el otro tipo t es cualquiera salvo una variable de tipo de canal idéntica, este caso se descarta ya que es definido en una regla anterior. Cuando la variable n ocurre dentro de t, no siendo t una variable de tipo idéntica a n, entonces no es posible unificación alguna.

```
ocurrST : (n : ℕ)(st : SType) →
          Dec (Any (_≡_ n) (vars st))
ocurrST n st = Any.any (_≟_ n) (vars st)
```

Figura 6.14: Función `ocurrST`

En el caso que ambos tipos de canal sean `end` devolvemos una unificación vacía de tipos básicos y una lista vacía de tipos de canal a ser unificados.

Finalmente la última regla determina que para todos los casos no considerados hasta ahora la unificación no existe.

6.2.2. Unificación de Listas de Pares de Tipos de Canal

Al igual que para listas de pares de tipos básicos, para unificar una lista de pares de tipos de canal recorremos la lista llevando dos acumuladores de sustituciones, una de tipos básicos y otra de tipos de canal. Para cada par de la lista, aplicamos la función presentada en la sección anterior, si no existe unificación alguna para la par considerada el algoritmo para determinando que no existe solución alguna. Si en cambio devuelve una sustitución de tipos básicos *lus* y

```
unifyST : SType × SType →
          Maybe ((List (SType × SType) × List (ℕ × Sort))
                 ⊎
                 (ℕ × SType))
unifyST (receiveS ss1 t1 , receiveS ss2 t2)
    with (length ss1 ≟ length ss2)
... | no _ = nothing
... | yes _ with unifyLS (zip ss1 ss2)
...   | nothing = nothing
...   | just us = just (inj₁ ([(t1 , t2)] , us))
unifyST (sendS ss1 t1 , sendS ss2 t2)
    with (length ss1 ≟ length ss2)
... | no _ = nothing
... | yes _ with unifyLS (zip ss1 ss2)
...   | nothing = nothing
...   | just us = just (inj₁ ([(t1 , t2)] , us))
unifyST (receiveST t1 t2 , receiveST t3 t4)
    = just (inj₁ ((t1 , t3) ∷ (t2 , t4) ∷ [] , []))
unifyST (sendST t1 t2 , sendST t3 t4)
    = just (inj₁ ((t1 , t3) ∷ (t2 , t4) ∷ [] , []))
unifyST (varST n , varST m) with n ≟ m
... | yes _ = just (inj₁ ([] , []))
... | no _  = just (inj₂ (n , varST m))
unifyST (varSTCompl n , varSTCompl m) with n ≟ m
... | yes _ = just (inj₁ ([] , []))
... | no _  = just (inj₂ (n , varST m))
unifyST (varST n , t) with ocurrST2 n t
... | yes _   = nothing
... | no _  = just (inj₂ (n , t))
unifyST (t , varST n) with ocurrST2 n t
... | yes _   = nothing
... | no _  = just (inj₂ (n , t))
unifyST (varSTCompl n , t) with ocurrST2 n (complST t)
... | yes _   = nothing
... | no _  = just (inj₂ (n , complST t))
unifyST (t , varSTCompl n) with ocurrST2 n (complST t)
... | yes _   = nothing
... | no _  = just (inj₂ (n , complST t))
unifyST (end , end) = just (inj₁ ([] , []))
unifyST (t1 , t2) = nothing
```

Figura 6.15: Unificación de dos tipos de canal

una lista de pares de tipos de canal *lpst*2, entonces agregamos a la lista de pares que queda por recorrer la lista de pares *lpst*2 y aplicamos a este resultado la sustitución *lus*. También aplicamos a ambos acumuladores la sustitución *lus*. Finalmente agregamos al acumulador de tipos básicos *lus*. Mientras que si devuelve una sustitución de tipos de canal *st*, aplicamos esta sustitución al resto de la lista por recorrer y al acumulador de sustituciones de canal, para finalmente agregar *st* a este acumulador. Cuando la par de listas de tipos de canal se vacía

devolvemos ambos acumuladores. En la figura 6.16 podemos ver la definición completa.

```
unifyLPST : List (SType × SType) → List (ℕ × Sort) →
              List (ℕ × SType) →
              Maybe (List (ℕ × Sort) × List (ℕ × SType))
unifyLPST []           as at = just (as , at)
unifyLPST (pst :: lpst) as at with unifyST pst
... | nothing = nothing
... | just (inj₁ (lpst2 , lus)
  = unifyLPST (lsubSLPST lus (lpst2 ++ lpst))
              (lus ++ (lsubSSS lus as))
              (lsubSLSST lus at)
... | just (inj₂ st)
  = unifyLPST (subSTLPST st lpst)
              as
              (st :: (subSTLST st at))
```

Figura 6.16: Codificación tentativa para la unificación de listas de pares de tipos de canal

Al igual que antes esta definición no pasa la comprobación de terminación del compilador de Agda debido a que no es evidente que la recursión termine. La lista de pares de tipos de canal a ser unificados sobre la que hacemos la recursión a veces crece en la llamada recursiva y/o se le aplica una sustitución. Al igual que en la sección anterior utilizamos la teoría de recursión bien fundada, presentada en la sección 4.2.8, para codificar la demostración de la parada de este algoritmo y así pasar la comprobación de terminación. Ahora además de utilizar que los naturales son bien fundados bajo la relación $<$ (subsección 4.2.8), utilizamos también la biblioteca de inducción lexicográfica de Agda presentada en la subsección 4.2.8.

Para la demostrar de la parada del algoritmo de unificación definimos el predicado de accesibilidad particular `UniLPST-Acc`. Podemos ver en la figura 6.17 la defínción del predicado de accesibilidad, el cual definimos basándonos en el manejo dado a las listas de pares a ser unificadas en las reglas del algoritmo de unificación presentado en la figura 6.16.

En el Apéndice B.2.1 demostramos que todas las listas de pares de tipos de canal satisfacen este predicado de accesibilidad, demostrando así que no existen secuencias infinitas decrecientes. Para esto definimos tres funciones que mapean la lista de pares de tipos de canal en ternas de naturales, para luego demostrar que estas ternas decrecen mirando las reglas del predicado `UniLPST-Acc` de derecha a izquierda. Esto significa que cada constructor inductivo de este predicado se aplica sobre un argumento con tipo indexado por una lista de pares de tipos de canal cuya terna correspondiente es menor que la terna que corresponde a la lista que indexa el tipo del elemento construido. Dado que las ternas de naturales definen un orden bien fundado comprobamos entonces que para cualquier lista de pares de tipos *lpst* existe al menos un elemento de tipo `UniLPST-Acc lpst`, cuya construcción brinda una secuencia finita decreciente de listas de pares de tipos de canal. Dado que el predicado de accesibilidad fue definido en base al algoritmo de unificiación, justamente sobre esta secuencia

```
data UniLPST-Acc : List (SType × SType) → Set where
  uniLPST[] : UniLPST-Acc []
  uniLPST-noth : (pst : SType × SType)
    (lpst : List (SType × SType)) →
    unifyST pst ≡ nothing →
    UniLPST-Acc (pst :: lpst)
  uniLPST-justinj1 : (pst : SType × SType)
    (lpst : List (SType × SType)) →
    ∃ (λ lpst2 →
       ∃ (λ lus →
              unifyST pst ≡ just (inj₁ (lpst2 , lus)) ×
              UniLPST-Acc (lsubSLPST lus (lpst2 ++ lpst)))) →
    UniLPST-Acc (pst :: lpst)
  uniLPST-justinj2 : (pst : SType × SType)
    (lpst : List (SType × SType)) →
    ∃ (λ st → unifyST pst ≡ just (inj₂ st) ×
             UniLPST-Acc (subSTLPST st  lpst)) →
    UniLPST-Acc (pst :: lpst)
```

Figura 6.17: Predicado de accesibilidad

finita de listas de pares decrecientes trabaja el algoritmo de unificación.

Con la demostración completa del lema `allUnifyLPST-acc` que determina que todas las listas de pares de tipos de canal son accesibles, o sea satisfacen el predicado `UniLPST-Acc` definido en la figura 6.17, podemos ahora redefinir el algoritmo de unificación de listas de pares de tipos de canal `unifyLPST`. Utilizamos para esto la función auxiliar `unifyLPST-ac`, esta función recibe como argumento adicional una demostración de que la lista argumento es accesible para así hacer recursión primitiva sobre este parámetro extra. Vemos la codificación en la figura 6.18. Verificamos nuevamente al igual que en la unificación de listas de pares de tipos básicos que esta forma de introducir la recursión bien fundada de forma de garantizar la terminación no altera la estructura básica de la definición tentativa dada al comienzo de esta sección.

Al igual que como hicimos con la unificación de listas de pares de tipos básicos en la sección anterior presentamos ahora algunas definiciones y resultados útiles para la demostración de la corrección del algoritmo de unificación de listas de pares de tipos de canal, dejando algunas definiciones y resultados auxiliares en el Apéndice B.2. Debido al tamaño del trabajo se decidió recortar la demostración de corrección, de todas formas estimamos que la demostración faltante seguiría un patrón similar al ya seguido para demostrar la corrección de la unificación de tipos básicos. Presentamos en las próximas secciones algunos resultados parciales sí demostrados. A pesar de que no completamos la demostración de corrección la cual presentamos en la sección 6.2.5, sí hacemos uso de este resultado más adelante en la demostración de corrección del algoritmo de inferencia de tipos.

```
unifyLPST-ac : (l : List (SType × SType)) → UniLPST-Acc l →
                 List (ℕ × Sort) → List (ℕ × SType) →
                 Maybe (List (ℕ × Sort) × List (ℕ × SType))
unifyLPST-ac .[] uniLPST[] as at = just (as , at)
unifyLPST-ac .(pst :: lpst)
  (uniLPST-noth pst lpst punifSTpst≡nothing)
  as at = nothing
unifyLPST-ac .(pst :: lpst)
  (uniLPST-justinj1 pst lpst
                    (lpst2 , lus ,
                     (punifSTpst≡inj1lpst2lus ,
                      uniLSTAcclpst            )))
  as at = unifyLPST-ac (lsubSLPST lus (lpst2 ++ lpst))
                       uniLSTAcclpst
                       (lus ++ (lsubSSS lus as))
                       (lsubSLSST lus at)
unifyLPST-ac .(pst :: lpst)
  (uniLPST-justinj2 pst lpst
                    (st , (punifSTpst≡inj2st , uniLSTAcclpst)))
  as at = unifyLPST-ac (subSTLPST st lpst) uniLSTAcclpst as
                       (st :: (subSTLST st at))

unifyLPST : List (SType × SType) →
            Maybe (List (ℕ × Sort) × List (ℕ × SType))
unifyLPST ts = unifyLPST-ac ts (allUnifyLPST-acc ts) [] []
```

Figura 6.18: Algoritmo de unificación de listas de pares de tipos de canal

6.2.3. Unificador más General

Introducimos la propiedad de ser el unificador más general. Dado que el algoritmo presentado en este capítulo en realidad halla en algunos casos sustituciones tanto de tipos de canal como de tipos básicos, necesitamos expresar ahora que una pareja de sustituciones de tipos de básicos y de canal es el unificador más general. Presentamos a continuación algunas definiciones previas necesarias.

Vemos a continuación la definición de que una lista de pares de tipos de canal unifique bajo una pareja de sustituciones de tipos básicos y de canal, utilizamos en esta definición el predicado `All` presentado en la sección 4.2.5.

```
unifiesLPST : List (ℕ × Sort) → List (ℕ × SType) →
              List (SType × SType) → Set
unifiesLPST ss sst lpst
  = All (uncurry _≡_) (lsubSTLPST sst (lsubSLPST ss lpst))
```

Análogamente definimos cuando una pareja de sustituciones unifica otra sustitución.

```
unifiesSST : List (ℕ × Sort) → List (ℕ × SType) →
             List (ℕ × SType) → Set
```

```
unifiesSST ss sst sst'
  = All (λ p → lsubSSTPCST sst (lsubSST ss (varST (proj₁ p))) ≡
               lsubSSTPCST sst (lsubSST ss (proj₂ p)))
        sst'
```

Finalmente introducimos la definición de ser el unificador más general en la figura 6.20. Una sustitución de tipos básicos ss y una sustitución de tipos de canal sst son el unificador más general de una lista de pares de tipos de canal $lpst$; si ss y sst unifican $lpst$ y para cualesquiera otras dos sustituciones $ss2$ y $sst2$ de tipos básicos y de canal respectivamente, que también unifiquen $lpst$, se cumple que la pareja de sustituciones ss y sst es tan general como $ss2$ y $sst2$. Donde por "tan general" se entiende como que existe una sustitución de tipos básicos $ss3$ y de tipos de canal $sst3$ tales que para todo tipo de canal st se cumple que $(st\ ss2)\ st2 \equiv (((st\ ss1)\ sst1)\ ss3)\ sst3$.

```
_≤ST_ : List (ℕ × Sort) × List (ℕ × SType) →
        List (ℕ × Sort) × List (ℕ × SType) → Set
(ss1 , sst1) ≤ST (ss2 , sst2)
  = ∃ (λ sst3 →
      ∃ (λ ss3 →
        ((st : SType) →
           lsubSSTPCST sst2 (lsubSST ss2 st) ≡
           lsubSSTPCST sst3
                       (lsubSST
                          ss3 (lsubSSTPCST sst1
                              (lsubSST ss1 st))))))
```

Figura 6.19: Relación "tan general como"

```
mguST : List (ℕ × Sort) → List (ℕ × SType) →
        List (SType × SType) → Set
mguST ss sst lpst
  = (unifiesLPST ss sst lpst) ×
    ((ss2 : List (ℕ × Sort))(sst2 : List (ℕ × SType)) →
    unifiesLPST ss2 sst2 lpst →
  ((ss , sst) ≤ST (ss2 , sst2)))
```

Figura 6.20: Unificador más general

6.2.4. Propiedades de la Sustitución

En esta sección presentamos algunas propiedades de la sustitución. La primera propiedad ≡underSST establece que dada una variable n, dos tipos de canal $st1$ y $st2$, y una sustitución sst, si n y $st1$ tienen la misma imagen bajo la sustitución sustituciones sst, entonces los tipos $st2$ y $st2\ \langle x, s1\rangle$ también tienen la misma imagen bajo sst.

```
≡underSST : (n : ℕ)(st1 st2 : SType)(sst : List (ℕ × SType)) →
  lsubSSTPCST sst (varST n) ≡ lsubSSTPCST sst st1 →
```

```
  lsubSSTPCST sst st2 ≡ lsubSSTPCST sst (subSTST (n , st1) st2)
```

Esta propiedad se demuestra por inducción en el tipo de canal $st1$, y en su demostración se utilizan los resultados auxiliares `receiveS-lsubSSSTPCST`, `sendS-lsubSSSTPCST`, `receiveST-lsubSSSTPCST` y `sendST-lsubSSSTPCST` que determinan el comportamiento de los constructores `receiveS`, `sendS`, `receiveST` y `sendST` respectivamente bajo una sustitución sst. Las demostraciones de estos lemas auxiliares son por inducción en la sustitución.

```
receiveS-lsubSSSTPCST : (ls : List Sort)(sst : List (ℕ × SType))
  (st : SType) →
  receiveS ls (lsubSSTPCST sst st) ≡
                                lsubSSTPCST sst (receiveS ls st)

sendS-lsubSSSTPCST : (ls : List Sort)(sst : List (ℕ × SType))
  (st : SType) →
  sendS ls (lsubSSTPCST sst st) ≡
                                lsubSSTPCST sst (sendS ls st)

receiveST-lsubSSSTPCST : (sst : List (ℕ × SType))
  (st1 st2 : SType) →
  receiveST (lsubSSTPCST sst st1) (lsubSSTPCST sst st2) ≡
                              lsubSSTPCST sst (receiveST st1 st2)

sendST-lsubSSSTPCST : (sst : List (ℕ × SType))
  (st1 st2 : SType) →
  sendST (lsubSSTPCST sst st1) (lsubSSTPCST sst st2) ≡
                                lsubSSTPCST sst (sendST st1 st2)
```

Figura 6.21: Comportamiento de los constructores `receiveS`, `sendS`, `receiveST` y `sendST` bajo una sustitución

En la demostración de la propiedad ≡`underSST` también utilizamos el lema auxiliar `complSTlsubSTST`, el cual establece que aplicar una sustitución al complemento de un tipo de canal st es igual a primero aplicar la sustitución al tipo st y luego aplicar la operación complemento al resultado de la anterior sustitución. Podemos ver la demostración de esta propiedad en la figura 6.22. Esta es interesante por el uso de la propiedad de `foldr-fusion` de la operación `foldr` demostrada en la biblioteca `List.Properties` de Agda. En la figura 5.10 se puede ver que la definición de la función de sustitución `lsubSSTPCST` la implementamos utilizando la operación `foldr` de la función `subSTST`, de esta forma podemos ahora aprovechar la propiedad `foldr-fusion` definida en la figura 6.23, la cual determina que si se satisface que $\forall x\ y, h\ (f\ x\ y) \equiv g\ x\ (h\ y)$ entonces la función $h \circ (foldr\ f\ e)$ es una función equivalente punto a punto a la función $foldr\ g\ (h\ e)$. Entonces tomando $h = complST$, $e = st$, $g = f = subSTST$, la propiedad nos dice que $complST\ \circ (foldr\ subSTST\ st)$ es una función equivalente a $foldr\ subSTST\ (complST st)$ tal cual queremos demostrar. Solo no resta demostrar entonces que $\forall sst\ st, complST\ (subSTST\ sst\ st) \equiv subSTST\ sst\ (complST\ st)$, donde st es un tipo de canal y sst es una sustitución de tipos de canal, para satisfacer así las hipótesis de la propiedad de `foldr-fusion`. Demostramos este último resultado por inducción en el tipo de

canal *st*, se puede ver esta demostración en la figura 6.24. Esta demostración ejemplifica como en la mayor parte de este trabajo tratamos de aprovechar resultados genéricos de las operaciones de la biblioteca de Agda utilizadas.

```
complSTlsubSTST : (st : SType)(sst : List (ℕ × SType)) →
  lsubSSTPCST sst (complST st) ≡ complST (lsubSSTPCST sst st)
complSTlsubSTST st sst
  = sym ((foldr-fusion complST st complSTsubSTST) sst)
```

Figura 6.22: Irrelevancia del orden de aplicación entre la operación de complemento y la aplicación de una sustitución.

```
foldr-fusion : ∀ {a b c}{A : Set a}{B : Set b}{C : Set c}
  (h : B → C) {f : A → B → B} {g : A → C → C} (e : B) →
  (∀ x y → h (f x y) ≡ g x (h y)) →
  h ∘ foldr f e ≗ foldr g (h e)
```

Figura 6.23: Propiedad `foldr-fusion` de la operación de `foldr` de listas

```
complSTsubSTST : (sst : ℕ × SType)(st : SType) →
  complST (subSTST sst st) ≡ subSTST sst (complST st)
complSTsubSTST sst (receiveS ls st)
  = cong (sendS ls) (complSTsubSTST sst st)
complSTsubSTST sst (sendS ls st)
  = cong (receiveS ls) (complSTsubSTST sst st)
complSTsubSTST sst (receiveST st1 st2)
  = cong (sendST (subSTST sst st1)) (complSTsubSTST sst st2)
complSTsubSTST sst (sendST st1 st2)
  = cong (receiveST (subSTST sst st1)) (complSTsubSTST sst st2)
complSTsubSTST (n , st) end = refl
complSTsubSTST (m , st) (varST n)  with m ≟ n
complSTsubSTST (.n , st) (varST n) | yes refl = refl
complSTsubSTST (m , st) (varST n)  | no n≢m = refl
complSTsubSTST (m , st) (varSTCompl n) with m ≟ n
complSTsubSTST (.n , st) (varSTCompl n) | yes refl
  = complSTidempotent st
complSTsubSTST (m , st) (varSTCompl n)  | no n≢m = refl
```

Figura 6.24: Irrelevancia del orden de aplicación entre la operación `subSTST` y `complST`

Como es esperable, si el conjunto de variables en el dominio de una sustitución y el conjunto de variables que aparece en un tipo de canal son disjuntos, entonces aplicar la sustitución al tipo de canal no tiene efecto alguno. Esta propiedad se demuestra por inducción en la sustitución.

```
≡disjoinVarST : (st : SType)(sst : List (ℕ × SType)) →
  disjoin (varsST st) (dom sst) → st ≡ lsubSSTPCST sst st
```

Definición Una sustitución *st* es *idempotente* si para cualquier tipo básico *s* se cumple que $s\ st \equiv (s\ st)\ st$.

En la figura 6.25 podemos ver esta definición extendida a listas de sustituciones de tipos de canal.

```
IdempotentSTST : (st : List (ℕ × SType)) → Set
IdempotentSTST sst = (s : SType) →
  lsubSSTPCST sst (lsubSSTPCST sst s) ≡ lsubSSTPCST sst s
```

Figura 6.25: Sustitución idempotente

Vemos que este formato de definición no aporta información explícita de cuando se cumple la idempotencia para una sustitución, por tanto introducimos la definición de la figura 6.26 que explicita las condiciones necesarias para que se cumpla esta propiedad, esto es cuando las variables de tipo que aparecen en el dominio y codominio de la sustitución son disjuntos.

```
data IdemST : List (ℕ × SType) → Set where
  []idemST : IdemST []
  ::idemST : (sst : List (ℕ × SType))(n : ℕ)(s : SType) →
             IdemST sst →
             disjoin (varsST s) (dom ((n , s) :: sst)) →
             n ∉ dom sst → n ∉ varsSSSTS sst →
             IdemST ((n , s) :: sst)
```

Figura 6.26: Definición explícita de las condiciones bajo las cuales se cumple la propiedad de idempotencia

La próxima propiedad establece que si una sustitución de tipos de canal *sst* es idempotente, entonces el conjunto de las variables de tipo en el tipo *st ss* para un tipo de canal *st* cualquiera, y el dominio de *ss* son disjuntos.

```
disj-idemST : (st : SType)(sst : List (ℕ × SType)) →
  IdemST sst →
  disjoin (varsST (lsubSSTPCST sst st)) (dom sst)
```

Demostramos ahora utilizando los dos resultados anteriores que si una sustitución satisface el predicado `IdemST` entonces también satisface el predicado `IdempotentSTST`.

```
IdemToIdempotentST : (sst : List (ℕ × SType)) →
  IdemST sst → IdempotentSTST sst
IdemToIdempotentST sst idemsst st
  = sym (≡disjoinVarST (lsubSSTPCST sst st) sst
                       (disj-idemST st sst idemsst))
```

Figura 6.27: Equivalencia entre las dos definiciones de idempotencia vistas

6.2.5. Corrección del Algoritmo de Unificación

En esta sección definimos el resultado de corrección de la unificación. Tal cual comentamos anteriormente no completamos su demostración pero hacemos uso de este postulado en la demostracción de la corrección del algoritmo de inferencia de tipos presentado en el próximo capítulo. Estimamos que la demostración restante seguiría un formato similar al presentado en la sección 6.1.5. Al ser la estructura de datos de los tipos de canal más grande que la de tipos básicos el trabajo es más grande, pero creemos que no aumenta la complejidad de este. Aunque podríamos obviar la demostración de que la sustitución hallada es la más general, considerando que en no hacemos uso de este resultado al no demostrar que el algoritmo de inferencia de tipos halla el tipado más general posible.

```
postulate
  sessionTypeUnificationSoundness : (ss : List (ℕ × Sort))
    (sst : List (ℕ × SType))(lpst : List (SType × SType)) →
    unifyLPST lpst ≡ just (ss , sst) →
    unifiesLPST ss sst lpst
```

Figura 6.28: Corrección de la unificación

Capítulo 7

Formalización de la Inferencia de Tipos de Sesión

En este capítulo presentamos la formalización en Agda del algoritmo de inferencia de tipos de la sección 3, utilizando las formalizaciones del sistema de tipos y la unificación dados en los dos capítulos anteriores. El algoritmo de inferencia recibe un proceso P e infiere los contextos de esquemas de tipos más generales Φ, Γ y Δ bajo los cuales se satisface el juicio $\Phi; \Gamma \vdash P \triangleright \Delta$ introducido en la sección 3.3, si es que esto es posible. Demostramos formalmente la solidez del algoritmo con respecto al sistema de tipos, resultado que ya fue demostrado (informalmente) en la sección 3.5.5. Esto es, probamos que si el algoritmo de inferencia infiere los contextos Φ, Γ y Δ para un proceso P, entonces necesariamente se satisface el juicio $\Phi; \Gamma \vdash P \triangleright \Delta$.

7.1. Algoritmo de Inferencia en Procesos

El algoritmo de inferencia de tipos se implementa usando recursión primitiva en la estructura del proceso P dado como argumento, aplicando según corresponda las reglas del algoritmo presentado en la sección 3.4. Recordamos que en la figura 5.7 definimos los contextos como listas de pares mediante las estructuras de datos: `Sort`, `Types` y `STypes` respectivamente. Para codificar la posibilidad de que no existan contextos de tipado para el proceso argumento utilizamos el tipo de datos `Maybe` presentado en la sección 4.2.2.

Para la codificación de la extensión a listas de expresiones del algoritmo de inferencia de tipos en expresiones presentado en la figura 3.5 reutilizamos la función `extractSorts` presentada en la sección 5.5.

En las próximas secciones detallamos algunas de las funciones auxiliares utilizadas en la implementación de este algoritmo.

Join

Codificamos el operador $\bowtie$ utilizado en la regla [ConcInf] mediante la función `join`, la cual implementamos genéricamente para ser utilizada en cualquier

tipo de contextos en el módulo `Enviroment` introducido en la sección 5.5. Esta función, tal cual se puede ver en la figura figura 7.1, agrupa en pares la información de tipos para nombres en común entre dos contextos dados.

```
join : List (K × D) → List (K × D) → List (D × D)
join [] e = []
join ((k , d1) :: e1) e2 with e2 •p k
... | nothing = join e1 e2
... | just d2 = (d1 , d2) :: (join e1 e2)
```

Figura 7.1: Función *join*

Unión Disjunta de Contextos de Canales

Presentamos la codificación del operador $\otimes$ en la figura 7.2. Este operador une dos contextos si es que sus dominios son disjuntos, permitiendo sólo información duplicada para un mismo canal si en al menos uno de los contextos esta información es `end`. Si los contextos no cumplen esta condición entonces devuelve `nothing`.

```
_⊗_ : STypes → STypes → Maybe STypes
[] ⊗ Δ = just Δ
((x , t) :: Δ1) ⊗ Δ2 with t ≟ST end
... | yes _ =  Δ1 ⊗ Δ2
... | no  _ with (Δ2 •c x) ≟ST end
...          | no  _ = nothing
...          | yes _ with Δ1 ⊗ Δ2
...                  | nothing = nothing
...                  | just Δ  = just (Δ <+c (x , t))
```

Figura 7.2: Operación $\otimes$

Variables Frescas

El algoritmo de inferencia de expresiones presentado en 3.5 requiere hallar variables de tipos básicos frescas en la regla [NameInfV]. Presentamos la forma en que resolvemos la búsqueda de variables de tipo frescas necesarias para inferir el tipo más general de un proceso.

Resolvimos la búsqueda de variables de tipos básicos frescas para contextos dados Φ, Δ y Γ, sumando uno al mayor identificador de variable de tipo básico hallado en estos contextos. Llamamos `freshVarS` a esta función.

La regla [SendInf] es utilizada para procesos de la forma $k?(\tilde{ln})$ `in` P, para la cual necesitamos una variable fresca para cada nombre $\tilde{ln}$ no definido en Φ. Para esto generamos tantas variables frescas como podríamos llegar a necesitar, o sea, generamos de antemano un número de variables frescas igual al largo de $\tilde{ln}$, mediante la función `gen`: `(gen (length ln) (freshVarS Φ Δ`

Γ)) devuelve una lista de variables de tipos básicos con identificadores crecientes unitariamente tan larga como su primer argumento, cuya numeración comienza a partir del segundo argumento.

Por otro lado, el propio algoritmo de inferencia en procesos requiere variables de tipos de canal frescas en la regla [ThrInf], lo cual resolvimos de forma análoga a la ya explicada. Identificamos a esta función con el nombre `freshVarST`. La única diferencia es que como el contexto Φ no contiene variables de tipos de canal, esta función solo recibe como argumentos los contextos de canales Δ y de puertos Γ.

En la regla [Conc] debemos hallar dos sustituciones, una de tipos básicos y otra de tipos de canal, que renombren las variables en una terna de contextos de forma de hacerlas disjuntas con otra terna de contextos. Utilizamos las funciones `renameS` y `renameST` para generar respectivamente estas sustituciones, las cuales presentamos en la figura 7.3, donde las funciones `varSocurr` y `varSTocurr` devuelven las listas de identificadores de variables de tipos básicos y de canal respectivamente que ocurren en los contextos dados como argumento.

```
renameS : ℕ → Sorts → STypes → Types → List (ℕ × Sort)
renameS n Φ Δ Γ
  = Data.List.zip (varSocurr Φ Δ Γ)
                  (map (λ x → varS (x + n)) (varSocurr Φ Δ Γ))

renameST : ℕ → STypes → Types → List (ℕ × SType)
renameST n Δ Γ
  = Data.List.zip (varSTocurr Δ Γ)
                  (map (λ x → varST (x + n)) (varSTocurr Δ Γ))
```

Figura 7.3: Sustituciones de renombre de variables de tipos

Dadas dos ternas de contextos $(\Phi_1, \Delta_1, \Gamma_1)$ y $(\Phi_2, \Delta_2, \Gamma_2)$, la expresión `renameS (maxOcurr (varSocurr Φ1 Δ1 Γ1 )) Φ2 Δ2 Γ2` devuelve una posible sustitución π_s que hace disjuntos los conjuntos de ocurrencias de variables de tipos básicos de la primer terna, y la terna resultante de aplicar la sustitución π_s a cada uno de los contextos de la segunda terna. Análogamente `renameST (maxOcurr (varSTocurr Δ1 Γ1)) Δ2 Γ2` devuelve una posible sustitución π_{st} de tipos de canal que hace disjuntas las variables de tipos de canal.

Presentamos a continuación la codificación final del algoritmo de inferencia de tipos en procesos. Esta codificación es casi directa dadas las reglas del algoritmo ya presentadas en la sección 3.6. Las únicas salvedades son: el manejo de de las variables libres ya explicadas, y el manejo de las sustituciones e unificación en los esquemas de tipos. En la figura 3.6 abstrajimos la unificación de esquemas en un único operador $\sqcup$, el cual devuelve una sustitución θ que reúne tanto las sustituciones de tipos básicos como de canales. En nuestra codificación manejamos estos conceptos de forma separada. Además para la aplicación de las sustituciones a contextos que antes denotábamos simplemente yuxtaponiendo un contexto y una sustitución, ahora debemos utilizar las distintas funciones auxiliares de sustitución que aplican según el tipo de contexto y sustitución dadas como argumento (presentadas en 5.10).

Detallamos ahora el caso correspondiente a la codificación de la regla [ConcInf].

En el capítulo 6 presentamos dos algoritmos de unificación que devuelven dos clases de sustituciones. Por este motivo la codificación de la regla [ConcInf] cambia al tener que manejar por separado la unificación de tipos básicos y de canales. Sean las ternas $(\Phi_1, \Delta_1, \Gamma_1)$ y $(\Phi_2, \Delta_2, \Gamma_2)$ el resultado de aplicar recursivamente el algoritmo de inferencia a los procesos que componen la composición en paralelo respectivamente. Entonces aplicamos primero el algoritmo de unificación a la concatenación las parejas de primeras y segundas componentes del `join` de Γ_1 y la aplicación de dos sustituciones de renombres, de tipos básicos y de canal, a Γ_2. Esto nos da una pareja de sustituciones ss y sst, de tipos básicos y de canales respectivamente, que unifican los esquemas de tipos de puertos en común en los contextos Γ_1 y Γ_2 renombrado. Ahora necesitamos unificar la información de esquemas de tipos básicos en Δ_1 ss y Δ_2' ss, donde Δ_2' es el contexto Δ_2 renombrado. En caso de ser exitosa esta unificación tenemos entonces un sustitución de tipos básicos $ss2$ que los unifica. A partir de este punto el resto de la codificación es directa.

```
typeInf : Process → Maybe (Sorts × STypes × Types)
typeInf inact = just ([] , [] , [])
typeInf (accept n c p) with typeInf p
... | nothing = nothing
... | (just (Φ , Δ , Γ)) with Any.any (MT._≟×1_ n) Γ
...   | yes n∈Γ
  with unifyLPST ((fstPar (Γ •∈p n∈Γ) , Δ •c (var c)) ::
                  (sndPar (Γ •∈p n∈Γ) , complST (Δ •c (var c))) :: [])
... | nothing           = nothing
... | (just (ss , sst))
  = just (lsubSSS ss Φ ,
          (lsubSTLPCST sst (lsubSLPCST ss Δ)) \\c (var c) ,
          lsubSTLPNT sst (lsubSLPNT ss Γ))
typeInf (accept n c p)
    | (just (Φ , Δ , Γ))
      | no  n∉Γ
  = just (Φ , Δ \\c (var c) , (Γ <+p (n ,  par α (complST α))))
  where α = Δ •c (var c)
typeInf (request n c p) with typeInf p
... | nothing = nothing
... | just (Φ , Δ , Γ) with Any.any (MT._≟×1_ n) Γ
...   | yes n∈Γ
  with unifyLPST ((sndPar (Γ •∈p n∈Γ) , Δ •c (var c)) ::
                  (fstPar (Γ •∈p n∈Γ) , complST (Δ •c (var c))) :: [])
... | nothing = nothing
... | just (ss , sst)
  = just (lsubSSS ss Φ ,
          (lsubSTLPCST sst (lsubSLPCST ss Δ)) \\c (var c) ,
          lsubSTLPNT sst (lsubSLPNT ss Γ))
typeInf (request n c p)
    | just (Φ , Δ , Γ)
      | no  n∉Γ
  = just (Φ , Δ \\c (var c) , (Γ <+p (n , par (complST α) α)))
  where α = Δ •c (var c)
typeInf (catch c cv p)  with c ≟c (var cv)
... | yes _ = nothing
```

```
... | no  _  with typeInf p
...   | nothing          = nothing
...   | just (Φ , Δ , Γ)
  = just (Φ ,
          (Δ <+c (c , receiveST (Δ •c (var cv))
                                (Δ •c c)))     \\c (var cv),
          Γ)
typeInf (throw c1 c2 p) with c1 ≟c c2
... | yes _ = nothing
... | no _  with typeInf p
...   | nothing          = nothing
...   | just (Φ , Δ , Γ) with Any.any (MST._≟×1_ c2) Δ
...     | yes _ = nothing
...     | no  _
  = just (Φ , (Δ <+c (c1 , sendST α (Δ •c c1)) <+c (c2 , α)) , Γ)
  where α = varST (freshVarST Δ Γ) -- varST (getNumCh c2)
typeInf (receive c ln p) with typeInf p
... | nothing          = nothing
... | just (Φ , Δ , Γ)
  = just (Φ \\[]s ln ,
          (Δ <+c (c , receiveS
                       (proj₁ (extractSorts
                                (map name ln)
                                Φ
                                (gen (length ln) (freshVarS Φ Δ Γ))))
                       (Δ •c c))) ,
          Γ)
typeInf (send c le p) with typeInf p
... | nothing          = nothing
... | just (Φ , Δ , Γ)
  = just (Φ ++ (proj₂
                 (extractSorts le Φ
                   (gen (length le) (freshVarS Φ Δ Γ)))) ,
          Δ <+c (c , sendS
                       (proj₁
                         (extractSorts le Φ
                           (gen (length le) (freshVarS Φ Δ Γ))))
                       (Δ •c c)) ,
          Γ)
typeInf (pipe p1 p2) with typeInf p1 | typeInf p2
... | nothing | _ = nothing
... | _ | nothing = nothing
... | just (Φ1 , Δ1 , Γ1) | just (Φ2 , Δ2 , Γ2)
  with unifyLPST
         ((map (map× fstPar fstPar)
               (ET.join Γ1
               (lsubSTLPNT (renameST
                             (maxOcurr (varSTocurr Δ1 Γ1))
                             Δ2 Γ2)
                           (lsubSLPNT
                             (renameS (maxOcurr (varSocurr Φ1 Δ1 Γ1))
                                      Φ2 Δ2 Γ2) Γ2))))
       ++ (map (map× sndPar sndPar)
```

```
              (ET.join Γ1
              (lsubSTLPNT (renameST
                            (maxOcurr (varSTocurr Δ1 Γ1))
                            Δ2 Γ2)
                          (lsubSLPNT
                            (renameS (maxOcurr (varSocurr Φ1 Δ1 Γ1))
                                     Φ2 Δ2 Γ2) Γ2)))))
... | nothing         = nothing
... | just (ss , sst)
  with unifyLS
         (ES.join (lsubSSS ss Φ1)
                  (lsubSSS ss
                  (lsubSSS (renameS (maxOcurr (varSocurr Φ1 Δ1 Γ1))
                                    Φ2 Δ2 Γ2)
                           Φ2)))
... | nothing = nothing
... | just ss2
  with (lsubSLPCST ss2 (lsubSTLPCST sst (lsubSLPCST ss Δ1))) ⊗
       (lsubSLPCST ss2
         (lsubSTLPCST sst
           (lsubSLPCST ss
             (lsubSTLPCST
               (renameST (maxOcurr (varSTocurr Δ1 Γ1)) Δ2 Γ2)
               (lsubSLPCST (renameS (maxOcurr (varSocurr Φ1 Δ1 Γ1))
                                    Φ2 Δ2 Γ2)
                           Δ2)))))
... | nothing = nothing
... | just Δ
  = just ((lsubSSS ss2 (lsubSSS ss Φ1)) ∪s
          (lsubSSS ss2
            (lsubSSS ss
              (lsubSSS (renameS (maxOcurr (varSocurr Φ1 Δ1 Γ1))
                                Φ2 Δ2 Γ2)
                       Φ2))) ,
          Δ ,
          (lsubSLPNT ss2
            ((lsubSTLPNT sst (lsubSLPNT ss Γ1)) ∪p
             (lsubSTLPNT sst
               (lsubSLPNT ss
                 (lsubSTLPNT (renameST (maxOcurr (varSTocurr Δ1 Γ1))
                                       Δ2 Γ2)
                             (lsubSLPNT
                               (renameS (maxOcurr (varSocurr Φ1 Δ1 Γ1))
                                        Φ2 Δ2 Γ2)
                               Γ2)))))))
typeInf (vn n p) with typeInf p
... | nothing = nothing
... | just (Φ , Δ , Γ) = just (Φ , Δ , Γ \\p n)
typeInf (vc c p) = typeInf p
```

7.2. Aplicación del Algoritmo de Inferencia

A continuación mostramos el resultado de aplicar el algoritmo de inferencia al proceso presentado en la figura 2.9. En la figura 2.10 comprobamos que este proceso tipa correctamente bajo $\Delta = \emptyset$, $\Phi = \emptyset$ y $\Gamma = \{a : \langle \alpha, \overline{\alpha} \rangle, b : \langle \beta, \overline{\beta} \rangle\}$, con $\alpha = ![\beta]\,;\mathtt{end}$, y $\beta = ![\mathtt{nat}, \mathtt{bool}]; \mathtt{end}$.

```
just ([] ,
      [] ,
      (1 , par (sendST (sendS (nat :: boolean :: []) end) end)
               (receiveST (sendS (nat :: boolean :: []) end) end))
      ::
     (5 , par (sendS (nat :: boolean :: []) end)
              (receiveS (nat :: boolean :: []) end))
      :: [])
```

En la figura 5.2 dimos la codificación de este proceso. Vemos en esta codificación que los puertos de comunicación a y b son codificados mediante los naturales 1 y 5 respectivamente. Así comprobamos que el algoritmo de inferencia de tipos devuelve efectivamente el mismo resultado comprobado manualmente.

7.3. Solidez del Algoritmo de Inferencia de Tipos

En esta sección presentamos la demostración de solidez del algoritmo de inferencia de tipos. Esto es, que si el algoritmo infiere los contextos Φ, Γ y Δ para un proceso P, entonces necesariamente se satisface el juicio $\Phi; \Gamma \vdash P \triangleright \Delta$. Podemos ver en la figura 7.4 la codificación de este enunciado.

```
typeInferenceSoundness : (p : Process)(Φ : Sorts)
  (Δ : STypes)(Γ : Types) →
  typeInf p ≡ just (Φ , Δ , Γ) → (Φ , Γ) ⊢ p ▷ Δ
```

Figura 7.4: Enunciado de solidez del algoritmo de inferencia de tipos

La demostración de este resultado es por inducción en el proceso p. En la sección 7.3.5 describimos los distintos casos de la inducción, pero antes presentamos algunos lemas útiles para esta demostración.

7.3.1. Contextos Γ y Φ sin Primeras Proyecciones Repetidas

El resultado `typeInferenceΓNoRep` presentado en la figura 7.5 establece que si el algoritmo de inferencia de tipos infiere un contexto de tipos de canal Γ, entonces este contexto no tiene tuplas con su primera proyección idéntica. Demostramos este resultado haciendo inducción primitiva en el proceso p. La

demostración es directa y utiliza lemas que establecen que todas las operaciones hechas a los contextos en el algoritmo de inferencia de tipos preservan la propiedad de no contener tuplas con su primera proyección repetida.

```
typeInferenceΓNoRep : (p : Process)(Φ : Sorts)
  (Δ : STypes)(Γ : Types) →
  typeInf p ≡ just (Φ , Δ , Γ) → NRL.NoRep Γ
```

Figura 7.5: Contexto Γ inferido por el algoritmo de inferencia de tipos no contiene tuplas con su primera proyección idéntica

El lema `typeInferenceΦNoRep` presentado a continuación es análogo al anterior estableciendo ahora que el contexto de tipos básicos Φ inferido por el algoritmo de inferencia de tipos no tiene tuplas con su primera proyección idéntica. Esta demostración es también por inducción primitiva en el proceso p y utiliza los mismos lemas utilizados en el anterior resultado que establecen que las operaciones hechas a los contextos no introducen tuplas con su primera proyección repetida.

```
typeInferenceΦNoRep : (p : Process)(Φ : Sorts)
  (Δ : STypes)(Γ : Types) →
  typeInf p ≡ just (Φ , Δ , Γ) → NRLs.NoRep Φ
```

Figura 7.6: Contexto Φ inferido por el algoritmo de inferencia de tipos no contiene tuplas con su primera proyección idéntica

7.3.2. Permutaciones de los Contextos Φ y Γ bajo las operaciones de ∪ y ∩

Introducimos el resultado para los contextos de puertos Γ que, dadas una lista de sustituciones de tipos básicos ss, una lista de sustituciones de tipos de canales sst, y dos contextos de tipos de puertos Γ1 y Γ2 cualesquiera, si la información de puertos en común entre los anteriores contextos unifica mediante las sustituciones ss y sst dadas, y los contextos no tienen tuplas con su primera proyección repetida, entonces establece que $((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2\ ss)\ sst)$ es una permutación de $((\Gamma 2\ ss)\ sst) \cup ((\Gamma 1\ ss)\ sst)$.

Para demostrar el anterior resultado hacemos uso de un lema previo que establece que bajo las mismas premisas se cumple que $((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst)$ es una permutación de $((\Gamma 2\ ss)\ sst) \cap ((\Gamma 1\ ss)\ sst)$. A su vez para demostrar este lema usamos el resultado `intersect-perm` del módulo `Enviroment` el cual determina que para dos contextos cualesquiera, si todas las tuplas que coinciden en su primera proyección también coinciden en su segunda proyección, entonces cambiar el orden de los argumentos en la operación de intersección de contextos devuelve permutaciones del mismo contexto. Podemos ver en la figura 7.7 la codificación de este resultado.

Utilizamos el anterior resultado instanciando $e1 = (\Gamma 1\ ss)\ sst$ y $e2 = (\Gamma 2\ ss)\ sst$. Solamente resta demostrar que la información en común de los

```
intersect-perm : (e1 e2 : List (K × D)) →
  ((k : K)(d1 d2 : D) →
          (k , d1) ∈ e1 → (k , d2) ∈ e2 → d1 ≡ d2) →
  permutation (e2 ∩₁ e1) (e1 ∩₁ e2)
```

Figura 7.7: Cambiar el orden de los argumentos en la intersección de contextos devuelve permutaciones del mismo contexto

contextos ($\Gamma 1\ ss)\ sst$ y ($\Gamma 2\ ss)\ sst$) es idéntica, para lo cual utilizamos la hipótesis de que la información en común entre los contextos $\Gamma 1$ y $\Gamma 2$ fue unificada dando como resultado las sustituciones ss y sst, utilizando entonces el resultado presentado en la sección 6.2.5 que establece la corrección de la unificación. Queda así demostrado que:

$$((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst) \text{ es una permutación de } ((\Gamma 2\ ss)\ sst) \cap ((\Gamma 1\ ss)\ sst) \tag{7.1}$$

Otros tres lemas útiles demostrados acerca de las permutaciones de contextos establecen que para cualesquiera contextos $e1$, $e2$ y $e3$ se cumple:

1. $e1 +\!\!+ e2$ es una permutación de $e2 +\!\!+ e1$
2. si $e1$ es una permutación de $e3$ entonces $e1 +\!\!+ e2$ es una permutación de $e3 +\!\!+ e2$
3. la permutación es una relación de equivalencia

También utilizamos la asociatividad de la operación de concatenación en listas demostrado en la biblioteca estándar de Agda.

$$e1 +\!\!+ (e2 +\!\!+ e3) \equiv (e1 +\!\!+ e2) +\!\!+ e3 \tag{7.2}$$

Podemos ahora usar los anteriores resultados para terminar la demostración planteada al comienzo de la sección mediante el siguiente razonamiento, donde utilizamos $\sim$ para denotar la relación de permutación.

$$((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2\ ss)\ sst) \equiv$$

$$\{\ \text{Por definición de la operación } \cup\ \}$$

$$\underbrace{(((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst))}_{a1} +\!\!+ \underbrace{((((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst)) +\!\!+ (((\Gamma 2\ ss)\ sst) \backslash ((\Gamma 1\ ss)\ sst)))}_{a2} \sim$$

$$\{\ \text{Por el lema 1 de las permutaciones tomando } e1 = a1 \text{ y } e2 = a2\ \}$$

$$\left(\overbrace{(((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst))}^{a1} +\!\!+ \overbrace{(((\Gamma 2\ ss)\ sst) \backslash ((\Gamma 1\ ss)\ sst))}^{a2} \right) +\!\!+ \underbrace{(((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst))}_{b2} \sim$$

$$\left\{\begin{array}{c}\text{Aplicamos el lema 2 con } e1 = a1 +\!\!+ a2,\ e2 = b2 \text{ y } e3 = (((\Gamma 2\ ss)\ sst) \cap ((\Gamma 1\ ss)\ sst)) +\!\!+ a2,\\ \text{para utilizar el anterior lema necesitamos saber que } e1 \text{ es una permutación de } e3, \text{ lo cual}\\ \text{demostramos aplicando otra vez el mismo lema 2 pero ahora con } e1 = a1,\ e2 = a2\\ \text{y } e3 = (((\Gamma 2\ ss)\ sst) \cap ((\Gamma 1\ ss)\ sst)).\\ \text{A su vez para utilizar el anterior resultado necesitamos tener que } e1 \text{ es una permutación de } e3,\\ \text{lo cual se cumple por el lema 7.1.}\end{array}\right\}$$

$$\left(\overbrace{(((\Gamma 2\ ss)\ sst) \cap ((\Gamma 1\ ss)\ sst))}^{a1} +\!\!+ \overbrace{(((\Gamma 2\ ss)\ sst) \backslash ((\Gamma 1\ ss)\ sst))}^{a2}\right) +\!\!+ \underbrace{(((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst))}_{b2} \sim$$

$$\left\{\begin{array}{c}\text{Por el lema 2 de permutaciones con } e1 = a1 +\!\!+ a2,\ e2 = b2 \text{ y } e3 = a2 +\!\!+ a1,\\ \text{para utilizar este lema necesitamos tener que } e1 \text{ es una permutación de } e3,\\ \text{lo cual se demuestra aplicando el lema 1 de permutaciones con } e1 = a1 \text{ y } e2 = a2,\end{array}\right\}$$

$$((((\Gamma 2\ ss)\ sst) \backslash ((\Gamma 1\ ss)\ sst)) +\!\!+ (((\Gamma 2\ ss)\ sst) \cap ((\Gamma 1\ ss)\ sst))) +\!\!+ (((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst)) \sim$$

$$\left\{\ \text{Por el lema 7.2 de asociatividad de la concatencación}\ \right\}$$

$$(((\Gamma 2\ ss)\ sst) \backslash ((\Gamma 1\ ss)\ sst)) +\!\!+ ((((\Gamma 2\ ss)\ sst) \cap ((\Gamma 1\ ss)\ sst)) +\!\!+ (((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst))) \equiv$$

$$\left\{\ \text{Por definición de la operación } \cup\ \right\}$$

$$((\Gamma 2\ ss)\ sst) \cup ((\Gamma 1\ ss)\ sst)$$

Finalmente dado que la relación de permutación $\sim$ es una relación de equivalencia aplicamos la clausura reflexo-transitiva para finalmente obtener el resultado desado. En la figura 7.8 mostramos la codificación del anterior razonamiento, donde los lemas de permutaciones se mapean en la codificación de la siguiente mantera; `permutation-++` se corresponde con el lema 1 , `permutation-++-cons` se corresponde con el lema 2 y `permutation-trans` se corresponde con el resultado de transitividad de la relación de permutación. Por otro lado `perm-`$\Gamma 2 \Theta \cap \Gamma 1 \theta$ codifica el resultado 7.1 y `assoc` codifica el resultado 7.2 de asociatividad de la concatenación.

Siguiendo un razonamiento análogo al anterior demostramos un resultado equivalente pero ahora para los contextos de tipos básicos Φ. Esto es, dada una lista de sustituciones de tipos básicos ss, y dos contextos de tipos básicos $\Phi 1$ y $\Phi 2$, si la información de tipos básicos en común entre los anteriores contextos unifica mediante la sustitución ss dada, y los contextos no tienen tuplas con su primera proyección repetida, entonces:

$$(\Phi 1\ ss) \cup (\Phi 2\ ss) \text{ es una permutación de } (\Phi 2\ ss) \cup (\Phi 1\ ss) \qquad (7.3)$$

7.3.3. Lemas Auxiliares del Juicio de Tipado

Presentamos en esta sección cuatro resultados auxiliares con respecto al juicio de tipado. El primer resultado establece que dadas una lista de sustituciones de tipos básicos ss, una lista de sustituciones de tipos de canales sst, un contexto de tipos básicos Φ, dos contextos de tipos de puertos $\Gamma 1$ y $\Gamma 2$, un contexto de tipos de canal Δ y un proceso p cualesquiera, si se cumple que el proceso p satisface el juicio de tipado para los contextos Φ, $\Gamma 1$ y Δ, y también que los contextos $\Gamma 1$ y $\Gamma 2$ no tienen información de puertos repetida, entonces el proceso p también satisface el juicio de tipado para los contextos $\Phi\ ss$, $((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2\ ss)\ sst)$ y $(\Delta\ ss)\ sst$, o sea se satisface la expresión:

$$(\Phi\ ss)\ sst, ((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2\ ss)\ sst) \vdash p\ \triangleright (\Delta\ ss)\ sst \tag{7.4}$$

Damos a continuación una descripción de la codificación de la demostración de este resultado. Comenzamos aplicando el resultado de conservación del juicio de tipado bajo sustituciones presentado en la sección 5.6.5, dado que por hipótesis se cumple el juicio $\Phi, \Gamma 1 \vdash p\ \triangleright \Delta$. Mediante este lema tenemos entonces que también se satisface el juicio $(\Phi\ ss)\ sst, (\Gamma 1\ ss)\ sst \vdash p\ \triangleright (\Delta\ ss)\ sst$. Ahora utilizamos el resultado auxiliar que establece que para cualquier par de contextos $e1$ y $e2$ se cumple que $(e1 \backslash e2) +\!\!+ (e1 \cap e2)$ es una permutación de $e1$. Instanciando este resultado con $e1 = (\Gamma 1\ ss)\ sst$ y $e2 = (\Gamma 2\ ss)\ sst$ podemos entonces utilizarlo como premisa de la regla `Permutation-env` del juicio de tipado que nos permite permutar los contextos en este juicio, llegando entonces a que se satisface el juicio:

$$(\Phi\ ss)\ sst, (((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst)) +\!\!+ (((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst)) \vdash p\ \triangleright (\Delta\ ss)\ sst$$

Intentamos ahora utilizar el lema de weakening derecho para contextos de tipos de puertos, presentado en la sección 5.6.4, el cual nos permite agregar información al contexto de tipos de puertos siempre y cuando no se agregue información de puertos ya existentes, y la información agregada no contenga información repetida. Agregamos entonces al contexto de tipos de puertos la siguiente información $((\Gamma 2\ ss)\ sst) \backslash (((\Gamma 1\ ss)\ sst)$, utilizamos las propiedades `prop1_[]`$\times$ y `prop2_[]`$\times$ del módulo `Enviroment` las cuales establecen que si para dos contextos $e1$ y $e2$, y para un puerto n tal que para algún tipo de puertos t se cumple que la tupla $\langle n, t\rangle \in e2 \backslash e1$ y ninguno de los contextos posee información de puertos repetida, entonces no existe $t2$ tipo de puerto alguno para el cual se cumpla que $\langle n, t2\rangle \in e1 \backslash e2$ o $\langle n, t2\rangle \in e1 \cap e2$. Para hacer uso de las anteriores dos propiedades antes debemos demostrar que tanto $(((\Gamma 1\ ss)\ sst)$ como $(((\Gamma 2\ ss)\ sst)$ no contienen información de puertos repetida. Para esto utilizamos las premisas de que $\Gamma 1$ y $\Gamma 2$ no tienen información repetida y el lema `map-noRep` que determina que aplicar una lista de sustituciones a un contexto no introduce información repetida. Utilizamos ahora el resultado $\notin$`++` del módulo `Enviroment` que demuestra que si un elemento no pertenece a dos listas tampoco pertenece a su concatenación para obtener que entonces no existe ningún tipo de puerto $t2$ que cumpla que $\langle n, t2\rangle \in (((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst)) +\!\!+ (((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst))$, demostrando así que efectivamente que la información de puertos brindada por la

expresión $((\Gamma 2\ ss)\ sst) \backslash (((\Gamma 1\ ss)\ sst))$ no contiene información de puertos ya existentes en $(((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst)) +\!\!+ (((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst))$. Utilizamos el lema `\[]×-noRep` que determina que aplicar la operación $\backslash$ a un contexto mantiene la propiedad de no tener repetidos y `map-noRep` ya mencionado previamente para obtener que $((\Gamma 2\ ss)\ sst) \backslash (((\Gamma 1\ ss)\ sst)$ no tiene tampoco información repetida de puertos. Con estos dos resultados previos tenemos que se satisfacen las hipótesis del lema de weakening derecho el cual determina que se cumple entonces el juicio:

$$(\Phi\ ss)\ sst,\quad ((((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst)) +\!\!+ (((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst))) +\!\!+ (((\Gamma 2\ ss)\ sst) \backslash (((\Gamma 1\ ss)\ sst))) \vdash p\ \triangleright\ (\Delta\ ss)\ sst$$

Por asociatividad de la concatenación de listas tenemos que:

$$\left(\underbrace{(((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst)) +\!\!+ (((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst))}_{a}\right) +\!\!+ \underbrace{(((\Gamma 2\ ss)\ sst) \backslash (((\Gamma 1\ ss)\ sst)))}_{b} \equiv$$

$$\underbrace{(((\Gamma 1\ ss)\ sst) \backslash ((\Gamma 2\ ss)\ sst))}_{b} +\!\!+ \left(\underbrace{(((\Gamma 1\ ss)\ sst) \cap ((\Gamma 2\ ss)\ sst)) +\!\!+ (((\Gamma 2\ ss)\ sst) \backslash (((\Gamma 1\ ss)\ sst)))}_{a}\right)$$

Finalmente aplicando la definición de la operación de $\cup$ obtenemos que se satisface el juicio deseado:

$$(\Phi\ ss)\ sst, ((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2\ ss)\ sst) \vdash p\ \triangleright\ (\Delta\ ss)\ sst \tag{7.5}$$

Mostramos en la figura 7.9 la codificación de esta demostración. Un corolario directo de este resultado se deduce agregando como hipótesis que la información de puertos en común entre los anteriores contextos de puertos $\Gamma 1$ y $\Gamma 2$ unifica mediante las listas de sustituciones ss y sst dadas. A partir de esta hipótesis adicional podemos usar el resultado 7.1 demostrado en la sección anterior para tener entonces que $((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2\ ss)\ sst)$ es una permutación de $((\Gamma 2\ ss)\ sst) \cup ((\Gamma 1\ ss)\ sst)$. Aplicando ahora la regla de permutación de contextos `Permutation-env` del juicio de tipado al juicio demostrado anteriormente tenemos entonces que:

$$(\Phi\ ss)\ sst, ((\Gamma 2\ ss)\ sst) \cup ((\Gamma 1\ ss)\ sst) \vdash p\ \triangleright\ (\Delta\ ss)\ sst \tag{7.6}$$

Siguiendo un razonamiento similar demostramos el siguiente resultados para contextos de tipos básicos, esto es, dada una lista de sustituciones de tipos básicos ss, dos contextos de tipos básicos $\Phi 1$ y $\Phi 2$, un contexto de puertos Γ, un contexto de tipos de canal Δ y un proceso p cualesquiera, si se cumple $\Phi 1, \Gamma \vdash p\ \triangleright \Delta$, y también que los contextos $\Phi 1$ y $\Phi 2$ no tienen información repetida, entonces el proceso p también satisface el juicio de tipos para los contextos $(\Phi 1\ ss) \cup (\Phi 2\ ss)$, $\Gamma\ ss$ y $\Delta\ ss$, o sea, se cumple:

$$(\Phi 1\ ss) \cup (\Phi 2\ ss), \Gamma\ ss \vdash p\ \triangleright \Delta\ ss \tag{7.7}$$

No detallamos esta demostración por ser muy similar pero un poco más sencilla a la que anteriormente presentamos, por involucrar una única lista de

sustituciones. Al igual que antes agregando como hipótesis que la información en común entre los contextos de tipos básicos $\Phi 1$ y $\Phi 2$ unifica mediante la lista de sustituciones ss dada, podemos usar el lema 7.3 presentado en la anterior sección para demostrar que $(\Phi 1\ ss) \cup (\Phi 2\ ss)$ es una permutación de $(\Phi 2\ ss) \cup (\Phi 1\ ss)$. Con esta premisa podemos entonces aplicar la regla de permutación de contextos `Permutation-env` del juicio de tipos, teniendo así que también se satisface el siguiente juicio.

$$(\Phi 2\ ss) \cup (\Phi 1\ ss), \Gamma\ ss \vdash p\ \triangleright \Delta\ ss \tag{7.8}$$

7.3.4. Propiedades de la función `extractSorts`

Comenzamos viendo el resultado que establece que el contexto Φ devuelto con las asignaciones de variables de tipos básicos realizadas por la función `extractSorts` no contiene pares con su primer componente repetida. Se puede comprobar esta propiedad fácilmente viendo la implementación de esta función que sólo agrega información para un nombre en el contexto devuelto si no existe ya.

Otra propiedad para el contexto Φ devuelto por esta función es que no contiene nombres del contexto dado como argumento. También se puede fácilmente corroborar esta propiedad viendo que esta función sólo agrega información para un nombre si éste no pertenece al contexto dado como argumento

Finalmente vemos la codificación del resultado que establece que para una lista de expresiones le, un contexto Φ y una lista de tipos básicos ls cualesquiera, se satisface el juicio de tipado para listas expresiones para la lista le, y el tipo básico y contexto devueltos por la función `extractSorts`. O sea este resultado certifica que la función `extractSorts` infiere correctamente el tipo de una lista de expresiones bajo un contexto parcialmente dado.

```
extractSortsSoundness : (le : List Expression)
  (Φ : Sorts)(ls : List Sort) →
  (Φ ++ (proj₂ (extractSorts le Φ ls))) ⊢ le ▶
                                          (proj₁ (extractSorts le Φ ls))
```

```
perm-Γ2Θ∪Γ1θ : (ss : List (ℕ × Sort))(sst : List (ℕ × SType))
               (Γ1 Γ2 : Types) →
  unifyLPST ((map (map× fstPar fstPar) (ET.join Γ1 Γ2)) ++
             (map (map× sndPar sndPar) (ET.join Γ1 Γ2)))   ≡
  just (ss , sst) → NRL.NoRep Γ1 → NRL.NoRep Γ2 →
  permutation ((lsubSTLPNT sst (lsubSLPNT ss Γ1)) ∪n
               (lsubSTLPNT sst (lsubSLPNT ss Γ2)))
              ((lsubSTLPNT sst (lsubSLPNT ss Γ2)) ∪n
               (lsubSTLPNT sst (lsubSLPNT ss Γ1)))
perm-Γ2Θ∪Γ1θ ss sst Γ1 Γ2 unifyLPST noRepΓ1 noRepΓ2 with
  (((lsubSTLPNT sst (lsubSLPNT ss Γ2))) \\[]×n
    ((lsubSTLPNT sst (lsubSLPNT ss Γ1))))          ++
  ((((lsubSTLPNT sst (lsubSLPNT ss Γ2))) ∩n
    ((lsubSTLPNT sst (lsubSLPNT ss Γ1))))       ++
   (((lsubSTLPNT sst (lsubSLPNT ss Γ1))) \\[]×n
    ((lsubSTLPNT sst (lsubSLPNT ss Γ2)))))       |
  LMT.assoc (((lsubSTLPNT sst (lsubSLPNT ss Γ2))) \\[]×n
              ((lsubSTLPNT sst (lsubSLPNT ss Γ1))))
            (((lsubSTLPNT sst (lsubSLPNT ss Γ2))) ∩n
             ((lsubSTLPNT sst (lsubSLPNT ss Γ1))))
            (((lsubSTLPNT sst (lsubSLPNT ss Γ1))) \\[]×n
              ((lsubSTLPNT sst (lsubSLPNT ss Γ2))))
... |
  .(((((lsubSTLPNT sst (lsubSLPNT ss Γ2))) \\[]×n
      ((lsubSTLPNT sst (lsubSLPNT ss Γ1))))       ++
    (((lsubSTLPNT sst (lsubSLPNT ss Γ2))) ∩n
     ((lsubSTLPNT sst (lsubSLPNT ss Γ1)))))        ++
   (((lsubSTLPNT sst (lsubSLPNT ss Γ1))) \\[]×n
    ((lsubSTLPNT sst (lsubSLPNT ss Γ2)))))  |
   refl =
  ET.permutation-trans
    (ET.permutation-trans
      (ET.permutation-++
        {e1 = (lsubSTLPNT sst (lsubSLPNT ss Γ1)) \\[]×n
              (lsubSTLPNT sst (lsubSLPNT ss Γ2))}
        {e2 = (((lsubSTLPNT sst (lsubSLPNT ss Γ1)) ∩n
                (lsubSTLPNT sst (lsubSLPNT ss Γ2)))      ++
               ((lsubSTLPNT sst (lsubSLPNT ss Γ2)) \\[]×n
                (lsubSTLPNT sst (lsubSLPNT ss Γ1))))})
       (ET.permutation-++-cons
         {e2 = (lsubSTLPNT sst (lsubSLPNT ss Γ1)) \\[]×n
               (lsubSTLPNT sst (lsubSLPNT ss Γ2))}
         (ET.permutation-++-cons
            {e2 = (lsubSTLPNT sst (lsubSLPNT ss Γ2)) \\[]×n
                  (lsubSTLPNT sst (lsubSLPNT ss Γ1))}
            (perm-Γ2Θ∩Γ1θ ss sst Γ1 Γ2
                          unifyLPST noRepΓ1 noRepΓ2))))
    (ET.permutation-++-cons
       {e2 = (lsubSTLPNT sst (lsubSLPNT ss Γ1)) \\[]×n
             (lsubSTLPNT sst (lsubSLPNT ss Γ2))}
       (ET.permutation-++
          {e1 = (lsubSTLPNT sst (lsubSLPNT ss Γ2)) ∩n
                (lsubSTLPNT sst (lsubSLPNT ss Γ1))}
          {e2 = (lsubSTLPNT sst (lsubSLPNT ss Γ2)) \\[]×n
                (lsubSTLPNT sst (lsubSLPNT ss Γ1))})))
```


Figura 7.8: Permutación de la unión de los contextos (Γ1 *ss*) *sst* y (Γ2 *ss*) *sst*

```
ΦΓ1⊢p▷Δ→ΦθΓ1θ∪Γ2θ⊢p▷Δθ : (ss : List (ℕ × Sort))
  (sst : List (ℕ × SType))(Φ : Sorts)(Γ1 Γ2 : Types)
  (Δ : STypes)(p : Process) →
  (Φ , Γ1) ⊢ p ▷ Δ → NRL.NoRep Γ1 → NRL.NoRep Γ2 →
  (lsubSSS ss Φ , (lsubSTLPNT sst (lsubSLPNT ss Γ1)) ∪n
                  (lsubSTLPNT sst (lsubSLPNT ss Γ2))) ⊢ p ▷
                                lsubSTLPCST sst (lsubSLPCST ss Δ)
ΦΓ1⊢p▷Δ→ΦθΓ1θ∪Γ2θ⊢p▷Δθ ss sst Φ Γ1 Γ2 Δ p ΦΓ1⊢p▷Δ
  noRepΓ1 noRepΓ2 with
  (lsubSTLPNT sst (lsubSLPNT ss Γ1)) \\[]×n
  (lsubSTLPNT sst (lsubSLPNT ss Γ2))          ++
  (lsubSTLPNT sst (lsubSLPNT ss Γ1)) ∩n
  (lsubSTLPNT sst (lsubSLPNT ss Γ2))          ++
  (lsubSTLPNT sst (lsubSLPNT ss Γ2)) \\[]×n
  (lsubSTLPNT sst (lsubSLPNT ss Γ1))
  |
  LMT.assoc ((lsubSTLPNT sst (lsubSLPNT ss Γ1)) \\[]×n
             (lsubSTLPNT sst (lsubSLPNT ss Γ2)))
             ((lsubSTLPNT sst (lsubSLPNT ss Γ1)) ∩n
             (lsubSTLPNT sst (lsubSLPNT ss Γ2)))
             ((lsubSTLPNT sst (lsubSLPNT ss Γ2)) \\[]×n
             (lsubSTLPNT sst (lsubSLPNT ss Γ1)))
... | .(((lsubSTLPNT sst (lsubSLPNT ss Γ1)) \\[]×n
         (lsubSTLPNT sst (lsubSLPNT ss Γ2))         ++
         (lsubSTLPNT sst (lsubSLPNT ss Γ1)) ∩n
         (lsubSTLPNT sst (lsubSLPNT ss Γ2)))            ++
         (lsubSTLPNT sst (lsubSLPNT ss Γ2)) \\[]×n
         (lsubSTLPNT sst (lsubSLPNT ss Γ1)))
    | refl =
 weakening-fold-r-Γ
   (λ p → λ p∈Γ2θ\\Γ1θ →
      (ET.∉++ (ET.prop1_\\[]×
                  (ET.map-noRep (subSTT sst)
                  (ET.map-noRep (subST ss) noRepΓ1))
                  (p∈Γ2θ\\Γ1θ))
              (ET.prop2_\\[]× {k = p}
                  {e1 = (lsubSTLPNT sst (lsubSLPNT ss Γ1))}
                  {e2 = (lsubSTLPNT sst (lsubSLPNT ss Γ2))}
                  (ET.map-noRep (subSTT sst)
                     (ET.map-noRep (subST ss) noRepΓ2))
                     (p∈Γ2θ\\Γ1θ))))
   (ET.\\[]×-noRep {e2 = (lsubSTLPNT sst (lsubSLPNT ss Γ1))}
                   (ET.map-noRep (subSTT sst)
                      (ET.map-noRep (subST ss) noRepΓ2)))
   (Permutation-env
      ES.permutation-refl
      (ET.permutation-sym (ET.env-permutation
                             (lsubSTLPNT sst (lsubSLPNT ss Γ1))
                             (lsubSTLPNT sst (lsubSLPNT ss Γ2))))
      EST.permutation-refl
      (conservation-env ss sst ΦΓ1⊢p▷Δ))
```

Figura 7.9: Primer lema auxiliar del juicio de tipos para contextos de puertos

7.3.5. Demostración de Solidez del Algoritmo de Inferencia de tipos

A partir de los resultados demostrados en las dos secciones anteriores podemos ahora presentar la demostración de solidez con del algoritmo de inferencia de tipos presentado respecto al juicio de tipos. Como ya mencionamos hacemos inducción en el proceso p. A continuación detallamos los distintos casos de esta inducción.

Comenzamos por el caso base cuando el proceso es `inact`. Tal cual explicamos en la sección 7.1, en este caso el algoritmo devuelve tres contextos vacíos. Para la demostración de este caso base aplicamos la regla `Inact` del juicio de tipos vista en la sección 5.5. Para usar esta regla necesitamos demostrar que los contextos no tienen información repetida lo cual es trivial por ser vacíos. También tenemos que ver que se cumpla que toda la información de tipos de canales sea de tipo `end` lo cual es también trivial por ser vacía la información de tipos de canales.

En el caso inductivo cuando el proceso es de la forma $\texttt{accept}\ n(c)\ \texttt{in}\ p$, dado que el algoritmo de inferencia de tipos pudo inferir los contextos correctamente entonces la llamada recursiva con el proceso p también tiene que ser exitosa y lograr inferir contextos Φ , Γ y Δ para el proceso p. Como la llamada recursiva devuelve estos contextos para el proceso p entonces podemos aplicar la hipótesis inductiva, tenemos así que se cumple el juicio $\Phi, \Gamma \vdash p \ \triangleright \Delta$. La demostración se divide a continuación en dos casos dependiendo de si la variable de puerto n pertenece o no al contexto Γ. En el caso de que n aparezca en el contexto Γ, identificamos mediante la expresión $n{\in}\Gamma$ la demostración de esta afirmación. Examinando el algoritmo de inferencia en la casuística seguida hasta ahora, vemos que para que haya un resultado existoso por parte de éste tienen que existir listas de sustituciones ss y sst que unifican la información de tipos de canal de la primera y segunda proyección del tipo de puertos devuelto por el contexto Γ para el puerto n. Esto se corresponde con las siguientes expresiones $fstPar(\Gamma \bullet \in \ n{\in}\Gamma)$ y $sndPar(\Gamma \bullet \in \ n{\in}\Gamma)$, con la información de tipos y su complemento de la variable c dada por el contexto Δ, o sea $\Delta \bullet (\texttt{var}\ c)$ y $\overline{\Delta \bullet (\texttt{var}\ c)}$ respectivamente. Vemos también que los contextos devueltos en este caso tienen que ser de la forma $(\Phi\ ss)\ sst$, $(\Gamma\ ss)\ sst$ y $((\Delta\ ss)\ sst)\backslash c$, por lo que debemos demostrar que se cumple el juicio $(\Phi\ ss)\ sst, (\Gamma\ ss)\ sst \vdash \texttt{accept}\ n(c)\ \texttt{in}\ p \ \triangleright ((\Delta\ ss)\ sst)\backslash c$. Aplicamos ahora el lema de conservación presentado en la sección 5.6.5 al juicio $\Phi, \Gamma \vdash p \triangleright \Delta$ válido por hipótesis inductiva, obteniendo entonces que se cumple $(\Phi\ ss)\ sst, (\Gamma\ ss)\ sst \vdash p \ \triangleright (\Delta\ ss)\ sst$, con lo cual aplicando la regla `Accept` tenemos que se satisface el juicio deseado $(\Phi\ ss)\ sst, (\Gamma\ ss)\ sst \vdash \texttt{accept}\ n(c)\ \texttt{in}\ p \ \triangleright ((\Delta\ ss)\ sst)\backslash c$. Para aplicar esta regla del juicio de tipos necesitamos ver primero que se cumple que la variable de puerto n pertenece al contexto $(\Gamma\ ss)\ sst$, segundo que la primera proyección del tipo de puerto de la variable n en el anterior contexto es idéntica a la información de canal brindada por el contexto $(\Delta\ ss)\ sst$ para la variable c, y tercero que la segunda proyección del tipo de puerto de la variable n en el contexto $(\Gamma\ ss)\ sst$ es idéntico al complemento del tipo de canal asociado en el contexto $(\Delta\ ss)\ sst$ a la variable c. O sea que se cumplen las siguientes condiciones:

$$n \in (\Gamma\ ss)\ sst) \tag{7.9}$$

$$fstPar\ (((\Gamma\ ss)\ sst) \bullet\in\ n{\in}((\Gamma\ ss)\ sst)) \equiv (((\Delta\ ss)\ sst) \bullet (\texttt{var}\ c)) \quad (7.10)$$

$$sndPar\ (((\Gamma\ ss)\ sst) \bullet\in\ n{\in}((\Gamma\ ss)\ sst)) \equiv \overline{(((\Delta\ ss)\ sst) \bullet (\texttt{var}\ c))} \quad (7.11)$$

Demostramos el enunciado 7.9 utilizando la hipótesis de este caso que establece que n aparece en el contexto Γ como premisa para el lema $\texttt{map}{\in_1}$, este lema establece que si existe información para cierta clave en un contexto entonces va seguir existiendo esta información al aplicar una lista de sustituciones, aplicamos dos veces este lema una vez para la lista de sustituciones de tipos básicos ss y otra para la lista de sustituciones de canal sst, obteniendo así que n aparece en $(\Gamma\ ss)\ sst)$. Denotamos la demostración de este resultado mediante la expresión $n{\in}((\Gamma\ ss)\ sst)$. Para demostrar el segundo enunciado expuesto en 7.10 usamos el resultado de solidez de la unificación de tipos de canales visto en el capítulo 6.2. Mediante este resultado sabemos que las listas de sustituciones ss y sst, halladas al aplicar el algoritmo de unificación a la primera proyección del tipo de puertos devuelto por el contexto Γ para el canal n y la información de la variable c dada por el contexto Δ, realmente unifican las anteriores expresiones. También usamos el lema $\texttt{subs}\Delta\bullet$ que establece que es equivalente obtener la información de tipos de una clave en un contexto de tipos de canal y luego aplicar una lista de sustituciones al tipo obtenido, a aplicar la lista de sustituciones a todo el contexto de tipos de canal, y luego obtener la información de la clave. Otro lema similar utilizado es $\texttt{subs}\Gamma\bullet$, el cual afirma que la primera proyección del tipo de puertos correspondiente a la variable n en el contexto $(\Gamma\ ss)\ sst$ equivale a aplicar las listas de sustituciones ss y sst al tipo de canal resultante de obtener la primera proyección del tipo de puerto correspondiente a la variable n en el contexto Γ. Vemos a continuación una secuencia de equivalencias obtenidas mediante los anteriores resultados descritos.

$$\begin{array}{lcl}
fstPar(((\Gamma\ ss)\ sst) \bullet\in\ n{\in}((\Gamma\ ss)\ sst)) & \equiv & \{\text{por lema } \texttt{subs}\Gamma\bullet\} \\
((fstPar(\Gamma \bullet\in\ n{\in}\Gamma))\ ss)\ sst & \equiv & \{\text{por solidez de la unificación}\} \\
((\Delta \bullet (\texttt{var}\ c))\ ss)\ sst & \equiv & \{\text{por lema } \texttt{subs}\Delta\bullet\} \\
((\Delta\ ss)\ sst) \bullet (\texttt{var}\ c) & &
\end{array}$$

Finalmente aplicando la clausura transitiva a las anteriores equivalencias tenemos entonces la demostración del resultado 7.10 deseado.

Nos resta entonces sólo demostrar el enunciado presentado en 7.11 para terminar este caso. Nuevamente comenzamos usando el resultado de solidez de la unificación de tipos de canal tenemos que $((sndPar(\Gamma \bullet\in\ n{\in}\Gamma))\ ss)\ sst)$ equivale a $(\overline{\Delta \bullet (\texttt{var}\ c)}\ ss)\ sst$. Utilizamos también el lema $\texttt{subs}\Delta\bullet$ ya descrito anteriormente, y el resultado $\texttt{subs}\Gamma\bullet\texttt{2}$ similar a $\texttt{subs}\Gamma\bullet$ pero que utiliza ahora la segunda proyección en vez de la primera. Otro lema utilizado es `complSTlsubSTST` el cual enuncia que equivale aplicar una lista de sustituciones de tipos de canal al complemento de un tipo de canal, a aplicar el complemento a la aplicación de la lista de sustituciones al tipo de canal. Tenemos también el lema equivalente `complSTlsubSST` pero para una lista de sustituciones de tipos básicos ahora. Presentamos ahora una secuencia de equivalencias resultantes de la aplicación de los anteriores lemas que demuestran el enunciado 7.11.

$$
\begin{array}{lcl}
sndPar(((\Gamma\ ss)\ sst) \bullet\in\ n\in((\Gamma\ ss)\ sst)) & \equiv & \{\text{por lema } \texttt{subs}\Gamma\bullet\texttt{2}\} \\
((sndPar(\Gamma \bullet\in\ n\in\Gamma))\ ss)\ sst & \equiv & \{\text{por solidez de la unificación}\} \\
\overline{(\Delta \bullet (\texttt{var}\ c)}\ ss)\ sst & \equiv & \{\text{por lema } \texttt{complSTlsubSST}\} \\
\overline{(\Delta \bullet (\texttt{var}\ c))}\ ss\ sst & \equiv & \{\text{por lema } \texttt{complSTlsubSTST}\} \\
\overline{((\Delta \bullet (\texttt{var}\ c))\ ss)\ sst} & \equiv & \{\text{por lema } \texttt{subs}\Delta\bullet\} \\
\overline{((\Delta\ ss)\ sst) \bullet (\texttt{var}\ c)} & &
\end{array}
$$

Vemos ahora la demostración cuando se cumple que n no aparece en Γ, en este caso viendo la estructura del algoritmo de inferencia de tipos en este caso vemos que los contextos devueltos en este caso son Φ, $\Delta\backslash(\texttt{var}\ c)$ y $\Gamma <+\langle n, par\ (\Delta \bullet (\texttt{var}\ c))\ (complST\ (\Delta \bullet (\texttt{var}\ c)))\rangle$, por lo que ahora debemos demostrar que se satisface el siguiente juicio:

$$\Phi, \Gamma <+\langle n, par\ (\Delta\bullet(\texttt{var}\ c))\ (complST\ (\Delta\bullet(\texttt{var}\ c)))\rangle \vdash \texttt{accept}\ n(c)\ \texttt{in}\ p \triangleright \Delta\backslash(\texttt{var}\ c) \tag{7.12}$$

Aplicamos el resultado de *weakening lemma* para contextos de tipos de puertos dado en la sección 5.6.3, utilizando la hipótesis inductiva y que n no pertenece a Γ, como premisas de este lema llegamos a que se cumple el juicio:

$$\Phi, \Gamma <+\langle n, par\ (\Delta \bullet (\texttt{var}\ c))\ (complST\ (\Delta \bullet (\texttt{var}\ c)))\rangle \vdash p\ \triangleright \Delta$$

Sobre el anterior resultado aplicamos ahora la regla `Accept` y tenemos que se satisface entonces el juicio 7.12 deseado. Para aplicar esta regla debemos además demostrar que n aparece en $\Gamma <+\langle n, par\ (\Delta\bullet(\texttt{var}\ c))\ (complST\ (\Delta\bullet(\texttt{var}\ c)))\rangle$, denotemos a esta demostración mediante la expresión $n \in \Gamma\ <+n$, y que la primera proyección del tipo de puerto de la variable n en el anterior contexto es idéntica a la información de canal brindada por el contexto Δ para la variable c, o sea que se cumple:

$$fstPar\ ((\Gamma <+\langle n, par\ (\Delta\bullet(\texttt{var}\ c))\ (complST\ (\Delta\bullet(\texttt{var}\ c)))\rangle) \bullet\in\ n\in\Gamma <+n) \equiv \Delta \bullet (\texttt{var}\ c) \tag{7.13}$$

Usamos ahora el resultado $\texttt{k}\in\texttt{e}\texttt{<+}\texttt{kd}\wedge\texttt{e}\texttt{<+}\texttt{kd}\bullet\texttt{k}\equiv\texttt{d}$ del módulo `Enviroment` el cual establece que para cualquier contexto e, clave $k1$ y dato $d1$, si no existe información para la clave $k1$ en el contexto e, entonces se cumple que $k1$ pertenece al contexto $e <+\langle k1, d1\rangle$ y $(e <+\langle k1, d1\rangle) \bullet k1 \equiv d1$. Instanciando este resultado con $e = \Gamma$, $k1 = n$ y $d1 = \langle n, par\ (\Delta\bullet(\texttt{var}\ c))\ (complST\ (\Delta\bullet(\texttt{var}\ c)))\rangle$ tenemos entonces que n aparece en $\Gamma <+\langle n, par\ (\Delta \bullet (\texttt{var}\ c))\ (complST\ (\Delta \bullet (\texttt{var}\ c)))\rangle$ tal cual queríamos demostrar, y también que:

$$(\Gamma <+\langle n, par\ (\Delta \bullet (\texttt{var}\ c))\ (complST\ (\Delta \bullet (\texttt{var}\ c)))\rangle) \bullet n \equiv par\ (\Delta \bullet (\texttt{var}\ c))\ (complST\ (\Delta \bullet (\texttt{var}\ c)))$$

Aplicando la función $fstPar$ a cada lado de la anterior relación de identidad, obtenemos entonces que se cumple la segunda condición expuesta en 7.13,

necesaria para poder aplicar la regla de juicio `Accept`. Dado que se satisfacen las premisas de esta regla obtenemos así finalmente que se satisface el juicio 7.12 buscado en este caso.

El caso inductivo cuando el proceso es de la forma `request` $n(c)$ `in` p es análogo al anterior, con lo cual ahora presentamos el caso que el proceso sea $c![\tilde{le}]; p$. Viendo el algoritmo de inferencia para este caso, vemos que la única forma que tiene un resultado exitoso es cuando la llamada recursiva para el proceso p tiene éxito también, devolviendo contextos Φ, Γ y Δ. En este caso el resultado entonces es dado por los contextos $\Phi \cup \Phi'$, Γ y $\Delta <+\langle c, \texttt{sendS}\ \tilde{ls}\ (\Delta \bullet c)\rangle$, donde $\tilde{ls}$ y Φ' son el resultado de aplicar la función `extractSorts` a la lista de expresiones $\tilde{le}$, Φ, y una lista de variables de tipo frescas. En 7.3.4 vimos que el contexto Φ' devuelto por esta función no contiene pares con su primer componente repetida, y su dominio es disjunto del dominio de Φ. Podemos entonces aplicar el weakening derecho, visto en 5.6.2, a la hipótesis inductiva obteniendo así que se satisface $\Phi \cup \Phi', \Gamma \vdash p\ \triangleright \Delta$. Utilizando el lema `extractSortsSoundness` obtenemos que $\Phi \cup \Phi' \vdash \tilde{le}\ \triangleright \tilde{ls}$. Finalmente usamos la regla `Send` del juicio de tipos con los dos anteriores resultados para obtener de esta forma que se satisface el juicio 7.14 deseado.

$$\Phi \cup \Phi', \Gamma \vdash c![\tilde{le}]; p\ \triangleright \Delta <+\langle c, \texttt{sendS}\ \tilde{ls}\ (\Delta \bullet c)\rangle \tag{7.14}$$

Cuando el proceso es de la forma $c?(\tilde{ln})$ `in` p vemos que para que haya un resultado exitoso la llamada recursiva para el proceso p tiene que ser exitosa también, devolviendo entonces ciertos contextos Φ, Γ y Δ. Así el resultado viene dado por los contextos $\Phi \backslash \tilde{ln}$, Γ y $\Delta <+\langle c, \texttt{receiveS}\ \tilde{ls}\ (\Delta \bullet c)\rangle$, donde el vector de tipos básicos $\tilde{ls}$ es el devuelto por la función `extractSorts`. Obtenemos que se satisface el juicio 7.15 aplicando la regla `Receive` a la hipótesis inductiva $\Phi, \Gamma \vdash p\ \triangleright \Delta$.

$$\Phi \backslash\ \tilde{ln}, \Gamma \vdash c?(\tilde{ln})\ \texttt{in}\ p\ \triangleright \Delta <+\langle c, \texttt{receiveS}\ \tilde{ls}\ (\Delta \bullet c)\rangle \tag{7.15}$$

Si el proceso es de la forma `throw` $c1(c2)$ p vemos que en este caso para que el algoritmo de inferencia de tipos devuelva un resultado exitoso el canal $c1$ tiene que ser distinto que el canal $c2$, y la llamada recursiva al algoritmo con el proceso p tiene que devolver contextos Φ, Γ y Δ. También vemos que el canal $c2$ no puede pertenecer al contexto Δ. En este caso entonces los contextos devueltos son Φ , $\Delta <+\langle c, \texttt{sendST}\ (\Delta \bullet c1)\rangle <+\langle c2, \texttt{varST}\ (getNumCh\ c2)\rangle$ y Γ. Aplicando la regla `Throw` a las premisas obtenidas que establecen que $c1 \neq c2$, $c2$ no pertenece a Δ y la hipótesis inductiva, tenemos entonces que se satisface el juicio de tipos para los contextos encontrados, esto es:

$$\Phi, \Gamma \vdash \texttt{throw}\ c1(c2)\ p \triangleright \Delta <+\langle c, \texttt{sendST}\ (\Delta \bullet c1)\rangle <+\langle c2, \texttt{varST}\ (getNumCh\ c2)\rangle$$

Para un proceso de la forma `catch` $c(cv)$ `in` p analizando el algoritmo en este caso vemos que necesariamente para que encuentre contextos el canal c tiene que ser distinto que la variable de canal cv, y la llamada recursiva con el proceso p tiene que encontrar contextos Φ , Γ y Δ, con lo que los contextos devueltos para el proceso original son entonces Φ, $(\Delta <+\langle c, \texttt{receiveST}\ (\Delta \bullet (\texttt{var}\ cv))\ (\Delta \bullet c)\rangle) \backslash (\texttt{var}\ cv)$ y Γ. Aplicando la regla `Catch` usando como premisas que el canal $c \neq \texttt{var}\ cv$ y la hipótesis inductiva demostramos que se satisface el siguiente juicio buscado.

$$\Phi, \Gamma \vdash \texttt{catch}\ c(cv)\ \texttt{in}\ p \triangleright (\Delta <+ \langle c, \texttt{receiveST}\ (\Delta \bullet (\texttt{var}\ cv))\ (\Delta \bullet c)\rangle) \setminus (\texttt{var}\ cv)$$

Para un proceso de la forma $p_1 \mid p_2$ analizando el algoritmo de inferencia de tipos para este caso vemos que la única forma en que el algoritmo devuelva con éxito contextos de tipado es que se cumplan las siguientes condiciones:

1. las llamadas recursivas para los procesos p_1 y p_2 devuelvan exitosamente contextos $\Phi 1$, $\Gamma 1$ y $\Delta 1$ para el proceso p_1, y $\Phi 2$, $\Gamma 2$ y $\Delta 2$ para el proceso p_2
2. la unificación de la lista de pares de tipos de canal resultante de concatenar por un lado las parejas de primeras coordenadas de la información en común de los contextos $\Gamma 1$ y el $\Gamma 2'$, y por otro lado las segundas coordenadas de esta misma información común, devuelve dos listas de sustituciones: una de tipos básicos ss y otra de tipos de canal sst, donde $\Gamma 2' = (\Gamma 2\ \pi_s)\ \pi_{st}$ para π_s y π_{st} sustituciones que renombran las variables de tipos básicos y de canales respectivamente, de forma de que $\Gamma 2'$ no tenga variables de tipos en común con $\Gamma 1$
3. la unificación de la lista de pares de tipos básicos resultante de emparejar la información en común de los contextos $\Phi 1\ ss$ y $\Phi 2'\ ss$, da una lista de sustituciones de tipos básicos $ss2$. Donde $\Phi 2' = \Phi\ \pi_s$.
4. la aplicación de la función $\otimes$ a los contextos $((\Delta 1\ ss)\ sst)\ ss2$ y $((\Delta 2'\ ss)\ sst)\ ss2$ devuelve el contexto Δ

Bajo las anteriores condiciones vemos que el algoritmo de inferencia para este caso devuelve los contextos $((\Phi 1\ ss)\ ss2) \cup ((\Phi 2'\ ss)\ ss2)$, $(((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2'\ ss)\ sst))\ ss2$, y Δ, por lo que debemos entonces demostrar que se satisface el siguiente juicio:

$$((\Phi 1\ ss)\ ss2) \cup ((\Phi 2'\ ss)\ ss2), (((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2'\ ss)\ sst))\ ss2 \vdash p_1 \mid p_2 \triangleright \Delta \tag{7.16}$$

Aplicamos ahora la regla `Conc` del juicio de tipos para obtener así el anterior resultado. Esta regla requiere como premisas: la condición 4, y que se cumplan los siguientes dos juicios:

$$((\Phi 1\ ss)\ ss2) \cup ((\Phi 2'\ ss)\ ss2), (((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2'\ ss)\ sst))\ ss2 \vdash p_1 \triangleright (((\Delta 1\ ss)\ sst)\ ss2 \tag{7.17}$$

$$((\Phi 1\ ss)\ ss2) \cup ((\Phi 2'\ ss)\ ss2), (((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2'\ ss)\ sst))\ ss2 \vdash p_2 \triangleright (((\Delta 2'\ ss)\ sst)\ ss2 \tag{7.18}$$

Por otro lado, aplicando las hipótesis inductivas tenemos que se satisfacen los siguientes juicios.

$$\Phi 1, \Gamma 1 \vdash p_1 \triangleright \Delta 1 \tag{7.19}$$

$$\Phi 2, \Gamma 2 \vdash p_2 \triangleright \Delta 2$$

Aplicamos el lema de conservación al juicio anterior con las sustituciones de renombre de variables de tipos π_s y π_{st}, obteniendo así el siguiente resultado.

$$\Phi 2', \Gamma 2' \vdash p_2 \triangleright \Delta 2' \tag{7.20}$$

Para demostrar el juicio 7.17 comenzamos utilizando el resultado visto en la sección 7.3.1 para contextos de tipos de puertos. Dado que obtuvimos el contexto $\Gamma 1$ resultado de una llamada recursiva al algoritmo de inferencia, entonces mediante el resultado antes mencionado se cumple necesariamente que este contexto no contiene información repetida. Análogamente $\Gamma 2$ no contiene información repetida, con lo cual usando el lema `map-noRep`, que establece que aplicar una lista de sustituciones a un contexto no introduce información repetida, tenemos que $\Gamma 2'$ tampoco tiene información repetida. A partir de estas premisas podemos aplicar el lema 7.4 visto al comienzo de la sección 7.3.3 con las premisas de que los contextos de tipos de puertos no contienen información repetida y que se satisface el juicio 7.19 para obtener el siguiente resultado.

$$(\Phi 1\ ss)\ sst, ((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2'\ ss)\ sst) \vdash p_1 \triangleright (\Delta 1\ ss)\ sst \tag{7.21}$$

Aplicamos ahora el lema 7.7 al anterior resultado. Para esto instanciamos la variable ss de este lema con la sustitución $ss2$, la variable $\Phi 1$ de este lema con $(\Phi 1\ ss)\ sst$, mientras que $\Phi 2$ con $(\Phi 2'\ ss)\ sst$, Γ con $((\Gamma 1\ ss)\ sst) \cup ((\Gamma 2'\ ss)\ sst)$ y Δ con $(\Delta 1\ ss)\ sst$. Para usar este lema necesitamos demostrar también que los contextos $(\Phi 1\ ss)\ sst$ y $(\Phi 2'\ ss)\ sst$ no tienen información repetida. Mediante el resultado 7.3.1 para contextos de tipos básicos aplicados sabemos que como $\Phi 1$ y $\Phi 2$ fueron resultados del algoritmo de inferencia entonces estos contextos no tienen información repetida. Dado que $\Phi 2'$ es el resultado de aplicar sustituciones a Φ tampoco tiene información repetida por el lema `map-noRep`. Utilizando nuevamente el lema anterior tenemos finalmente que los contextos$(\Phi 1\ ss)\ sst$ y $(\Phi 2'\ ss)\ sst$ no tienen información repetida. Tenemos que se cumplen todas las premisas del lema 7.7 obteniendo entonces finalmente que se cumple el siguiente juicio 7.17.

La demostración del juicio 7.18 es similar a la demostración previa, la principal diferencia es que ahora usamos algunos lemas auxiliares similares a los anteriores pero que utilizan la solidez de los algoritmos de unificación utilizados en este caso. Como hicimos previamente usamos el resultado visto en la sección 7.3.1 para demostrar ahora que los contextos $\Gamma 1$ y $\Gamma 2$ no contienen información repetida. Mediante este resultado, y que se satisface el juicio 7.20 por hipótesis inductiva, y también usando la condición 2 podemos aplicar el lema 7.6, instanciando la variable del lema Φ con $\Phi 2'$, la variable de lema $\Gamma 1$ con $\Gamma 2'$, $\Gamma 2$ con $\Gamma 1$ y Δ con $\Delta 2'$, así obtenemos entonces que se satisface el siguiente enunciado.

$$(\Phi 2'\ ss)\ sst, ((\Gamma 2'\ ss)\ sst) \cup ((\Gamma 1\ ss)\ sst) \vdash p_2 \triangleright (\Delta 2'\ ss)\ sst$$

Utilizamos el lema 7.8 al anterior juicio, instanciando la variable ss de este lema con $ss2$, la variable $\Phi 1$ con $(\Phi 2'\ ss)\ sst$, $\Phi 2$ con $(\Phi 1\ ss)\ sst$, Γ con

$((\Gamma 2' \ ss) \ sst) \cup ((\Gamma 1 \ ss) \ sst)$ y Δ con $(\Delta 2' \ ss) \ sst$. Ya demostramos anteriormente que los contextos $(\Phi 1 \ ss) \ sst$ y $(\Phi 2' \ ss) \ sst$ no tienen información repetida, con lo cual usando además la condición 3 tenemos que se cumplen las hipótesis necesarias para aplicar este lema teniendo entonces que se cumple el juicio 7.18 que restaba demostrar.

Cuando el proceso es de la forma $(\nu n)p$ se cumple que los contextos devueltos tienen que ser Φ , $\Gamma \backslash n$ y Δ, donde Φ , Γ y Δ fueron los contextos devueltos por la llamada recursiva para el proceso p. Aplicamos entonces la regla `Nres` a la hipótesis inductiva demostrando así el resultado buscado.

Finalmente para un proceso de la forma $(\nu c)p$ vemos que los contextos devueltos por el algoritmo de inferencia son los mismos que los devueltos por la llamada recursiva con el proceso p, aplicando la regla `Cres'` obtenemos entonces la demostración del juicio buscado.

Dado que estos son todos los casos dados por la definición del conjunto de los procesos queda entonces demostrada la solidez del algoritmo de inferencia con respecto al juicio de tipos.

Capítulo 8

Conclusiones

La principal contribución de este trabajo es la demostración de que el teorema de existencia de esquema de tipo principal que vale para el cálculo lambda simplemente tipado *à la Curry* se puede extender a los Tipos de Sesión. Además hemos dado una formalización en la Teoría Constructiva de Tipos de este resultado. En términos prácticos, este teorema se traduce en que la condición de compatibilidad en los diálogos entre procesos concurrentes es verificable estáticamente sin necesidad de incluir anotaciones de tipos en los procesos. Este resultado no se encuentra en la literatura. El trabajo más cercano es [11] pero aplica al cálculo π y no es formalizado en un asistente de pruebas. En [37] realizan también una formalización en el asistente de pruebas Coq de un algoritmo de inferencia, más específicamente, para un fragmento del lenguaje ML. Pero no demuestran la corrección de la unificación, la cual axiomatizan para completar la demostración de correctitud del algoritmo de inferencia.

Este trabajo fue hecho para una simplifación del cálculo de Tipos de Sesión pero conjeturamos que es extensible a todo el cálculo. Queda como posible trabajo a futuro incluir la selección y aceptación sobre etiquetas y, más interesante aún, la recursión en el sistema de tipos. Existen extensiones al cálculo de los Tipos de Sesión original, como por ejemplo *Multiparty Sessions* [38] que extienden la comunicación a más de dos participantes. Otro posible camino entonces podría ser extender el algoritmo de inferencia propuesto a alguna de estas extensiones.

El algoritmo de inferencia de Tipos de Sesión requiere la implementación de la unificación en esquemas de Tipos de Sesión. Esta implementación aunque se basa en algoritmos de unificación estándares, requirió adaptar estos algoritmos debido a la necesidad de unificar simultáneamente dos estructuras de tipos distintas pero interrelacionadas. Esto fue resuelto mediante dos algoritmos de unificación, donde uno de ellos utiliza al otro. Estos algoritmos utilizan dos clases de variables de tipos, y sus correspondientes sustituciones asociadas. Además propusimos la idea de esquema complementario que fue necesaria para lograr resolver la unificación en Tipos de Sesión. Demostramos formalmente la corrección de uno de los algoritmos de unificación presentados. La actual formalización de este trabajo supera las 5800 líneas de código Agda, de los cuales casi la mitad corresponden a la formalización de la unificación, con lo cual por razones de tiempo no completamos la formalización de corrección del segundo algoritmo de unificación. Sin embargo estimamos que la demostración restante

sigue básicamente la idea de la prueba ya realizada para el primero de los algoritmos de unificación. La codificación elegida para resolver la unificación es poco extensible; esto es debido a que cualquier cambio en la estructura de los datos, termina impactando fuertemente en los predicados de accesibilidad necesarios para demostrar la terminación. Es necesario entonces en este caso reestructurar la demostración de que se satisface el predicado de accesibilidad. En [39] resuelven el problema de unificación de forma más general utilizando estructuras arborescentes genéricas, tal cual se explica en [40], y luego implementan una biyección de árboles en Tipos de Sesión. Sin embargo, resta estudiar cómo impacta esto sobre la formalización de las demostraciones.

El sistema de tipos y el algoritmo de inferencia fueron formalizados en Teoría Constructiva de Tipos, más específicamente en el asistente de pruebas de Agda. Se comprobó el poder de expresión de esta teoría, la cual permitió formalizar de forma adecuada este trabajo. Sin embargo se encontraron ciertos problemas al usar el compilador de Agda, debido a los cuales se debió quitar el chequeo de parada en los módulos que contienen los algoritmos de unificación, para así lograr completar la compilación de nuestra formalización. Sin embargo estos módulos compilan por separado correctamente con el chequeo de parada habilitado.

Codificamos exitosamente la prueba de solidez del algoritmo de inferencia de Tipos de Sesión. Debido a problemas de tiempo, tal cual explicamos en párrafos anteriores, quedó por demostrar la completitud del algoritmo. Aunque sí brindamos una prueba informal de este resultado, quedando como posible trabajo a futuro una formalización completa de este resultado.

Otro problema pendiente por cuestiones de tiempo fue la formalización adecuada del concepto de α-conversión que restaría solucionar para así terminar la pruebas de los lemas de weakening para los distintos contextos para el juicio de tipado.

Capítulo 9

Bibliografía

[1] K. Honda, "Types for dyadic interaction," in *CONCUR'93* (E. Best, ed.), vol. 715 of *Lecture Notes in Computer Science*, pp. 509–523, Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57208-2_35.

[2] H. B. Curry, *Combinatory logic / [by] Haskell B. Curry [and] Robert Feys. With two sections by William Craig.* North-Holland Pub. Co., Amsterdam, 1958.

[3] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type disciplines for structured communication-based programming," in *ESOP'98*, vol. 1381 of *LNCS*, pp. 22–138, Springer, 1998.

[4] M. Dezani-ciancaglini, "Sessions and session types: an overview," tech. rep., In 6th International Workshop on Web Services and Formal Methods (WS-FM'09, 2010.

[5] K. Takeuchi, K. Honda, and M. Kubo, "An interaction-based language and its typing system," in *PARLE* (C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, eds.), vol. 817 of *Lecture Notes in Computer Science*, pp. 398–413, Springer, 1994.

[6] N. Yoshida and V. T. Vasconcelos, "Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication," in *1st International Workshop on Security and Rewriting Techniques*, vol. 171(4) of *ENTCS*, pp. 73–93, Elsevier, 2007.

[7] V. T. Vasconcelos, M. Giunti, N. Yoshida, and K. Honda, "Type safety without subject reduction for session types." 2010.

[8] S. J. Gay and M. Hole, "Subtyping for session types in the pi calculus.," *Acta Inf.*, pp. 191–225, 2005.

[9] A. Igarashi and N. Kobayashi, "A generic type system for the pi-calculus," *Theor. Comput. Sci.*, vol. 311, pp. 121–163, Jan. 2004.

[10] N. Kobayashi, B. C. Pierce, and D. N. Turner, "Linearity and the pi-calculus," 1999.

[11] V. T. Vasconcelos and K. Honda, "Principal typing schemes in a polyadic π-calculus," in *CONCUR'93, LNCS 715*, pp. 524–538, Springer-Verlag, 1993.

[12] M. Neubauer and P. Thiemann, "An implementation of session types," in *PADL* (B. Jayaraman, ed.), vol. 3057 of *Lecture Notes in Computer Science*, pp. 56–70, Springer, 2004.

[13] R. Pucella and J. A. Tov, "Haskell session types with (almost) no class," *SIGPLAN Not.*, vol. 44, pp. 25–36, Sept. 2008.

[14] M. Sackman and S. Eisenbach, "Session Types in Haskell: Updating Message Passing for the 21st Century," tech. rep., June 2008.

[15] L. G. Mezzina, "How to infer finite session types in a calculus of services and sessions," in *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION'08, (Berlin, Heidelberg), pp. 216–231, Springer-Verlag, 2008.

[16] K. Imai, S. Yuen, and K. Agusa, "Session type inference in haskell," in *PLACES* (K. Honda and A. Mycroft, eds.), vol. 69 of *EPTCS*, pp. 74–91, 2010.

[17] P. Collingbourne and P. H. J. Kelly, "Inference of session types from control flow," *Electron. Notes Theor. Comput. Sci.*, vol. 238, pp. 15–40, June 2010.

[18] W. A. Howard, "The formulas-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism* (J. P. Seldin and J. R. Hindley, eds.), pp. 479–490, Academic Press, 1980. Reprint of 1969 article.

[19] N. Szasz, "A theory of specifications, programs and proofs," June 1997. PhD thesis, Department of Computer Science, Chalmers University of Technology.

[20] A. Bove, "Programming in Martin-Löf type theory: Unification - A non-trivial example," November 1999. Licentiate Thesis of the Department of Computer Science, Chalmers University of Technology.

[21] A. Bove, P. Dybjer, and U. Norell, "A brief overview of agda - a functional language with dependent types," in *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009* (S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, eds.), vol. 5674 of *LNCS*, pp. 73–78, Springer, August 2009.

[22] A. Bove and P. Dybjer, "Dependent types at work," in *Language Engineering and Rigorous Software Development International LerNet ALFA Summer School* (A. Bove, L. Barbosa, A. Pardo, and J. S. Pinto, eds.), vol. 5520 of *LNCS*, pp. 57–99, Springer, 2009.

[23] U. Norell, "Dependently typed programming in agda," in *TLDI* (A. Kennedy and A. Ahmed, eds.), pp. 1–2, ACM, 2009.

[24] R. Milner, "The polyadic π-calculus: a tutorial," tech. rep., Logic and Algebra of Specification, 1991.

[25] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, August 1978.

[26] R. Milner, *A Calculus of Communicating Systems.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.

[27] S. J. Gay, "A sort inference algorithm for the polyadic π-calculus," in *POPL'93*, ACM, 1993.

[28] N. Kobayashi, "A partially deadlock-free typed process calculus (i) – a simple system –," 1996.

[29] M. P. Jones, "Type classes with functional dependencies," in *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, (London, UK, UK), pp. 230–244, Springer-Verlag, 2000.

[30] P. M. Löf, "Constructive Mathematics and Computer Programming," in *6-th International Congress for Logic, Methodology and Philosophy of Science, 1979*, pp. 153–175, North–Holland, 1982.

[31] B. Nordstrom, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, USA, July 1990.

[32] T. Coquand, B. Nordström, J. M. Smith, and B. von Sydow, "Type theory and programming," *EATCS*, vol. 52, pp. 203–228, 1994.

[33] L. Magnusson and B. Nordström, "The alf proof editor and its proof engine," in *Proceedings of the international workshop on Types for proofs and programs*, TYPES '93, (Secaucus, NJ, USA), pp. 213–237, Springer-Verlag New York, Inc., 1994.

[34] T. Coquand, "Pattern matching with dependent types," in *In Proceedings of the Workshop on Types for Proofs and Programs*, pp. 85–92, 1992.

[35] U. Norell, *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[36] B. Nordström, "Terminating general recursion," 1988.

[37] C. Dubois and V. Ménissier-Morain, "Certification of a type inference tool for ml: Damas-milner within coq," *Journal of Automated Reasoning*, vol. 23, pp. 3–4, 1999.

[38] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida, "Global Progress in Dynamically Interleaved Multiparty Sessions," in *Proc. of Concur*, vol. 5201 of *LNCS*, pp. 418–433, Springer, 2008.

[39] L. G. Mezzina, "How to infer finite session types in a calculus of services and sessions," in *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION'08, (Berlin, Heidelberg), pp. 216–231, Springer-Verlag, 2008.

[40] J. A. Robinson, "Logic and logic programming," *Commun. ACM*, vol. 35, no. 3, pp. 40–65, 1992.

Apéndice A

Weakening Lemma

En este apéndice presentamos la demostración completa de este resultado, esta es por inducción en el árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$..

Teorema. *Sean* Θ, Γ, Δ *y* P *tales que se satisface* $\Theta; \Gamma \vdash P \triangleright \Delta$.
Entonces,

1. *Si* $X \notin dom(\Theta)$ *, entonces* $\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash P \triangleright \Delta$.
2. *Si* $a \notin dom(\Gamma)$ *, entonces* $\Theta; \Gamma \cdot a : S \vdash P \triangleright \Delta$.
3. *Si* $k \notin dom(\Delta)$ *, entonces* $\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \texttt{end}$.

A.1. Demostración de (1)

Caso Base [Inact]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\frac{\Delta \text{ completa}}{\Theta; \Gamma \vdash \texttt{inact} \triangleright \Delta}\ [\textsc{Inact}]$$

Con lo cual $P \equiv \texttt{inact}$ y Δ es completa, como $X \notin \text{dom}(\Theta)$([6] pág. 81), podemos escribir $\Theta \cdot X : \tilde{S}\tilde{\alpha}$, y así construir el siguiente árbol de derivación:

$$\frac{\Delta \text{ completa}}{\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash \texttt{inact} \triangleright \Delta}\ [\textsc{Inact}]$$

Obteniendo que $\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash P \triangleright \Delta$

Caso Base [Var]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\frac{\Delta' \text{ completa} \qquad \Gamma \vdash \tilde{e} \triangleright \tilde{S'}}{\Theta' \cdot X' : \tilde{S'}\tilde{\alpha'}; \Gamma \vdash X'[\tilde{e}\tilde{k}] \triangleright \Delta' \cdot \tilde{k} : \tilde{\alpha'}} \text{ [Var]}$$

Con lo cual $P \equiv X'[\tilde{e}\tilde{k}]$, $\Theta \equiv \Theta' \cdot X' : \tilde{S'}\tilde{\alpha'}$, $\Delta \equiv \Delta' \cdot \tilde{k} : \tilde{\alpha'}$ y Δ' es completa. Además como por hipótesis $X \notin \text{dom}(\Theta)$, entonces también se cumple necesariamente que $X \notin \text{dom}(\Theta')$ y $X \neq X'$, con lo cual están bien definidas las expresiones (por [6] pág. 81) $\Theta' \cdot X : \tilde{S}\tilde{\alpha}$ y $\Theta' \cdot X : \tilde{S}\tilde{\alpha} \cdot X' : \tilde{S'}\tilde{\alpha'}$

Además se cumple que:

$$\begin{gathered}\text{Si } X \notin \text{dom}(\Theta'),\ X' \notin \text{dom}(\Theta') \text{ y } X \neq X', \text{entonces} \\ (\Theta' \cdot X' : \tilde{S'}\tilde{\alpha'} \cdot X : \tilde{S}\tilde{\alpha}) \equiv (\Theta' \cdot X : \tilde{S}\tilde{\alpha} \cdot X' : \tilde{S'}\tilde{\alpha'})\end{gathered} \tag{A.1}$$

Podemos entonces construir la siguiente derivación:

$$\text{(A.1)}\ \frac{\dfrac{\Delta' \text{ completa} \qquad \Gamma \vdash \tilde{e} \triangleright \tilde{S'}}{\Theta' \cdot X : \tilde{S}\tilde{\alpha} \cdot X' : \tilde{S'}\tilde{\alpha'}; \Gamma \vdash X'[\tilde{e}\tilde{k}] \triangleright \Delta' \cdot \tilde{k} : \tilde{\alpha'}} \text{ [Var]}}{\Theta' \cdot X' : \tilde{S'}\tilde{\alpha'} \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash X'[\tilde{e}\tilde{k}] \triangleright \Delta' \cdot \tilde{k} : \tilde{\alpha'}}$$

Sustituyendo variables obtenemos: $\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash P \triangleright \Delta$, quedando finalmente demostrado este paso base.

Paso Inductivo [Acc]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\frac{\Gamma \vdash a \triangleright \sigma(\alpha') \qquad \dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k : \alpha'}}{\Theta; \Gamma \vdash \texttt{accept}\ a(k)\ \texttt{in}\ P' \triangleright \Delta} \text{ [Acc]}$$

Con lo cual $P \equiv \texttt{accept}\ a(k)\ \texttt{in}\ P'$.

hi.) Si $X \notin \text{dom}(\Theta)$, entonces $\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash P' \triangleright \Delta \cdot k : \alpha'$.

Como $X \notin \text{dom}(\Theta)$ podemos entonces construir el siguiente árbol de derivación:

$$\frac{\Gamma \vdash a \triangleright \sigma(\alpha') \qquad \text{por hi. (como } X \notin \text{dom}(\Theta)) \dfrac{\vdots}{\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash P' \triangleright \Delta \cdot k : \alpha'}}{\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash \texttt{accept}\ a(k)\ \texttt{in}\ P' \triangleright \Delta} \text{ [Acc]}$$

Obteniendo que $\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash P \triangleright \Delta$

Pasos Inductivos [Req], [Send], [Rcv], [Br], [Sel], [Thr], [Cat], [Conc], [If], [NRes], [CRes] y [CRes']

Análogos al anterior, estudiando las reglas se cumple que Θ no crece hacia arriba en el árbol de derivación en estas reglas. La única excepción a lo anterior sucede en [Def], por lo que se demuestra a continuación dicha excepción.

Paso Inductivo [Def]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\dfrac{\dfrac{\vdots}{\Theta \cdot X' : \tilde{S'}\tilde{\alpha'}; \Gamma \cdot \tilde{x} : \tilde{S'} \vdash P' \triangleright \tilde{k} : \tilde{\alpha'}} \quad \dfrac{\vdots}{\Theta \cdot X' : \tilde{S'}\tilde{\alpha'}; \Gamma \vdash Q \triangleright \Delta}}{\Theta; \Gamma \vdash \texttt{def}\ X'(\tilde{x}\tilde{k}) = P'\ \texttt{in}\ Q \triangleright \Delta}\ [\text{Def}]$$

Con lo cual $P \equiv \texttt{def}\ X'(\tilde{x}\tilde{k}) = P'\ \texttt{in}\ Q$.

hi. 1) Si $X \notin \text{dom}(\Theta \cdot X' : \tilde{S'}\tilde{\alpha'})$, entonces $\Theta \cdot X' : \tilde{S'}\tilde{\alpha'} \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \cdot \tilde{x} : \tilde{S'} \vdash P' \triangleright \tilde{k} : \tilde{\alpha'}$.

hi. 2) Si $X \notin \text{dom}(\Theta \cdot X' : \tilde{S'}\tilde{\alpha'})$, entonces $\Theta \cdot X' : \tilde{S'}\tilde{\alpha'} \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash Q \triangleright \Delta$.

x

1. Caso $X \neq X'$

   Como $X \notin \text{dom}(\Theta)$, tenemos que $X \notin \text{dom}(\Theta \cdot X' : \tilde{S'}\tilde{\alpha'})$, podemos utilizar (A.1), y construir el siguiente árbol de derivación:

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ \text{por hi. 1) (como } X \notin \text{dom}(\Theta \cdot X' : \tilde{S'}\tilde{\alpha'})) \\ \text{y por (A.1)}\end{array}}{\Theta \cdot X : \tilde{S}\tilde{\alpha} \cdot X' : \tilde{S'}\tilde{\alpha'}; \Gamma \cdot \tilde{x} : \tilde{S'} \vdash P' \triangleright \tilde{k} : \tilde{\alpha'}} \quad \dfrac{\begin{array}{c}\vdots \\ \text{por hi. 2) (como } X \notin \text{dom}(\Theta \cdot X' : \tilde{S'}\tilde{\alpha'})) \\ \text{y por (A.1)}\end{array}}{\Theta \cdot X : \tilde{S}\tilde{\alpha} \cdot X' : \tilde{S'}\tilde{\alpha'}; \Gamma \vdash Q \triangleright \Delta}}{\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash \texttt{def}\ X'(\tilde{x}\tilde{k}) = P'\ \texttt{in}\ Q \triangleright \Delta}\ [\text{Def}]$$

2. Caso $X = X'$

   Utilizando α-conversión $\texttt{def}\ X'(\tilde{x}\tilde{k}) = P'\ \texttt{in}\ Q \equiv_\alpha \texttt{def}\ X'(\tilde{x}\tilde{k}) = P'[X'/X]\ \texttt{in}\ Q[X'/X]$ con $X' \notin \text{fpv}(P')$ y $X' \notin \text{fpv}(Q')$, tenemos así que podemos renombrar la variable de proceso X' en el proceso $\texttt{def}\ X'(\tilde{x}\tilde{k}) = P'\ \texttt{in}\ Q$ por otra variable X'' tal que $X'' \neq X$, y $X'' \notin \text{fpv}(P')$ y $X'' \notin \text{fpv}(Q')$. En este caso se vuelve a estar en las premisas del caso anterior.

Obteniendo que $\Theta \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \vdash P \triangleright \Delta$

Estas son todas las reglas de tipado (fig. 6 [6]), quedando entonces demostrado 1 inductivamente en el árbol de derivación del tipo.

A.2. Demostración de (2)

Caso Base [INACT]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\frac{\Delta \text{ completa}}{\Theta; \Gamma \vdash \texttt{inact} \triangleright \Delta}\ [\text{INACT}]$$

Con lo cual $P \equiv \texttt{inact}$ y Δ es completa, como $a \notin \text{dom}(\Gamma)$ por [6] pág. 81) está bien definido $\Gamma \cdot a : S$ y podemos construir la siguiente derivación:

$$\frac{\Delta \text{ completa}}{\Theta; \Gamma \cdot a : S \vdash \texttt{inact} \triangleright \Delta}\ [\text{INACT}]$$

Obteniendo que $\Theta; \Gamma \cdot a : S \vdash P \triangleright \Delta$

Caso Base [VAR]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\frac{\Delta' \text{ completa} \qquad \Gamma \vdash \tilde{e} \triangleright \tilde{S'}}{\Theta' \cdot X : \tilde{S'}\tilde{\alpha}; \Gamma \vdash X[\tilde{e}\tilde{k}] \triangleright \Delta' \cdot \tilde{k} : \tilde{\alpha}}\ [\text{VAR}]$$

Con lo cual $P \equiv X[\tilde{e}\tilde{k}]$, $\Theta \equiv \Theta' \cdot X : \tilde{S'}\tilde{\alpha}$, $\Delta \equiv \Delta' \cdot \tilde{k} : \tilde{\alpha}$ y Δ' es completa. Como $a \notin \text{dom}(\Gamma)$ ([6] pág. 81) se cumple que podemos escribir $\Gamma \cdot a : S$. Previamente tenemos que demostrar que:

$$a \notin \text{dom}(\Gamma) \text{ y } \Gamma \vdash e \triangleright S', \text{ entonces } \Gamma \cdot a : S \vdash e \triangleright S'. \tag{A.2}$$

Demostramos la anterior proposición también por inducción en el árbol de derivación de $\Gamma \vdash e \triangleright S'$.

1. Caso Base [NAMEI] El árbol de derivación de $\Gamma \vdash e \triangleright S'$ es:

$$\frac{}{\Gamma' \cdot a' : S' \vdash a' \triangleright S'}\ [\text{NAMEI}]$$

Donde $\Gamma \equiv \Gamma' \cdot a' : S'$ y $e \equiv a'$. Como $a \notin \text{dom}(\Gamma)$ se cumple $a \notin \text{dom}(\Gamma')$ y $a \neq a'$, podemos (por [6] pág. 81) entonces escribir $\Gamma' \cdot a : S$ y a su vez $\Gamma' \cdot a : S \cdot a' : S'$. Con lo cual podemos construir el siguiente árbol de derivación:

$$\frac{}{\Gamma' \cdot a : S \cdot a' : S' \vdash a' \triangleright S'}\ [\text{NAMEI}]$$

A su vez por (A.1) $(\Gamma' \cdot a' : S' \cdot a : S) \equiv (\Gamma' \cdot a : S \cdot a' : S')$, obteniendo que $\Gamma' \cdot a' : S' \cdot a : S \vdash a' \triangleright S'$, lo que equivale cambiando variables a: $\Gamma \cdot a : S \vdash e \triangleright S'$.

2. Caso Base [Nat]

   El árbol de derivación de $\Gamma \vdash e \triangleright S'$ es:

$$\frac{}{\Gamma \vdash 1 \triangleright \text{nat}}\ [\text{Nat}]$$

   Donde $e \equiv 1$ y $S' \equiv \text{nat}$. Como $a \notin \text{dom}(\Gamma)$ podemos ([6] pág. 81) escribir $\Gamma \cdot a : S$. Con lo cual podemos construir el siguiente árbol de derivación:

$$\frac{}{\Gamma \cdot a : S \vdash 1 \triangleright \text{nat}}\ [\text{Nat}]$$

   Lo que finalmente equivale cambiando variables a: $\Gamma \cdot a : S \vdash e \triangleright S'$.

3. Caso Base [Bool]

   Ambos casos `true` y `false` son análogos al anterior.

4. Paso Inductivo [Sum]

   El árbol de derivación de $\Gamma \vdash e \triangleright S'$ es:

$$\frac{\Gamma \vdash e_i \triangleright \text{nat}}{\Gamma \vdash e_1 + e_2 \triangleright \text{nat}}\ [\text{Sum}]$$

   Donde $e \equiv e_1 + e_2$ y $S' \equiv \text{nat}$. Como $a \notin \text{dom}(\Gamma)$ podemos escribir $\Gamma \cdot a : S$.

   hi.) Si $a \notin \text{dom}(\Gamma)$ entonces $\Gamma \cdot a : S \vdash e_i \triangleright \text{nat}$.

   Con lo cual podemos construir el siguiente árbol de derivación:

$$\frac{\text{(por hi.), como } a \notin \text{dom}(\Gamma))\ \Gamma \cdot a : S \vdash e_i \triangleright \text{nat}}{\Gamma \cdot a : S \vdash e_1 + e_2 \triangleright \text{nat}}\ [\text{Sum}]$$

   Lo que finalmente equivale cambiando variables a: $\Gamma \cdot a : S \vdash e \triangleright S'$.

Estos son todos los casos posibles de árboles de derivación de $\Gamma \vdash e \triangleright S'$, con lo cual queda entonces demostrado (A.2).

Del resultado anterior (A.2) y como $a \notin \text{dom}(\Gamma)$, tenemos entonces que $\Gamma \cdot a : S \vdash \tilde{e} \triangleright \tilde{S}$, tenemos así el siguiente árbol de derivación:

$$\frac{\Delta' \text{ completa} \qquad \text{por (A.2)}\ \dfrac{\Gamma \vdash \tilde{e} \triangleright \tilde{S'}}{\Gamma \cdot a : S \vdash \tilde{e} \triangleright \tilde{S'}}}{\Theta' \cdot X : \tilde{S}\tilde{\alpha}; \Gamma \cdot a : S \vdash X'[\tilde{e}\tilde{k}] \triangleright \Delta' \cdot \tilde{k} : \tilde{\alpha'}}\ [\text{Var}]$$

Obteniendo que $\Theta; \Gamma \cdot a : S \vdash P \triangleright \Delta$.

Paso Inductivo [Acc]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\dfrac{\Gamma \vdash a' \triangleright \sigma(\alpha') \qquad \dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k : \alpha'}}{\Theta; \Gamma \vdash \texttt{accept}\ a'(k)\ \texttt{in}\ P' \triangleright \Delta}\ [\textsc{Acc}]$$

Con lo cual $P \equiv \texttt{accept}\ a'(k)\ \texttt{in}\ P'$.
hi.) Si $a \notin \text{dom}(\Gamma)$, entonces $\Theta; \Gamma \cdot a : S \vdash P' \triangleright \Delta \cdot k : \alpha'$.
Como $a \notin \text{dom}(\Gamma)$, utilizando el resultado previo (A.2), podemos entonces construir el siguiente árbol de derivación:

$$\dfrac{\text{por (A.2)}\ \dfrac{\Gamma \vdash a' \triangleright \sigma(\alpha')}{\Gamma \cdot a : S \vdash a' \triangleright \sigma(\alpha')} \qquad \text{por hi.)}\ \dfrac{\vdots}{\Theta; \Gamma \cdot a : S \vdash P' \triangleright \Delta \cdot k : \alpha'}}{\Theta; \Gamma \cdot a : S \vdash \texttt{accept}\ a'(k)\ \texttt{in}\ P' \triangleright \Delta}\ [\textsc{Acc}]$$

Obteniendo que $\Theta; \Gamma \cdot a : S \vdash P \triangleright \Delta$

Pasos Inductivos [Req], [Send], [Br], [Sel], [Thr], [Cat], [Conc], [If], [CRes] y [CRes']

Análogos al anterior, estudiando las reglas, se cumple que Γ no crece hacia arriba en el árbol de derivación. La única excepción a lo anterior sucede en [Rcv], [NRes] y [Def] , por lo que se demuestran un casos de estos a continuación.

Paso Inductivo [Rcv]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\dfrac{\dfrac{\vdots}{\Theta; \Gamma \cdot \tilde{x} : \tilde{S} \vdash P' \triangleright \Delta' \cdot k : \alpha}}{\Theta; \Gamma \vdash k?(\tilde{x})\ \texttt{in}\ P' \triangleright \Delta' \cdot k :?[\tilde{S}]; \alpha}\ [\textsc{Rcv}]$$

Con lo cual $P \equiv k?(\tilde{x})\ \texttt{in}\ P'$ y $\Delta \equiv \Delta' \cdot k :?[\tilde{S}]; \alpha$.
hi.) Si $a \notin \text{dom}(\Gamma \cdot \tilde{x} : \tilde{S})$, entonces $\Theta; \Gamma \cdot \tilde{x} : \tilde{S} \cdot a : S \vdash P' \triangleright \Delta' \cdot k : \alpha$.

1. Caso $a \notin \tilde{x}$

   Como $a \notin \tilde{x}$ y $a \notin \Gamma$, se cumple que $a \notin \text{dom}(\Gamma \cdot \tilde{x} : \tilde{S})$, con lo cual podemos aplicar la hipótesis inductiva y $(\Gamma \cdot \tilde{x} : \tilde{S} \cdot a : S) \equiv (\Gamma \cdot a : S \cdot \tilde{x} : \tilde{S})(*)$ y construir el siguiente árbol .

$$\text{por hi.) y (*)}\ \frac{\dfrac{\vdots}{\Theta;\Gamma\cdot\tilde{x}:\tilde{S}\vdash P'\triangleright\Delta'\cdot k:\alpha}}{\dfrac{\Theta;\Gamma\cdot a:S\cdot\tilde{x}:\tilde{S}\vdash P'\triangleright\Delta'\cdot k:\alpha}{\Theta;\Gamma\cdot a:S\vdash k?(\tilde{x})\ \texttt{in}\ P'\triangleright\Delta'\cdot k:?[\tilde{S}];\alpha}\ [\textsc{Rcv}]}$$

2. Caso $a\in\tilde{x}$

   Podemos utilizar α-conversión, así $k?(\tilde{x})\ \texttt{in}\ P'\equiv_\alpha k?(\tilde{x'})\ \texttt{in}\ P'[\tilde{x'}/\tilde{x}]$ con $x'\notin \text{fn}(P')$ y $a\notin\tilde{x'}$ en este caso se vuelve a estar en las premisas del caso anterior.

Pasos Inductivos [NRes] y [Def]

Análogos al anterior.

Estas son todas las reglas de tipado (fig. 6 [6]), quedando entonces demostrado 2 inductivamente en el árbol de derivación del tipo.

A.3. Demostración de (3)

Caso Base [Inact]

El árbol de derivación de $\Theta;\Gamma\vdash P\triangleright\Delta$ es el siguiente:

$$\frac{\Delta\text{ completa}}{\Theta;\Gamma\vdash\texttt{inact}\triangleright\Delta}\ [\textsc{Inact}]$$

Con lo cual $P\equiv\texttt{inact}$ y Δ es completa, como $k\notin\text{dom}(\Delta)$ (por [6] pág. 81) podemos escribir $\Delta\cdot k:\texttt{end}$, luego:

$$\frac{\Delta\cdot k:\texttt{end}\text{ completa}}{\Theta;\Gamma\vdash\texttt{inact}\triangleright\Delta\cdot k:\texttt{end}}\ [\textsc{Inact}]$$

Obteniendo en ambos casos que $\Theta;\Gamma\vdash P\triangleright\Delta\cdot k:\texttt{end}$.

Caso Base [Var]

El árbol de derivación de $\Theta;\Gamma\vdash P\triangleright\Delta$ es el siguiente:

$$\frac{\Delta'\text{ completa}\qquad\Gamma\vdash\tilde{e}\triangleright\tilde{S'}}{\Theta'\cdot X':\tilde{S'}\tilde{\alpha'};\Gamma\vdash X'[\tilde{e}\tilde{k'}]\triangleright\Delta'\cdot\tilde{k'}:\tilde{\alpha'}}\ [\textsc{Var}]$$

Con lo cual $P\equiv X'[\tilde{e}\tilde{k'}]$, $\Theta\equiv\Theta'\cdot X':\tilde{S'}\tilde{\alpha'}$, $\Delta\equiv\Delta'\cdot\tilde{k'}:\tilde{\alpha'}$ y Δ' es completa, como $a\notin\text{dom}(\Delta)$ (por [6] pág. 81), podemos escribir $\Delta\cdot a:\tilde{\alpha}$.

Además se cumple que $\Delta'\cdot k:\texttt{end}\cdot\tilde{k'}:\tilde{\alpha'}\equiv\Delta'\cdot\tilde{k'}:\tilde{\alpha'}\cdot k:\texttt{end}$ (*).

$$\text{por (*)}\ \frac{\dfrac{\Delta'\cdot k:\texttt{end}\text{ completa}\qquad\Gamma\vdash\tilde{e}\triangleright\tilde{S'}}{\Theta'\cdot X':\tilde{S'}\tilde{\alpha'};\Gamma\vdash X'[\tilde{e}\tilde{k'}]\triangleright\Delta'\cdot k:\texttt{end}\cdot\tilde{k'}:\tilde{\alpha'}}\ [\textsc{Var}]}{\Theta'\cdot X':\tilde{S'}\tilde{\alpha'};\Gamma\vdash X'[\tilde{e}\tilde{k'}]\triangleright\Delta'\cdot\tilde{k'}:\tilde{\alpha'}\cdot k:\texttt{end}}$$

Obteniendo en ambos casos que $\Theta;\Gamma\vdash P\triangleright\Delta\cdot k:\texttt{end}$.

Paso Inductivo [Acc]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\dfrac{\Gamma \vdash a \triangleright \sigma(\alpha') \qquad \dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k' : \alpha'}}{\Theta; \Gamma \vdash \texttt{accept}\ a(k')\ \texttt{in}\ P' \triangleright \Delta}\ [\textsc{Acc}]$$

Con lo cual $P \equiv \texttt{accept}\ a(k')\ \texttt{in}\ P'$.
hi.) Si $k \notin \mathrm{dom}(\Delta \cdot k' : \alpha')$, entonces $\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k' : \alpha' \cdot k : \texttt{end}$.

1. Caso $k \neq k'$

   Como $k \notin \mathrm{dom}(\Delta)$ y $k \neq k'$, podemos escribir $\Delta \cdot k' : \alpha' \cdot k : \texttt{end}$, y $(\Delta \cdot k' : \alpha' \cdot k : \texttt{end}) \equiv (\Delta \cdot k : \texttt{end} \cdot k' : \alpha')$ (*), entonces tenemos el siguiente árbol de derivación:

$$\dfrac{\Gamma \vdash a \triangleright \sigma(\alpha') \qquad \begin{array}{r}\text{por hi. } (k \neq k') \\ \text{por } (*)\end{array} \dfrac{\dfrac{\dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k' : \alpha'}}{\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k' : \alpha' \cdot k : \texttt{end}}}{\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k : \texttt{end} \cdot k' : \alpha'}}{\Theta; \Gamma \vdash \texttt{accept}\ a(k')\ \texttt{in}\ P' \triangleright \Delta \cdot k : \texttt{end}}\ [\textsc{Acc}]$$

2. Caso $k = k'$

   Utilizando α-conversión, tenemos que $\texttt{accept}\ a(k')\ \texttt{in}\ P' \equiv_\alpha \texttt{accept}\ a(k'')\ \texttt{in}\ P'[k''/k']$ con $k'' \notin \mathrm{fc}(P')$ y $k \neq k''$ en este caso se vuelve a estar en las premisas del caso anterior.

Obteniendo en ambos casos que $\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \texttt{end}$.

Paso Inductivo [Req], [CRes]

Análogo al anterior.

Paso Inductivo [Send], [Rcv], [Br], [Sel], [Def]

Análogos a [Bot].

Paso Inductivo [Thr]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\dfrac{\dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta' \cdot k' : \beta}}{\Theta; \Gamma \vdash \texttt{throw}\ k'[k'']; P' \triangleright \Delta' \cdot k' :![\alpha']; \beta \cdot k'' : \alpha'}\ [\textsc{Thr}]$$

Tenemos entonces que $P \equiv \texttt{throw}\ k'[k'']; P'$ y $\Delta \equiv \Delta' \cdot k' :![\alpha']; \beta \cdot k'' : \alpha'$.

hi.) Si $k \notin \mathrm{dom}(\Delta' \cdot k' : \beta)$ entonces $\Theta; \Gamma \vdash P' \triangleright \Delta' \cdot k' : \beta \cdot k : \texttt{end}$.

Como $k \notin \mathrm{dom}(\Delta)$, podemos escribir $\Delta' \cdot k' :![\alpha']; \beta \cdot k'' : \alpha' \cdot k : \texttt{end}$, $(\Delta' \cdot k' : \beta \cdot k : \texttt{end}) \equiv (\Delta' \cdot k : \texttt{end} \cdot k' : \beta)$ (*) y $(\Delta' \cdot k' :![\alpha']; \beta \cdot k'' : \alpha' \cdot k : \texttt{end}) \equiv (\Delta' \cdot k : \texttt{end} \cdot k' :![\alpha']; \beta \cdot k'' : \alpha')$ (**)

$$\text{por (**)}\ \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta' \cdot k' : \beta}}{\Theta; \Gamma \vdash P' \triangleright \Delta' \cdot k : \texttt{end} \cdot k' : \beta}\ \text{por (*)}}{\Theta; \Gamma \vdash \texttt{throw}\ k'[k'']; P' \triangleright \Delta' \cdot k : \texttt{end} \cdot k' :![\alpha']; \beta \cdot k'' : \alpha'}\ \text{por hi.}}{\Theta; \Gamma \vdash \texttt{throw}\ k'[k'']; P' \triangleright \Delta' \cdot k' :![\alpha']; \beta \cdot k'' : \alpha' \cdot k : \texttt{end}}\ [\textsc{Thr}]$$

Obteniendo que $\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \texttt{end}$.

Paso Inductivo [Cat]

Análogo a [Acc].

Paso Inductivo [Conc]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\dfrac{\dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta'} \qquad \dfrac{\vdots}{\Theta; \Gamma \vdash Q \triangleright \Delta''}}{\Theta; \Gamma \vdash P' | Q \triangleright \Delta' \otimes \Delta''}\ [\textsc{Conc}]$$

hi. 1) Si $k \notin \mathrm{dom}(\Delta')$ entonces $\Theta; \Gamma \vdash P' \triangleright \Delta' \cdot k : \texttt{end}$.

hi. 2) Si $k \notin \mathrm{dom}(\Delta'')$ entonces $\Theta; \Gamma \vdash Q \triangleright \Delta'' \cdot k : \texttt{end}$.

Como $k \notin \Delta' \otimes \Delta''$, por definición de la operación $\otimes$ (sección 2.6), tenemos que $k \notin \Delta'$ y $k \notin \Delta''$.

Podemos entonces construir el siguiente árbol.

$$\text{por hi. 1) } (k \notin \Delta')\ \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta'}}{\Theta; \Gamma \vdash P' \triangleright \Delta' \cdot k : \texttt{end}} \qquad \dfrac{\vdots}{\Theta; \Gamma \vdash Q \triangleright \Delta''}}{\Theta; \Gamma \vdash P' | Q \triangleright (\Delta' \cdot k : \texttt{end}) \otimes \Delta''}}{\Theta; \Gamma \vdash P' | Q \triangleright (\Delta' \otimes \Delta'') \cdot k : \texttt{end}}\ [\textsc{Conc}]$$

Obteniendo que $\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \texttt{end}$.

Paso Inductivo [If]

El árbol de derivación de $\Theta; \Gamma \vdash P \triangleright \Delta$ es el siguiente:

$$\dfrac{\Gamma \vdash e \triangleright \text{bool} \qquad \dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta} \qquad \dfrac{\vdots}{\Theta; \Gamma \vdash Q \triangleright \Delta}}{\Theta; \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P'\ \texttt{else}\ Q \triangleright \Delta}\ [\textsc{If}]$$

Donde $P \equiv \texttt{if}\ e\ \texttt{then}\ P'\ \texttt{else}\ Q$.
hi. 1) Si $k \notin \text{dom}(\Delta)$ entonces $\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k : \texttt{end}$.
hi. 2) Si $k \notin \text{dom}(\Delta)$ entonces $\Theta; \Gamma \vdash Q \triangleright \Delta \cdot k : \texttt{end}$.
Podemos entonces construir el siguiente árbol.

$$\dfrac{\Gamma \vdash e \triangleright \text{bool} \qquad \text{por hi. 1}\ \dfrac{\dfrac{\vdots}{\Theta; \Gamma \vdash P' \triangleright \Delta}}{\Theta; \Gamma \vdash P' \triangleright \Delta \cdot k : \texttt{end}} \qquad \text{por hi. 2}\ \dfrac{\dfrac{\vdots}{\Theta; \Gamma \vdash Q \triangleright \Delta}}{\Theta; \Gamma \vdash Q \triangleright \Delta \cdot k : \texttt{end}}}{\Theta; \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P'\ \texttt{else}\ Q \triangleright \Delta \cdot k : \texttt{end}}\ [\textsc{If}]$$

Obteniendo que $\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \texttt{end}$.

Paso Inductivo [NRes]

Análogo al paso inductivo [If].

Estas son todas las reglas de tipado (fig. 6 [6]), quedando entonces demostrado 3 inductivamente en el árbol de derivación del tipo.

Apéndice B

Propiedades de la Unificación

B.1. Unificación de Tipos Básicos

B.1.1. Idempotencia

En esta sección probamos el lema `IdemToIdempotentS` que establece que una condición suficiente para que una sustitución sea idempotente es que las variables de tipo que aparecen en el dominio y el codominio de la sustitución sean disjuntos.

En las siguientes figuras repetimos las definiciones dadas en 6.1.4.

```
IdempotentS : (ss : List (ℕ × Sort)) → Set
IdempotentS ss = (s : Sort) →
                 lsubS ss (lsubS ss s) ≡ lsubS ss s
```

```
  data IdemS : List (ℕ × Sort) → Set where
  []idem : IdemS []
  ::idem : (ss : List (ℕ × Sort))(n : ℕ)(s : Sort) →
           IdemS ss →
           disjoin (varsS s) (dom ((n , s) :: ss)) →
           n ∉ dom ss → n ∉ varsSS ss →
           IdemS ((n , s) :: ss)
```

El siguiente resultado establece que dada una variable n, dos tipos básicos $s1$ y $s2$, y una sustitución ss, si n y $s1$ tienen la misma imagen bajo la sustitución ss, entonces los tipos $s2$ y $s2$ $\langle n, s1\rangle$ también tienen la misma imagen bajo ss. Esta propiedad se demuestra por inducción en el tipo básico $s1$.

Como es esperable, si el conjunto de variables en el dominio de una sustitución y el conjunto de variables que aparece en un tipo básico son disjuntos, entonces no tiene efecto aplicar la sustitución al tipo básico. Esta propiedad se demuestra por inducción en la sustitución.

```
≡underSS : (n : ℕ)(s1 s2 : Sort)(ss : List (ℕ × Sort)) →
           lsubS ss (varS n) ≡ lsubS ss s1 →
           lsubS ss s2 ≡ lsubS ss (subSS (n , s1) s2)
```

Figura B.1: Igualdad de imagenes bajo una sustitución

```
≡disjoinVarSS : (s : Sort)(ss : List (ℕ × Sort)) →
                disjoin (varsS s) (dom ss) →
                s ≡ lsubS ss s
```

Ahora definimos la propiedad que si una sustitución *ss* es idempotente, entonces el conjunto de las variables de tipo en el tipo t *ss* para un tipo básico t cualquiera, y el dominio de *ss* son disjuntos. Esta prueba ...

```
disj-idemSS : (s : Sort)(ss : List (ℕ × Sort)) → IdemS ss →
              disjoin (varsS (lsubS ss s)) (dom ss)
```

Finalmente probamos el resultado deseado sobre la idempotencia:

```
IdemToIdempotentS : (ss : List (ℕ × Sort)) → IdemS ss →
                    IdempotentS ss
IdemToIdempotentS ss idemSss s
        =  sym (≡disjoinVarSS (lsubS ss s)
                              ss
                              (disj-idemSS s ss idemSss))
```

La demostración es directa, dado que cuando una sustitución *ss* satisface `IdemS`, utilizando el resultado `disj-idemSS` sabemos que al aplicar *ss* a un tipo cualquiera no van a quedar variables de tipo del dominio de *ss*, entonces aplicando ahora `≡disjoinVarSS` sabemos que aplicar nuevamente *ss* no va cambiar el resultado.

B.1.2. Corrección de la Existencia de un Resultado

Los dos siguientes resultados prueban que el resultado del algoritmo de unificación no dice que no existe sustitución si y sólo si existe una sustitución que unifique la lista de pares dada como argumento.

Para probar estos dos resultados utilizamos el lema auxiliar `error∧unifies→⊥` cuya prueba damos a continucación:

A su vez, para demostrar el anterior lema utilizamos el resultado auxiliar `error∧unifies→⊥-acc` el cual recibe como parámetro extra una demostración de que la lista de entrada satisface el predicado `UniLS-Acc` presentado anteriormente en la figura 6.3. Hacemos la demostración por recursión en este parámetro extra. Cuando este parámetro es el caso base `uniLS[]` entonces dado que en este caso la función `unifyLS-ac` devuelve el acumulador (ver figura 6.5), entonces no existe la demostración de que esta función devuelva `nothing`, esto

```
unifiesS→¬error : (lps : List (Sort × Sort)) →
                  ∃ (λ ss → unifiesLPS ss lps) →
                  unifyLS lps ≢ nothing
unifiesS→¬error lps (ss , unifiesLPSsslps) unifyLSlps≡noth
  = error∧unifies→⊥ lps (unifyLSlps≡noth ,
                         (ss , unifiesLPSsslps))
```

Figura B.2: Propiedad de existencia 1

```
error→¬unifiesS : (lps : List (Sort × Sort)) →
                  unifyLS lps ≡ nothing  →
                  ¬ ∃ (λ ss → (unifiesLPS ss lps))
error→¬unifiesS lps unifyLSlps≡noth (ss , unifiesLPSsslps)
   = error∧unifies→⊥ lps (unifyLSlps≡noth ,
                          (ss , unifiesLPSsslps))
```

Figura B.3: Propiedad de existencia 2

es, diga que no existe unificación posible. Los demás casos inductivos actúan sobre una lista de pares de pares de tipos básicos no vacía $\langle s_1, s_2\rangle :: lsp$, donde la demostración de que existe una sustitución ss que unifica esta lista se convierte en $unifysss1s2 :: unifysslps$, donde $unifysss1s2$ es una demostración de que $s_1\ ss \equiv s_2\ ss$ y $unifysslps$ es una demostración de que el resto de la lista lps unifica bajo la sustitución ss . En el caso inductivo `uniLSnoth` utilizamos el resultado auxiliar `unifySNoth` que demuestra que no puede existir sustitución alguna que unifique s_1 y s_2.

```
error∧unifies→⊥ : (lps : List (Sort × Sort)) →
                  (unifyLS lps ≡ nothing) ×
                  ∃ (λ ss → (unifiesLPS ss lps)) → ⊥
error∧unifies→⊥ lps = error∧unifies→⊥-acc lps
                                          []
                                          (allListUniLS-Acc lps)
```

```
unifySNoth : (ps : Sort × Sort) →
             unifyS (proj₁ ps) (proj₂ ps) ≡ nothing →
             ¬ ∃ (λ ss → lsubS ss (proj₁ ps) ≡
                         lsubS ss (proj₂ ps))
```

El caso inductivo `uniLSjustnoth` dado que `unifyLS-ac` reduce a una llamada recursiva con el resto de la lista lps y el mismo acumulador, entonces utilizamos la hipótesis inductiva de que el resultados se cumple para la lista lps. Finalmente en el caso inductivo `uniLSjustjust` utilizamos el resultado auxiliar `unifySjustJust→s1s2var` que da información de los tipos básicos s_1 y s_2 y la sustitución s que los unifica. Esta información dice que existen dos casos:

- $s_1 \equiv varS\ n \wedge s \equiv \langle n, s_2 \rangle \wedge n \notin varsS\ s_2$ en este caso usamos la hipótesis inductiva para la lista $lps\ \langle n, s_2 \rangle$, con el correspondiente acumulador, la demostración de que esta lista es accesible, y finalmente para dar una demostración de que existe una sustitución que unifique ahora $lps\ \langle n, s_2 \rangle$, usamos que tenemos una sustitución ss que unifica lps y que $(varS\ n)\ ss \equiv s_2\ ss$, este resultado viene en la expresión $unifysss1s2$. Para así usar el resultado auxiliar `aux` que establece que esta misma sustitución ss unifica también $lps\ \langle n, s_2 \rangle$.
- $s_2 \equiv varS\ n \wedge s \equiv \langle n, s_1 \rangle \wedge n \notin varsS\ s_1$ este caso en análogo al anterior.

```
unifySjustJust→s1s2var : (s1 s2 : Sort)(s : (ℕ × Sort)) →
                         unifyS s1 s2 ≡ just (just s) →
                         ∃ (λ n → (s1 ≡ varS n ×
                                   s ≡ (n , s2) ×
                                   n ∉ varsS s2)
                                   ⊎
                                  (s2 ≡ varS n ×
                                  s ≡ (n , s1) ×
                                  n ∉ varsS s1))
```

```
aux : (n : ℕ)(s : Sort)(ss : List (ℕ × Sort))
      (lps : List (Sort × Sort)) →
      lsubS ss (varS n) ≡ lsubS ss s →
      unifiesLPS ss lps →
      unifiesLPS ss (subSLS (n , s) lps)
```

```
error∧unifies→⊥-acc : (lps : List (Sort × Sort))
                      (ac : List (ℕ × Sort)) →
                      (p : UniLS-Acc lps) →
                      (unifyLS-ac lps p ac ≡ nothing) ×
                      ∃ (λ ss → (unifiesLPS ss lps)) → ⊥
error∧unifies→⊥-acc .[] ac uniLS[] (() , (ss , unifiessslps))
error∧unifies→⊥-acc .((s1 , s2) :: lps) ac
                  (uniLSnoth s1 s2 lps unifys1s2≡noth)
                  (unifyLS≡noth ,
                  (ss , (unifysss1s2 :: unifysslps)))
  = ((unifySNoth (s1 , s2) unifys1s2≡noth) (ss , unifysss1s2))
error∧unifies→⊥-acc .((s1 , s2) :: lps) ac
                  (uniLSjustnoth s1 s2 lps
                                 unifySs1s2≡justnoth uni-acclps)
                  (unifyLS≡noth , (ss ,
                                   (unifysss1s2 :: unifysslps)))
  =  error∧unifies→⊥-acc lps ac uni-acclps
                         (unifyLS≡noth  , (ss , unifysslps))
error∧unifies→⊥-acc .((s1 , s2) :: lps) ac
                  (uniLSjustjust s1 s2 lps
                                 (u , (unifySs1s2≡justjustu ,
                                       uni-accsubSLSulps)))
                  (unifyLS≡noth ,
                   (ss , (unifysss1s2 :: unifysslps)))
  with unifySjustJust→unifyss1s2 s1 s2 u unifySs1s2≡justjustu
... | subSSus1≡subSSus2
                     with unifySjustJust→s1s2var s1 s2 u
                                            unifySs1s2≡justjustu
error∧unifies→⊥-acc .((varS n , s2) :: lps) ac
                     (uniLSjustjust .(varS n) s2 lps
                                    (.(n , s2) ,
                                     (unifySs1s2≡justjustu ,
                                      uni-accsubSLSulps)))
                     (unifyLS≡noth ,
                      (ss , (unifysss1s2 :: unifysslps)))
    | subSSus1≡subSSus2 | n , inj₁ (refl , refl , _)
  = error∧unifies→⊥-acc (subSLS (n , s2) lps)
                  ((n , s2) :: (subSSS (n , s2) ac))
                  uni-accsubSLSulps
                  (unifyLS≡noth ,
                   (ss  , aux n s2 ss lps unifysss1s2
                              unifysslps))
error∧unifies→⊥-acc .((s1 , varS n) :: lps) ac
                  (uniLSjustjust s1 .(varS n) lps
                                 (.(n , s1) ,
```

```
                                  (unifySs1s2≡justjustu ,
                                   uni-accsubSLSulps)))
                (unifyLS≡noth , (ss ,
                                   (unifysss1s2 :: unifysslps)))
  | subSSus1≡subSSus2 | n , inj₂ (refl , refl , _)
  =  error∧unifies→⊥-acc (subSLS (n , s1) lps)
                         ((n , s1) :: (subSSS (n , s1) ac))
                         uni-accsubSLSulps
                         (unifyLS≡noth ,
                          (ss ,
                           aux n s1 ss lps (sym unifysss1s2)
                               unifysslps ))
```

B.1.3. Propiedades de las Variables

En esta sección demostramos el resultado de la figura 6.12, el cual determina que el conjunto de variables que ocurren en la sustitución que halla el algoritmo de unificación está incluido en de las variables que ocurren en la lista de pares que unificados. Para demostrar esta propiedad usamos una técnica similar a la usada en la subsección previa. Esto es, definimos un lema auxiliar que hace recursión en la demostración de que la lista de pares de tipos básicos es accesible.

Este lema auxiliar (figura B.4) establece que si la lista de pares de tipos básicos *lps* satisface el predicado `UniLS-Acc`, si el algoritmo de unificación resulta en una sustitución *ss*, y si las variables que ocurren en el acumulador y las variables que ocurren en *lps* están incluidas en una lista l cualesquiera, entonces las variables que ocurren en la sustitución *ss* están también incluidas en la lista l. Cuando llamamos este lema auxiliar pasamos $varsSLPT\ lps$ como la lista l, entonces necesitamos pruebas de que $varsSLPT\ lps \subseteq varsSLPT\ lps$, y dado que la lista acumulador inicial es vacía $[] \subseteq varsSLPT\ lps$.

```
unifyVarsS⊆lps-acc : (ss ac : List (ℕ × Sort))
                     (lps : List (Sort ×  Sort))
                     (lps-acc : UniLS-Acc lps)(l : List ℕ) →
                     unifyLS-ac lps lps-acc ac ≡ just ss →
                     varsSS ac ++ dom ac ⊆ l →
                     varsSLPT lps ⊆ l →
                     varsSS ss ++ dom ss ⊆ l
```

Figura B.4: Lema auxiliar

B.1.4. Propiedad de Idempotencia

Para demostrar que si el algoritmo de unificación resulta en una sustitución, esta es necesariamente idempotente, demostramos que satisface el predicado `IdemS`, y luego usamos el resultado anteriormente presentado que determina que si una sustitución satisface este predicado entonces es idempotente.

```
unifyVarsS⊆lps : (ss : List (ℕ × Sort))
                 (lps : List (Sort × Sort)) →
                 unifyLS lps ≡ just ss →
                 varsSS ss ++ dom ss ⊆ varsSLPT lps
unifyVarsS⊆lps ss lps p≡
  = unifyVarsS⊆lps-acc ss [] lps
                       (allListUniLS-Acc lps)
                       (varsSLPT lps)
                       p≡
                       ([]⊆l (varsSLPT lps))
                       (l⊆l (varsSLPT lps))
```

Figura B.5: Propiedad de las variables de la sustitución resultado

```
idempotentUnifyS : (lps : List (Sort × Sort))
                   (ss : List (ℕ × Sort)) →
                   unifyLS lps ≡ just ss → IdempotentS ss
idempotentUnifyS lps ss unifyLSlps≡justss
  = IdemToIdempotentS ss (idemUnifyS lps ss unifyLSlps≡justss)
```

Nuevamente para demostrar que si el algoritmo de unificación resulta en una sustitución, entonces esta satisface `IdemS`, usamos la misma técnica utilizada en las anteriores subsecciones, y hacemos recursión en la demostración de que la lista de pares de tipos básicos de entrada cumple el predicado de accesibilidad `UniLS-Acc`.

```
idemUnifyS : (lps : List (Sort × Sort))
             (ss : List (ℕ × Sort)) →
             unifyLS lps ≡ just ss → IdemS ss
idemUnifyS lps ss unifyLSlps≡justss
  = idemUnifyS-ac lps (allListUniLS-Acc lps)
                  ss [] unifyLSlps≡justss [] []idem
```

Figura B.6: La sustitución hallada por el algoritmo de unificación es idempotente

B.1.5. Unificador más General

En esta subsección demostramos que si el algoritmo de unificación resulta en una sustitución para una lista de pares de tipos básicos dada como entrada, entonces el resultado es el unificador más general de la lista de entrada. Para esta demostración usamos dos lemas auxiliares sobre el algoritmo de unificación. Ambos lemas toman entre otros parámetros la demostración de que la lista de tipos básicos cumple el predicado de accesibilidad `UniLS-Acc`.

El primer lema auxiliar establece que si una la lista de pares de tipos básicos entrada *lps* satisface el predicado de accesibilidad `UniLS-Acc`, y el algoritmo de

unificación con entrada *lps* resulta en una sustitución *ss*, y para una sustitución ss' cualesquiera si esta unifica *lps* y la sustitución acumulada *ac*, entonces ss' también unifica *ss*.

```
unifiesLpsSb→unifiesSb : (lps : List (Sort × Sort))
                (lps-ac : UniLS-Acc lps)
                (ss' ss ac : List (ℕ × Sort)) →
                unifyLS-ac lps lps-ac ac ≡ just ss →
                unifiesSS ss' ac →
                unifiesLPS ss' lps →
                unifiesSS ss' ss
```

El segundo lema auxiliar establece que si una la lista de pares de tipos básicos entrada *lps* satisface el predicado de accesibilidad `UniLS-Acc`, y el algoritmo de unificación con entrada *lps* resulta en una sustitución *ss*, y para una sustitución ss' cualesquiera si esta unifica *ss*, entonces ss' también unifica *lps* y la sustitución acumulada *ac*.

```
unifiesSb→unifiesLpsSb : (lps : List (Sort × Sort))
              (lps-ac : UniLS-Acc lps)
              (ss' ss ac : List (ℕ × Sort)) →
              unifyLS-ac lps lps-ac ac ≡ just ss →
              unifiesSS ss' ss →
              unifiesLPS ss' lps × unifiesSS ss' ac
```

Figura B.7: Segundo lema

Usamos los dos resultados anteriores en la figura B.8 para demostrar que si el algoritmo de unificación con entrada *lps* resulta en una sustitución *ss*, entonces *ss* y *lps* son equivalentes, en el sentido que ambos tienen el mismo conjunto de unificadores. Esta noción de equivalencia fue introducida en la figura 6.6.

```
≡Lps-UnifyS : (lps : List (Sort × Sort))
              (ss : List (ℕ × Sort)) →
              unifyLS lps ≡ just ss →
              lps ≡Lps-SS ss
≡Lps-UnifyS lps ss unifyLSlps≡justss ss'
  =  (λ h → (unifiesLpsSb→unifiesSb lps (allListUniLS-Acc lps)
                                    ss' ss [] unifyLSlps≡justss
                                    (proj₂ h) (proj₁ h)))
     ,
     (λ h → (unifiesSb→unifiesLpsSb lps (allListUniLS-Acc lps)
                                    ss' ss []
                                    unifyLSlps≡justss h))
```

Figura B.8: Equivalencia entre la lista de entrada y la sustitución encontrada

El próximo resultado dice que dada sustitución satisface el predicado `IdemS`, entonces esta se unifica a sí misma.

```
idemS→unifiesSS : (ss : List (ℕ × Sort)) →
                  IdemS ss → unifiesSS ss ss
```

Usamos el anterior resultado, más los lemas `unifiesSb→unifiesLpsSb` y `idemUnifyS` presentados respectivamente en la figuras B.7 y B.6, para ahora demostrar en la figura B.9 que si el algoritmo de unificación resulta en una sustitución para una lista de pares de tipos básicos dada como entrada, entonces esta sustitución unifica efectivamente la entrada.

```
unifiesSprop : (lps : List (Sort × Sort))
               (ss : List (ℕ × Sort)) →
               unifyLS lps ≡ just ss →
               unifiesLPS ss lps
unifiesSprop lps ss unifiesLSlps≡justss
  = proj₁ (unifiesSb→unifiesLpsSb
           lps (allListUniLS-Acc lps)
           ss ss [] unifiesLSlps≡justss
           (idemS→unifiesSS ss
                            (idemUnifyS lps
                                        ss
                                        unifiesLSlps≡justss)))
```

Figura B.9: Resultado del algoritmo de unificación unifica efectivamente la entrada

Resta demostrar que para toda sustitución ss' que unifica la lista de tipos básicos de entrada, el resultado del algoritmo de unificación para esta entrada es tan general como ss'. Para esto primero introducimos el siguiente lema que establece que dadas dos listas de sustituciones ss y ss' tales que ss unifica ss', entonces para cualquier tipo básico s se cumple que $s\ ss \equiv (s\ ss')\ ss$. Este resultado se demuestra por inducción en la sustitución ss' e utiliza el resultado de la figura B.1.

```
unifiesSSssss'→≡underSS : (ss ss' : List (ℕ × Sort))
                          (s : Sort) →
                          unifiesSS ss ss' →
                          (lsubS ss s) ≡ (lsubS ss (lsubS ss' s))
```

Mediante el resultado anterior podemos ahora demostrar que si una sustitución ss unifica otra sustitución ss', entonces ss' es tan general como ss. A diferencia de [20] en nuestra demostración no necesitamos exigir que ss' satisfaga el predicado `IdemS`.

```
unifies→mgu-aux : (ss ss' : List (ℕ × Sort)) →
                  unifiesSS ss ss' →
                  ss' ≤S ss
unifies→mgu-aux ss ss' unifiesSSssssss'
  = ss ,
    λ s → unifiesSSssss'→≡underSS ss ss' s
                                  unifiesSSssssss'
```

Finalmente podemos demostrar en la figura B.10 que para toda sustitución ss' que unifica la lista de tipos básicos de entrada, el resultado del algoritmo de unificación para esta entrada es tan general como ss'.

```
mguSprop-aux : (lps : List (Sort × Sort))
               (ss ss' : List (ℕ × Sort)) →
               unifyLS lps ≡ just ss →
               unifiesLPS ss' lps →
               ss ≤S ss'
mguSprop-aux lps ss ss' unifiesLSlps≡justss unifiesLPSss'lps
  = unifies→mgu-aux ss' ss
          (unifiesLpsSb→unifiesSb lps
                                  (allListUniLS-Acc lps)
                                  ss' ss [] unifiesLSlps≡justss
                                  [] unifiesLPSss'lps)
```

Figura B.10: El resultado de la unificación es el unificador más general

Reunimos ahora los principales resultados vistos en las figuras B.9 y B.10 en un único resultado que se presenta en la figura B.11. Este demuestra que el algoritmo de unificación cumple con el predicado `mguS` visto en la figura 6.8.

```
mguSprop :  (lps : List (Sort × Sort))
            (ss : List (ℕ × Sort)) →
            unifyLS lps ≡ just ss →
            mguS ss lps
mguSprop lps ss unifiesLSlps≡justss
  = unifiesSprop lps ss unifiesLSlps≡justss ,
    λ ss' unifiesLPSss'lps →
             mguSprop-aux lps ss ss'
                          unifiesLSlps≡justss
                          unifiesLPSss'lps
```

Figura B.11: Demostración de la corrección del algoritmo de unificación para tipos básicos

B.2. Unificación de Tipos de Canal

B.2.1. Terminación del Algoritmo de Unificación

Para la demostrar de la parada del algoritmo de unificación definimos el predicado de accesibilidad particular `UniLPST-Acc`. Podemos ver en la figura B.12 la definición del predicado de accesibilidad, el cual definimos basándonos en el manejo dado a las listas de pares a ser unificadas en las reglas del algoritmo de unificación presentado en la figura 6.16.

```
data UniLPST-Acc : List (SType × SType) → Set where
  uniLPST[] : UniLPST-Acc []
  uniLPST-noth : (pst : SType × SType)
    (lpst : List (SType × SType)) →
    unifyST pst ≡ nothing →
    UniLPST-Acc (pst :: lpst)
  uniLPST-justinj1 : (pst : SType × SType)
    (lpst : List (SType × SType)) →
    ∃ (λ lpst2 →
       ∃ (λ lus →
              unifyST pst ≡ just (inj₁ (lpst2 , lus)) ×
              UniLPST-Acc (lsubSLPST lus (lpst2 ++ lpst)))) →
    UniLPST-Acc (pst :: lpst)
  uniLPST-justinj2 : (pst : SType × SType)
    (lpst : List (SType × SType)) →
    ∃ (λ st → unifyST pst ≡ just (inj₂ st) ×
             UniLPST-Acc (subSTLPST st  lpst)) →
    UniLPST-Acc (pst :: lpst)
```

Figura B.12: Predicado de accesibilidad

Demostramos ahora que todas las listas de pares de tipos de canal satisfacen este predicado de accesibilidad, demostrando así que no existen secuencias infinitas decrecientes. Para esto definimos tres funciones que mapean la lista de pares de tipos de canal en ternas de naturales, para luego demostrar que estas ternas decrecen mirando las reglas del predicado `UniLPST-Acc` de derecha a izquierda. Esto significa que cada constructor inductivo de este predicado se aplica sobre un argumento con tipo indexado por una lista de pares de tipos de canal cuya terna correspondiente es menor que la terna que corresponde a la lista que indexa el tipo del elemento construido. Dado que las ternas de naturales definen un orden bien fundado comprobamos entonces que para cualquier lista de pares de tipos *lpst* existe al menos un elemento de tipo `UniLPST-Acc lpst`, cuya construcción brinda una secuencia finita decreciente de listas de pares de tipos de canal. Dado que el predicado de accesibilidad fue definido en base al algoritmo de unificación, justamente sobre esta secuencia finita de listas de pares decrecientes trabaja el algoritmo de unificación.

Una de las funciones utilizadas en el mapeo de listas de tupas a ternas de naturales es `#var-LPST`, y cuenta el número de variables de tipo de canal que aparecen en la lista de pares de tipos de canal. La función `#fun-LPST` cuenta el número de constructores de tipos de canal distintos de los constructores de variables y variables complemento, esto es `varST` y `varSTCompl` respectivamente. Finalmente la función `#ids-LPST` cuenta la cantidad de pares que son variables o variables complemento idénticas.

En la figura B.13 vemos la demostración casi completa de que todas la listas de pares de tipos son accesibles, solamente resta terminar la demostración del lema auxiliar `allNxN-Acc`. El lema auxiliar `allNxN-Acc` establece que para toda terna de naturales p y toda lista de pares de tipos de canal l, si $p \equiv \langle \#var-LPST\ l, \#fun-LPST\ l, \#ids-LPST\ l \rangle$ entonces l satisface el predicado de accesibilidad. Para demostrar este lema auxiliar utilizamos que la

recursión lexicográfica es bien fundada y la recursión completa en los naturales. Así definimos la función local `aux` que recibe como parámetros una terna de naturales $\langle vars, funs, ids\rangle$, un par ordenado $\langle p_1, p_2\rangle$, una lista de pares de tipos de canal l y una demostración que $\langle vars, funs, ids\rangle \equiv \langle \#var{-}LPST\ l, \#fun{-}LPST\ l, \#ids{-}LPST\ l\rangle$. La par argumento $\langle p_1, p_2\rangle$ nos la provee el entorno de recursión lexicográfica de la biblioteca `Induction.Lexicographic` de Agda, y nos permite realizar llamadas recursivas con listas de pares de tipos de canal l_2 cualesquiera que cumplan que $\langle \#var{-}LPST\ l_2, \#fun{-}LPST\ l_2, \#ids{-}LPST\ l_2\rangle < \langle \#var{-}LPST\ l, \#fun{-}LPST\ l, \#ids{-}LPST\ l\rangle$, donde la relación de menor $<$ corresponde a un orden lexicográfico tal cual definimos en la figura 4.1. Para el caso de ternas este orden lexicográfico se reduce entonces a exigir que se cumpla una de las tres condiciones que componen la disyunción de la definición dada en B.1.

```
allUnifyLPST-AccPred : ℕ × ℕ × ℕ → Set
allUnifyLPST-AccPred p = (l : List (SType × SType)) →
     p ≡ (#var-LPST l ,  #fun-LPST l , #ids-LPST l) →
     UniLPST-Acc l

allNxN-Acc : (p : ℕ × ℕ × ℕ) → allUnifyLPST-AccPred p
...

allUnifyLPST-acc : (l : List (SType × SType)) →
                   UniLPST-Acc l
allUnifyLPST-acc l
  = allNxN-Acc (#var-LPST l , #fun-LPST l , #ids-LPST l)
               l refl
```

Figura B.13: Demostración de que todas las listas de pares de tipos de canal satisfacen el predicado de accesibilidad

$$\langle a_1, a_2, a_3\rangle < \langle b_1, b_2, b_3\rangle \Leftrightarrow \begin{cases} (a_1 <_{\mathbb{N}} b_1) \vee \\ (a_1 = b_1 \wedge a_2 <_{\mathbb{N}} b_2) \vee \\ (a_1 = b_1 \wedge a_2 = b_2 \wedge a_3 <_{\mathbb{N}} b_3) \end{cases} \tag{B.1}$$

Vemos ahora más en detalle el argumento $\langle p_1, p_2\rangle$, su primera proyección p_1 es a su vez otra par que contiene:

1. en su primera proyección una función que dado un natural n, una demostración de que n es menor que el número de variables idénticas ids, una lista de pares de tipos de canal l_2 cuyo número de variables idénticas es n y una demostración que $\langle \#var{-}LPST\ l, \#fun{-}LPST\ l, \#ids{-}LPST\ l_2\rangle \equiv \langle \#var{-}LPST\ l_2, \#fun{-}LPST\ l_2, \#ids{-}LPST\ l_2\rangle$, devuelve que la propiedad de accesibilidad se cumple para l_2. Justamente este caso es el definido por el último término de la disyunción en la definición del orden lexicográfico para las ternas.

2. en su segunda proyección una función que dado un natural n, una demostración de que n es menor que el número de funciones $funs$, otro natural m, una lista de pares de tipos de canal l_2 cuyo número de funciones

es n y su número de variables idénticas es m y una demostración de que $\langle \#var\text{–}LPST\ l, \#fun\text{–}LPST\ l_2, \#ids\text{–}LPST\ l_2\rangle \equiv \langle \#var\text{–}LPST\ l_2, \#fun\text{–}LPST\ l_2, \#ids\text{–}LPST\ l_2\rangle$, devuelve que la propiedad de accesibilidad se cumple para l_2. Este caso se corresponde con el segundo término de la disyunción que define el orden lexicográfico para las ternas.

La segunda proyección p_2 es una función que dado un natural n, una demostración de que n es menor que el número de variables $vars$, una par de naturales $\langle m, o\rangle$, una lista de pares de tipos de canal l_2 cuyo número de funciones es m y su número de variables idénticas es o, y una demostración de que $\langle \#var\text{–}LPST\ l_2, \#fun\text{–}LPST\ l_2, \#ids\text{–}LPST\ l_2\rangle \equiv \langle \#var\text{–}LPST\ l_2, \#fun\text{–}LPST\ l_2, \#ids\text{–}LPST\ l_2\rangle$, , devuelve que la propiedad de accesibilidad se cumple para l_2. Este caso se corresponde con el primer término de la disyunción que define el orden lexicográfico para las ternas.

Presentamos la demostración del lema auxiliar `allNxN-Acc` en las figuras B.14, B.15 y B.16. En esta hacemos inducción completa en la lista de pares de tipos de canal l bajo el orden lexicográfico definido por por la terna $\langle \#var-LPST\ l, \#fun-LPST\ l, \#ids\text{–}LPST\ l\rangle$. Cuando la lista l es vacía la demostración es directa utilizando el constructor `uniLPST[]` del predicado de accesibilidad `UniLPST-Acc`, este constructor establece justamente que la lista vacía es accesible.

Cuando la lista tiene al menos una par pst, siendo así de la forma $pst :: lpst$, si la par pst está formada por dos variables idénticas, o sea es de la forma $\langle varST\ n, varST\ n\rangle$ o de la forma $\langle varSTCompl\ n, varSTCompl\ n\rangle$, vemos en la figura 6.15 que la unificación de la par pst va a dar `just (inj`$_1$ `([] , [])`, o sea una sustitución de tipos básicos y de canal vacías. Este resultado se demuestra mediante el lema auxiliar `#ids1-unifyST`. Vemos ahora en la definición del predicado de accesibilidad `UniLPST-Acc` que corresponde aplicar el constructor `uniLPST-justinj1` ya que la unificación de la par pst es de la forma antes mencionada, este constructor pide que se demuestre la accesibilidad de la lista de pares de tipos $lsubSLPST\ []\ ([]\ ++\ lpst)$ que reduce a la expresión $lsubSLPST\ []\ lpst$ por definición del operador de concatenación de listas $++$. Entonces si la variable con nombre n pertenece a la lista $lpst$, al quitar esta par de la lista y aplicar la sustitución vacía de tipos básicos va a decrecer el número de pares que son variables idénticas, este resultado lo establece el lema auxiliar $\leq$`-#ids`. Mientras que el lema `#fun-var-`$\equiv$ determina que número de variables y funciones se va a mantener constante. En estas condiciones podemos utilizar la primera proyección del argumento p_1, que corresponde a utilizar una hipótesis de la inducción completa cuando la lista es lexicográficamente menor debido al último término de la disyunción en la definición del orden lexicográfico para ternas, obteniendo así que la lista $lsubSLPST\ []\ lpst$ es accesible. Para finalmente aplicar el constructor `uniLPST-justinj1` y así obtener la demostración de que la lista original es accesible.

Cuando la variable con nombre n no pertenece a la lista $lpst$, entonces el número de variables también va a decrecer, esto se demuestra en el lema $\leq$`-#var`. Podemos entonces utilizar el argumento p_2, que corresponde a utilizar una hipótesis de la inducción completa cuando la lista es lexicográficamente menor debido al primer término de la disyunción que define el orden lexicográfico, para obtener utilizando nuevamente el constructor `uniLPST-justinj1` que la lista original es accesible. Esta primera parte de la demostración puede observarse en la figura B.14 para el caso de que la lista l sea vacía y el caso de no

serlo y que la primera par sea un par de variables idénticas sean de la forma $\langle varST\ n, varST\ n\rangle$. En la figura B.15 tenemos la demostración análoga para cuando las variables idénticas son de la forma $\langle varSTCompl\ n, varSTCompl\ n\rangle$.

```
allNxN-Acc : (p : ℕ × ℕ × ℕ) → allUnifyLPST-AccPred p
allNxN-Acc p = build ([_⊗_] <-rec-builder
                           ([_⊗_] <-rec-builder <-rec-builder))
                     allUnifyLPST-AccPred aux p            where
    aux : (p : ℕ × ℕ × ℕ) →
          (<-Rec ⊗ (<-Rec ⊗ <-Rec)) allUnifyLPST-AccPred p →
          allUnifyLPST-AccPred p
    aux .(0 , 0 , 0) rec [] refl = uniLPST[]
    aux ._ rec (pst ∷ lpst) refl with #ids pst ≟ 1
    ... | yes #idspst≡1 with #ids1 pst #idspst≡1
    aux ._ rec (.(varST n , varST n) ∷ lpst) refl
        | yes #idspst≡1 | n , inj₁ refl
        with Any.any (_≟_ n) (var-LPST lpst)
    ... | yes n∈varslpst
      = uniLPST-justinj1 (varST n , varST n)  lpst
          ([] , [] ,
           #ids1-unifyST (varST n , varST n) #idspst≡1 ,
           (proj₁ (proj₁ rec)) (#ids-LPST (lsubSLPST [] lpst))
                               (≤-#ids (varST n , varST n) lpst
                                       #idspst≡1)
                               (lsubSLPST [] lpst)
                               (#fun-var-≡ n lpst n∈varslpst))
    ... | no n∉varslpst
      = uniLPST-justinj1 (varST n , varST n)  lpst
          ([] , [] ,
           #ids1-unifyST (varST n , varST n) #idspst≡1 ,
           (proj₂  rec) (#var-LPST (lsubSLPST [] lpst))
                        (≤-#var n lpst n∉varslpst)
                        (#fun-LPST (lsubSLPST [] lpst) ,
                         #ids-LPST (lsubSLPST [] lpst)  )
                        (lsubSLPST [] lpst) refl)
```

Figura B.14: Demostración predicado auxiliar `allNxN-Acc` parte 1

Cuando la par *pst* no está formada por dos variables idénticas inspeccionamos el resultado de unificar la par *pst*, si el resultado es `nothing` utilizamos el constructor `uniLPST-noth pst` que establece que la lista es accesible si el resultado de unificar la primera par es `nothing`. Si en cambio el resultado es de la forma `just (inj₁ (lpst2 , lus))` entonces por el lema auxiliar `p1` sabemos que el número de funciones decrece en la lista $lsubSLPST\ lus$ ($lpst2$ ++ $lpst$). Mediante el lema `p2` sabemos que el número de variables permanece constante cuando no estamos en el caso que la par *pst* se corresponda a variables idénticas. Estamos entonces en condiciones de utilizar la segunda proyección del parámetro p_1, que corresponde a usar una hipótesis de la inducción completa cuando la lista es lexicográficamente menor debido al segundo término de la disyunción que define el orden lexicográfico, para así utilizando el constructor `uniLPST-justinj1` obtener una demostración de que la lista original es accesible.

```
aux ._ rec (.(varSTCompl n , varSTCompl n) :: lpst) refl
    | yes #idspst≡1 | n , inj₂ refl
    with Any.any (_≟_ n) (var-LPST lpst)
... | yes n∈varslpst
  = uniLPST-justinj1 (varSTCompl n , varSTCompl n) lpst
      ([] , [] ,
       #ids1-unifyST (varSTCompl n , varSTCompl n)
                     #idspst≡1 ,
       (proj₁ (proj₁ rec)) (#ids-LPST (lsubSLPST [] lpst))
                           (≤-#ids (varSTCompl n ,
                                    varSTCompl n  )
                                   lpst #idspst≡1)
                           (lsubSLPST [] lpst)
                           (#fun-var-≡ n lpst n∈varslpst))
... | no n∉varslpst
  = uniLPST-justinj1 (varSTCompl n , varSTCompl n) lpst
      ([] , [] ,
       #ids1-unifyST (varSTCompl n , varSTCompl n)
                     #idspst≡1 ,
       (proj₂  rec) (#var-LPST (lsubSLPST [] lpst))
                    (≤-#var-compl n lpst n∉varslpst)
                    (#fun-LPST (lsubSLPST [] lpst) ,
                     #ids-LPST (lsubSLPST [] lpst)  )
                    (lsubSLPST [] lpst) refl)
```

Figura B.15: Demostración predicado auxiliar `allNxN-Acc` parte 2

Finalmente queda discutir el último caso que es cuando el resultado de la unificación de la par *lpst* es `just (inj₂ st)`, en este caso analizando el predicado de accesibilidad vemos que aplica el constructor `uniLPST-justinj2`, el cual pide una demostración de que la lista de pares de tipos de canal $subSTLPST\ st\ lpst$ es accesible. Por otra parte el lema `p3` dice que el número de variables de $subSTLPST\ st\ lpst$ decrece con respecto a la lista original $pst :: lpst$, podemos entonces utilizar el argumento p_2 que corresponde a utilizar una hipótesis de la inducción completa cuando la lista es lexicográficamente menor debido al primer término de la disyunción que define el orden lexicográfico. Utilizamos entonces p_2 pasándole entre sus argumentos el lema `p3` para obtener la demostración de que $subSTLPST\ st\ lpst$ es accesible. Mediante la anterior demostración podemos ahora utilizar el constructor `uniLPST-justinj2` para así finalmente obtener la demostración de que la lista original es accesible. Estos dos últimos casos de la demostración pueden verse en la figura B.16.

Teniendo ahora la demostración completa del lema `allUnifyLPST-acc` que determina que todas las listas de pares de tipos de canal son accesibles, o sea satisfacen el predicado `UniLPST-Acc` definido en la figura 6.17, podemos ahora redefinir el algoritmo de unificación de listas de pares de tipos de canal `unifyLPST`. Utilizamos para esto la función auxiliar `unifyLPST-ac`, esta función recibe como argumento adicional una demostración de que la lista argumento es accesible para así hacer recursión primitiva sobre este parámetro extra. Vemos la codificación en la figura B.17. Verificamos nuevamente al igual que en la unificación

```
aux ._ rec (pst ∷ lpst) refl | no  #idspst≢1
  with inspect (unifyST pst)
... | nothing with-≡ p≡ = uniLPST-noth pst lpst (sym p≡)
... | just (inj₁ (lpst2 , lus)) with-≡ p≡
  = uniLPST-justinj1 pst lpst
      (lpst2 , lus , sym p≡ ,
       (proj₂ (proj₁ rec)) (#fun-LPST
                              (lsubSLPST lus
                                 (lpst2 ++ lpst)))
                           (p1 pst lpst lpst2 lus
                               (sym p≡) #idspst≢1)
                           (#ids-LPST
                              (lsubSLPST lus
                                          (lpst2 ++ lpst)))
                           (lsubSLPST lus (lpst2 ++ lpst))
                           (p2 pst lpst lpst2 lus
                               (sym p≡) #idspst≢1))
... | just (inj₂ st) with-≡ p≡
  = uniLPST-justinj2 pst lpst
      (st , (sym p≡ ,
       (proj₂ rec) (#var-LPST (subSTLPST st  lpst))
                   (p3 pst lpst st (sym p≡))
                   (#fun-LPST (subSTLPST st  lpst) ,
                    #ids-LPST (subSTLPST st  lpst)  )
                   (subSTLPST st  lpst) refl))
```

Figura B.16: Demostración predicado auxiliar `allNxN-Acc` parte 3

de listas de pares de tipos básicos que esta forma de introducir la recursión bien fundada de forma de garantizar la terminación no altera la estructura básica de la definición tentativa dada al comienzo de esta sección.

```
unifyLPST-ac : (l : List (SType × SType)) → UniLPST-Acc l →
                List (ℕ × Sort) → List (ℕ × SType) →
                Maybe (List (ℕ × Sort) × List (ℕ × SType))
unifyLPST-ac .[] uniLPST[] as at = just (as , at)
unifyLPST-ac .(pst :: lpst)
  (uniLPST-noth pst lpst punifSTpst≡nothing)
  as at = nothing
unifyLPST-ac .(pst :: lpst)
  (uniLPST-justinj1 pst lpst
                    (lpst2 , lus ,
                     (punifSTpst≡inj1lpst2lus ,
                      uniLSTAcclpst            )))
  as at = unifyLPST-ac (lsubSLPST lus (lpst2 ++ lpst))
                       uniLSTAcclpst
                       (lus ++ (lsubSSS lus as))
                       (lsubSLSST lus at)
unifyLPST-ac .(pst :: lpst)
  (uniLPST-justinj2 pst lpst
                    (st , (punifSTpst≡inj2st , uniLSTAcclpst)))
  as at = unifyLPST-ac (subSTLPST st lpst) uniLSTAcclpst as
                       (st :: (subSTLST st at))

unifyLPST : List (SType × SType) →
            Maybe (List (ℕ × Sort) × List (ℕ × SType))
unifyLPST ts = unifyLPST-ac ts (allUnifyLPST-acc ts) [] []
```

Figura B.17: Algoritmo de unificación de listas de pares de tipos de canal

Apéndice C

Tipado del Factorial

En la siguiente página presentamos el tipado completo del proceso que devuelve el factorial de un número enviado presentado en la sección 2.4. Observar la utilidad de la primitiva (νf) para que en (5) cuando aplicamos la regla [Nres] aparezca f en Γ'', y así Fact$[f]$ sea tipable. Recordamos en la siguiente figura la definición del proceso factorial.

$$
\begin{array}{l}
\texttt{def}\ \mathrm{Fact}(f) = \quad \texttt{accept}\ f(k)\ \texttt{in}\ k?(x)\ \texttt{in} \\
\qquad\qquad\qquad\quad \texttt{if}\ x = 1\ \texttt{then}\ k![1]; \texttt{inact} \\
\qquad\qquad\qquad\quad \texttt{else}\ (\nu b)(\mathrm{Fact}[b] \mid \texttt{request}\ b(k')\ \texttt{in}\ k'![x-1]; k'?(y)\ \texttt{in}\ k![x*y]; \texttt{inact}) \\
\texttt{in}\ (\nu f)(\mathrm{Fact}[f] \mid \texttt{request}\ f(k)\ \texttt{in}\ k![3]; k?(x)\ \texttt{in}\ \texttt{inact})
\end{array}
$$

$$\dfrac{[\textsc{Sum}]\ \dfrac{[\textsc{NameI}]\ \dfrac{}{\Gamma \cdot y : \mathrm{nat} \vdash x \triangleright \mathrm{nat}} \qquad [\textsc{NameI}]\ \dfrac{}{\Gamma \cdot y : \mathrm{nat} \vdash y \triangleright \mathrm{nat}}}{\Gamma \cdot y : \mathrm{nat} \vdash x * y \triangleright \mathrm{nat}}}{(2)}$$

$$\dfrac{\dfrac{[\textsc{Nat}]\ \dfrac{}{\{x : \mathrm{nat}\} \vdash 1 \triangleright \mathrm{nat}} \quad [\textsc{Sum}] \quad [\textsc{NameI}]\ \dfrac{}{\{x : \mathrm{nat}\} \vdash x \triangleright \mathrm{nat}}}{\{x : \mathrm{nat}\} \vdash x - 1 \triangleright \mathrm{nat}}\ [\textsc{Send}] \qquad \dfrac{[\textsc{Sum}]\ \dfrac{(2)}{\Gamma \cdot y : \mathrm{nat} \vdash x * y \triangleright \mathrm{nat}} \quad [\textsc{Send}] \quad [\textsc{Inact}]\ \dfrac{\{k' : \mathtt{end}, k : \mathtt{end}\}\ \text{es Completa}}{\Theta; \Gamma \cdot y : \mathrm{nat} \vdash \mathtt{inact} \triangleright \{k' : \mathtt{end}, k : \mathtt{end}\}}}{[\textsc{Rcv}]\ \dfrac{\Theta; \Gamma \cdot y : \mathrm{nat} \vdash k![x * y]; \mathtt{inact} \triangleright \{k' : \mathtt{end}, k :![\mathrm{nat}]; \mathtt{end}\}}{\Theta; \Gamma \vdash k'?(y) \cdots \triangleright \{k' :?[\mathrm{nat}]; \mathtt{end}, k :![\mathrm{nat}]; \mathtt{end}\}}}}{\Theta; \Gamma \vdash k'![x - 1] \cdots \triangleright \{k' :![\mathrm{nat}]; ?[\mathrm{nat}]; \mathtt{end}, k :![\mathrm{nat}]; \mathtt{end}\}}\ (1)$$

$$\dfrac{[\textsc{Var}]\ \dfrac{\emptyset\ \text{es Completa} \qquad [\textsc{NameI}]\ \dfrac{}{\Gamma \vdash b \triangleright \sigma(?[\mathrm{nat}]; ![\mathrm{nat}]; \mathtt{end})}}{[\textsc{Conc}]\ \dfrac{}{\Theta; \Gamma \vdash \mathrm{Fact}[b] \triangleright \emptyset}} \qquad \dfrac{[\textsc{NameI}]\ \dfrac{}{\Gamma \vdash b \triangleright \sigma(?[\mathrm{nat}]; ![\mathrm{nat}]; \mathtt{end})} \quad [\textsc{Req}] \quad \dfrac{(1)}{\Theta; \Gamma \vdash k'![x - 1] \cdots \triangleright \{k' :![\mathrm{nat}]; ?[\mathrm{nat}]; \mathtt{end}, k :![\mathrm{nat}]; \mathtt{end}\}}}{\Theta; \Gamma \vdash \mathtt{request}\ \ b(k') \cdots \triangleright \{k :![\mathrm{nat}]; \mathtt{end}\}}}{[\textsc{Nres}]\ \dfrac{\Theta; \Gamma \vdash \mathrm{Fact}[b] \mid \mathtt{request}\ \ b(k') \cdots \triangleright \{k :![\mathrm{nat}]; \mathtt{end}\}}{\Theta; \Gamma' \cdot x : \mathrm{nat} \vdash (\nu b)(\mathrm{Fact}[b] \mid \mathtt{request}\ \ \ldots) \triangleright \{k :![\mathrm{nat}]; \mathtt{end}\}}\ (3)}$$

$$[\textsc{If}]\ \dfrac{\Gamma' \cdot x : \mathrm{nat} \vdash x = 1 : \mathrm{bool} \qquad \dfrac{[\textsc{Nat}]\ \dfrac{}{\Gamma' \cdot x : \mathrm{nat} \vdash 1 \triangleright \mathrm{nat}} \quad [\textsc{Send}] \quad [\textsc{Inact}]\ \dfrac{\{k : \mathtt{end}\}\ \text{es Completa}}{\Theta; \Gamma' \cdot x : \mathrm{nat} \vdash \mathtt{inact} \triangleright \{k : \mathtt{end}\}}}{\Theta; \Gamma' \cdot x : \mathrm{nat} \vdash k![1]; \mathtt{inact} \triangleright\ \{k :![\mathrm{nat}]; \mathtt{end}\}} \qquad [\textsc{Nres}]\ \dfrac{(3)}{\Theta; \Gamma' \cdot x : \mathrm{nat} \vdash (\mathrm{vb})(\mathrm{Fact}[b] \mid \mathtt{request}\ \ \ldots) \triangleright \{k :![\mathrm{nat}]; \mathtt{end}\}}}{\Theta; \Gamma' \cdot x : \mathrm{nat} \vdash \mathtt{if}\ \ x = 1 \cdots \triangleright \{k :![\mathrm{nat}]; \mathtt{end}\}}\ (4)$$

$\Gamma = \{f : \sigma(?[\mathrm{nat}]; ![\mathrm{nat}]; \mathtt{end}), x : \mathrm{nat}, b : \sigma(?[\mathrm{nat}]; ![\mathrm{nat}]; \mathtt{end})\} = \Gamma' \cdot x : \mathrm{nat} \cdot b : \sigma(?[\mathrm{nat}]; ![\mathrm{nat}]; \mathtt{end})$
$\Gamma' = \{f : \sigma(?[\mathrm{nat}]; ![\mathrm{nat}]; \mathtt{end})\}$
$\Gamma'' = \{f : \sigma(?[\mathrm{nat}]; ![\mathrm{nat}]; \mathtt{end})\}$
$\Theta = \{\mathrm{Fact} : \sigma(?[\mathrm{nat}]; ![\mathrm{nat}]; \mathtt{end})\}$

$$\dfrac{\dfrac{}{\Gamma'' \vdash 3 \triangleright \text{nat}}\,[\textsc{Nat}] \qquad \dfrac{\dfrac{\{k : \texttt{end}\} \text{ es Completa}}{\Theta; \Gamma'' \cdot \{x : \text{nat}\} \vdash \texttt{inact} \triangleright \{k : \texttt{end}\}}\,[\textsc{Inact}]}{\Theta; \Gamma'' \vdash k?(x)\ \texttt{in inact} \triangleright \{k :?[\text{nat}]; \texttt{end}\}}\,[\textsc{Rcv}]}{\Theta; \Gamma'' \vdash k![3]; k?(x)\ \texttt{in inact} \triangleright \{k :![\text{nat}]; ?[\text{nat}]; \texttt{end}\}}\,[\textsc{Send}] \quad (6)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \text{ es Completa}}\,[\textsc{Var}] \qquad \dfrac{}{\Gamma'' \vdash f \triangleright \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end})}\,[\textsc{NameI}]}{\Theta; \Gamma'' \vdash \text{Fact}[f] \triangleright \emptyset}\,[\textsc{Conc}] \qquad \dfrac{}{\Gamma'' \vdash f \triangleright \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end})}\,[\textsc{NameI}]}{\Theta; \Gamma'' \vdash \texttt{request}\ f(k)\ \texttt{in}\ k![3]; k?(x)\ \texttt{in inact} \triangleright \emptyset}\,[\textsc{Req}] \qquad \dfrac{(6)}{\Theta; \Gamma'' \vdash k![3]; k?(x)\ \texttt{in inact} \triangleright \{k :![\text{nat}]; ?[\text{nat}]; \texttt{end}\}}\,[\textsc{Send}]}{\Theta; \Gamma'' \vdash \text{Fact}[f] \mid \texttt{request}\ f(k)\ \texttt{in}\ k![3]; k?(x)\ \texttt{in inact} \triangleright \emptyset}\,[\textsc{Conc}]}{\Theta; \emptyset \vdash (\nu f)(\text{Fact}[f] \mid \texttt{request}\ f(k)\ \texttt{in}\ k![3]; k?(x)\ \texttt{in inact}) \triangleright \emptyset}\,[\textsc{NRes}] \quad (5)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash f \triangleright \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end})}\,[\textsc{NameI}] \qquad \dfrac{\dfrac{(4)}{\Theta; \Gamma' \cdot x : \text{nat} \vdash \texttt{if}\ \ x = 1 \cdots \triangleright \{k :![\text{nat}]; \texttt{end}\}}}{\Theta; \Gamma' \vdash k?(x)\ \texttt{in} \cdots \triangleright \{k :?[\text{nat}]; ![\text{nat}]; \texttt{end}\}}\,[\textsc{Rcv}]}{\Theta; \Gamma' \vdash \texttt{accept} f(k) \texttt{in} \cdots \triangleright \emptyset}\,[\textsc{Acc}] \qquad \dfrac{(5)}{\Theta; \emptyset \vdash (\nu f)(\text{Fact}[f] \mid \texttt{request}\ f(k)\ \texttt{in}\ k![3]; k?(x)\ \texttt{in inact}) \triangleright \emptyset}\,[\textsc{Nres}]}{\emptyset; \emptyset \vdash \texttt{def}\ \text{Fact}(f) = \texttt{accept} \ldots \texttt{in}\ (\nu f)(\text{Fact}[f] \mid \texttt{request}\ f(k)\ \texttt{in}\ k![3]; k?(x)\ \texttt{in inact}) \triangleright \emptyset}\,[\textsc{Def}]$$

$\Gamma = \{f : \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end}), x : \text{nat}, b : \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end})\} = \Gamma' \cdot x : \text{nat} \cdot b : \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end})$
$\Gamma' = \{f : \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end})\}$
$\Gamma'' = \{f : \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end})\}$
$\Theta = \{\text{Fact} : \sigma(?[\text{nat}]; ![\text{nat}]; \texttt{end})\}$