UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Análisis estático de tipos para lenguajes de tipado dinámico

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Gorka Suárez García

Directores

Francisco Javier López Fraguas
Manuel Montenegro Montes

Madrid

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Análisis estático de tipos para lenguajes de tipado dinámico

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Gorka Suárez García

DIRECTORES

Francisco Javier López Fraguas
Manuel Montenegro Montes

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA
Departamento de Sistemas Informáticos y Computación

Análisis estático de tipos para lenguajes de tipado dinámico

TESIS DOCTORAL

Presentada por:
Gorka Suárez García

Bajo la dirección de los Doctores:
Francisco Javier López Fraguas
Manuel Montenegro Montes

Madrid, 4 de marzo de 2022

A mis hermanas gemelas, Angara y Arantxa.

Índice general

Agradecimientos

En primer lugar, quiero agradecer la labor de mis directores Francisco J. López Fraguas y Manuel Montenegro Montes, sobre todo su apoyo y dedicación durante todos estos años de investigación, pues no hay palabras para expresar con justicia todos el apoyo que me han dado bajo su dirección. Sin su inestimable ayuda, este trabajo no hubiera sido posible.

En segundo lugar, quiero agradecer a mis amigos Alan Somoza, Bruno Avendaño, César López, Lourdes García, Marco Lerma y Raquel García, por todo el amor y las alegrías que me brindan en este curioso universo que compartimos todos juntos. Además quiero agradecer en especial a Joaquín Gayoso, que—además de un buen amigo—ha sido durante buena parte de estos interesantes años de tesis el principal pilar que me mantenía cuerdo. A todos ellos quiero dar las gracias desde lo más profundo de mi corazón, pues son mi segunda familia en este mundo.

En tercer lugar, pero no menos importante por ello, quiero dar las gracias a mi familia: Gorka, Gema y Thaisa. No sería la persona que soy hoy día sin todo lo que ellos me han aportado a mi vida. Sobre todo agradecer a mi padre que me enseñara a amar las matemáticas y la ciencia.

En cuarto lugar, agradecer a los hermanos Iván José y Víctor Manuel Pérez Colado todos estos años de apoyo durante el máster y el doctorado como compañeros de fatigas, sobre todo recordando los innumerables buenos momentos que hemos tenido en tiempos tan difíciles. Así como agradecer también a los compañeros del Aula 16 por todos los buenos momentos que hemos vivido a lo largo de estos últimos años.

En quinto lugar, quiero agradecer a Josep Silva Galiana y Lars-Åke Fredlund por sus excelentes revisiones, pues sus críticas constructivas han contribuido a poder mejorar esta tesis.

También quiero agradecer a la música todo lo que me ha aportado en esta vida, ella es el mayor amor de mi vida. Miles de palabras no bastarían para empezar a expresar cuán importante es para mí el mundo de la música, pero quiero—aunque sea brevemente—destacar mi amor incondicional a la obra musical de Yoko Kanno, Nobuo Uematsu, Joe Hisaishi, John Williams, Björk, King Crimson, Vangelis y Pyotr Ilyich Tchaikovsky. No obstante, muchos son los nombres que faltan ahí. Termino esta ronda de agradecimientos musicales, agradeciendo a la música City Pop y la música Techno las fuerzas que me han dado para escribir la tesis, sobre todo en este último año.

Finalmente, agradecer a la Universidad Complutense de Madrid toda su ayuda, en especial a Narciso Martí Oliet por su gran paciencia y ayuda durante el doctorado, así como agradecer a los proyectos de investigación CAVI-ART (TIN2013-44742-C4-3-R) y CAVI-ART-2 (TIN2017-86217-R), financiados por el MINECO, N-GREENS Software-CM (S2013/ICE-2731), financiado por la Comunidad de Madrid, y BLOQUES-CM (S2018/TCS-4339), financiado por la Comunidad de Madrid y los fondos EIE de la Unión Europea.

Gorka Suárez García
Madrid, 4 de marzo de 2022

Resumen

Los sistemas de tipos son una herramienta formal que permiten clasificar las distintas construcciones de un lenguaje de programación (valores, expresiones, etc.) en distintas categorías, llamadas *tipos*. Con ello se pretende detectar posibles inconsistencias entre las distintas variables y expresiones de un programa. Existen distintos enfoques en la aplicación de un sistema de tipos. Por un lado tenemos lenguajes como *C++* con *tipado estático*, donde la comprobación de tipos se realiza en tiempo de compilación, y por el otro tenemos los lenguajes con *tipado dinámico* como *Erlang*, donde la comprobación de tipos se realiza en tiempo de ejecución. Como consecuencia de la naturaleza de los lenguajes de tipado dinámico, la detección de errores en los programas se realiza durante las fases de prueba y depuración. No obstante, existen herramientas que permiten la detección automática de errores de tipo en un programa sin necesidad de ejecutarlo. Estas herramientas aplican metodologías propias de análisis estático de tipos a lenguajes de tipado dinámico.

Erlang es un lenguaje de programación funcional concurrente de tipado dinámico, que dispone de una herramienta propia (*Dialyzer*) para analizar programas escritos en *Erlang* y poder detectar, en tiempo de compilación, discrepancias en el uso de los tipos. Esta herramienta se basa en la noción de *success types*, que son una sobreaproximación a la semántica de las expresiones del lenguaje, de modo que la evaluación de una expresión mal tipada necesariamente fallará en tiempo de ejecución. *Dialyzer* permite a un programador especificar tipos polimórficos (que capturan relaciones entre las entradas y la salida de una función) y tipos sobrecargados (que permiten reflejar las distintas ramas de ejecución de una función). No obstante, la información relativa al polimorfismo y sobrecarga solamente sirve a efectos de documentación de los programas, y *Dialyzer* descarta esta información a la hora de realizar su comprobación de tipos. Esto supone una pérdida de precisión en el análisis, lo cual se traduce en una menor capacidad para detectar errores en un programa.

En esta tesis presentamos un *sistema de tipos polimórficos sobrecargados*, basado en la noción de *success types*, con el que poder analizar programas escritos en un subconjunto del lenguaje *Erlang*. Gracias al polimorfismo podemos conectar con mayor precisión las entradas y salidas de un tipo funcional a la hora de sobreaproximar la semántica de las funciones. Además, al incorporar sobrecarga, podemos capturar de un modo más preciso la distinción entre las ramas de ejecución que pueden darse al evaluar una expresión. Al disponer de ambas características, conseguimos realizar un análisis de tipos más preciso que el realizado por el sistema de *success types* monomórfico de *Dialyzer*. Además

de describir un sistema para derivar *success types* polimórficos sobrecargados, también presentamos un algoritmo de inferencia acompañado de una herramienta que infiere y muestra los *success types* de un conjunto de funciones dado. En particular, hemos utilizado esta herramienta para inferir los tipos de algunas funciones habituales en programación funcional.

En esta tesis no solo presentamos un sistema de *success types* polimórficos sobrecargados con el que derivar e inferir tipos, sino que también aportamos resultados de corrección que demuestran que los tipos obtenidos son *success types* de las expresiones analizadas. Esto es importante, ya que *Dialyzer* carecía de un marco teórico sólido sobre el que poder razonar a la hora de incorporar tipos polimórficos. De hecho, esta carencia ha sido la que ha motivado gran parte de nuestra investigación, que procura ofrecer unos fundamentos teóricos para futuros sistemas de tipos basados en *success types*.

Abstract

Type systems are a formal tool that allows the different constructions of a programming language (values, expressions, etc.) to be classified into different categories, called *types*. Type systems can detect possible inconsistencies between the different variables and expressions of a program. There are different approaches to applying a type system. On the one hand we have languages like *C++* with *static typing*, where type checking is done at compile time, and on the other hand we have languages with *dynamic typing* like *Erlang*, where type checking is done at run time. Due to the nature of dynamically typed languages, error detection in programs is done during the testing and debugging phases. However, there are tools that allow the automatic detection of type errors in a program without having to execute it. These tools apply their own static type analysis methodologies to dynamic typing languages.

Erlang is a dynamically-typed concurrent functional programming language, which has its own tool (*Dialyzer*) to analyse programs written in *Erlang* and to detect, at compile time, discrepancies in the use of types. This tool is based on the notion of *success types*, which are over-approximations of the semantics of language expressions, so that the evaluation of an ill-typed expression will necessarily fail at runtime. *Dialyzer* allows a programmer to specify polymorphic types (which capture connections between the inputs and the output of a function) and overloaded types (which reflect the different execution branches of a function). However, the information regarding polymorphism and overloading is only for program documentation purposes, and *Dialyzer* discards this information when doing its type checking. This implies a loss of precision in the analysis, which turns into a reduced ability to detect errors in a program.

In this thesis we present a *system of overloaded polymorphic types*, based on the notion of *success types*, with which to analyse programs written in a subset of the *Erlang* language. Thanks to polymorphism, we can more precisely connect the inputs and outputs of a functional type when it comes to over-approximating the semantics of the functions. Also, by incorporating type overload, we can more precisely capture the distinction between execution branches that can be taken when evaluating an expression. By having both characteristics, we are able to perform a more precise type analysis than that carried out by the monomorphic *success types* system of *Dialyzer*. In addition to describing a system for deriving overloaded polymorphic *success types*, we also introduce an inference algorithm

accompanied by a tool that infers and displays the *success types* of a given set of functions. In particular, we have used this tool to infer the types of some common functions in functional programming.

In this thesis we not only introduce a system of overloaded polymorphic *success types* with which to derive and infer types, but we also provide correctness results that show that the types obtained are *success types* of the analysed expressions. This is important, since *Dialyzer* lacked a solid theoretical framework on which to reason when incorporating polymorphic types. In fact, it is this deficiency that has motivated much of our research, which seeks to provide a theoretical foundation for future type systems based on *success types*.

Capítulo 1

Introducción

En este capítulo vamos a introducir al lector en el problema que afronta nuestra investigación para *analizar tipos* en lenguajes de programación de *tipado dinámico*, en concreto en el lenguaje *Erlang*, sobre el que se ha centrado nuestro trabajo. En la sección 1.1 vamos a explicar qué son los sistemas de tipos. En la sección 1.2 hablaremos sobre el *polimorfismo*, una característica muy relevante presente en muchos sistemas de tipos. En la sección 1.3 nos vamos a centrar en una clase de sistemas de tipos concreta denominada *success types*. En la sección 1.4 explicaremos en más detalle la herramienta *Dialyzer* [99], que analiza si existen errores con los tipos en una aplicación *Erlang* mediante el uso de *success types*, propuestos en [61, 62] como una forma adecuada de aproximarse al tipado de lenguajes de tipado dinámico, de naturaleza habitualmente más flexible que la que se en cuentra en lenguajes de tipado estático. En la sección 1.5 veremos algunas de las alternativas disponibles que existen para analizar y comprobar tipos en lenguajes de programación. Por último en la sección 1.6 hablaremos de los objetivos y contribuciones de nuestra investigación.

1.1. Sistemas de tipos

En [91] se define un *sistema de tipos*, en un lenguaje de programación, como un *sistema lógico* que comprende un conjunto de reglas que asignan una propiedad llamada *tipo* a diversas construcciones de un programa, tales como variables, expresiones, funciones o módulos. Por ejemplo, una variable de programa es una forma de asignar un nombre a un valor, que puede modificarse o no durante el transcurso de la ejecución, y dicho valor ha de pertenecer al conjunto de valores denotados por un tipo. Con los *tipos* formalizamos las categorías implícitas que se usan en los programas para implementar tipos de datos algebraicos, estructuras de datos u otros componentes (tales como números, cadenas de texto, arrays, funciones, etcétera). El principal propósito que se busca con un sistema de tipos [14], es reducir al máximo posible los errores que hay en un programa definiendo interfaces entre las diferentes partes del mismo programa, de modo que luego se pueda comprobar que dichas

partes han sido conectadas de forma consistente. Estas comprobaciones pueden realizarse durante la compilación (estáticas), en tiempo de ejecución (dinámicas) o mediante una combinación de ambas. Como esta distinción entre *tipado estático* y *tipado dinámico* es importante, vamos a desarrollarla a continuación.

Los lenguajes de *tipado estático* son aquellos en los que la comprobación que realiza el sistema de tipos está implementada en el compilador, para que sea en tiempo de compilación cuando se realicen estas comprobaciones y después de compilar se presenten al usuario los errores obtenidos en caso de fallar la comprobación. Hay muchos lenguajes de tipado estático conocidos en el mundo de la programación, como es el caso de *C* [56], *C++* [112], *C#* [4], *Haskell* [48, 47, 63], *Java* [40], *Pascal* [121], *Rust* [76], *Scala* [88], etcétera. Para entenderlo mejor veamos el siguiente ejemplo escrito en lenguaje *C*:

```
void mostrar(const char * mensaje);

void prueba() {
    mostrar("Practical Forms of Type Theory");
    mostrar("Alan M. Turing");
    mostrar(1948);
}
```

Aquí tenemos una función `mostrar` que acepta como argumento valores que son cadenas de caracteres, seguida de una función `prueba` que está llamando a `mostrar` tres veces, los argumentos de las dos primeras llamadas son valores constantes de cadenas y el de la última es una constante numérica. Al compilar este código se genera un mensaje, que varía dependiendo del compilador usado, similar al siguiente:

```
error C2664: 'void mostrar(const char *)': cannot convert
             argument 1 from 'int' to 'const char *'
```

El primer mensaje indica que se ha provocado un error de compilación relativo a los tipos que estamos usando, ya que no puede convertir el argumento primero en la aplicación de `mostrar` de `int` (número entero) a `const char *` (cadena de texto constante). Esto es así porque `mostrar` solo acepta cadenas de texto como parámetro de entrada y no puede manejar un número entero. Para solucionar esto, el programador, o bien define una nueva función que haga lo mismo que `mostrar` pero recibiendo un número como parámetro, o bien define o utiliza alguna función que convierta números enteros a cadenas de texto.

Algunos lenguajes de tipado estático requieren que el programador defina correctamente los tipos y programe las conversiones entre estructuras de tipos de forma explícita en multitud de ocasiones. De lo contrario, los programas no compilarán porque podría haber discrepancias entre tipos y, en ese

caso, la comprobación del sistema de tipos implementado devolverá errores. Esto hace más seguros los programas, pero al mismo tiempo requiere un mayor esfuerzo al tener que detallar explícitamente todo comportamiento relacionado con los tipos. En ocasiones se puede llegar a inferir parte de la información de tipos, analizando las expresiones del programa, evitando así tener que indicar de forma explícita los tipos. Esto último es el caso de *Haskell*, que utiliza una extensión del sistema de tipos de *Hindley-Milner* [29], un sistema de tipos clásico para el cálculo lambda, que introduce polimorfismo paramétrico y que se utiliza sobre todo en lenguajes de programación funcional. Pero a pesar de estas facilidades, los sistema de tipos *Hindley-Milner* tienen algunos inconvenientes. Por ejemplo, tomemos el siguiente ejemplo escrito en *Haskell*:

```
head [1982, "Blade Runner", True]
```

Aquí tenemos una lista heterogénea de tipos de datos, de la que intentamos extraer la cabeza de la lista. Si intentamos compilar o interpretar esta expresión en *Haskell* obtendremos un error, porque al intentar inferir el tipo de la lista se obtienen diferentes tipos en cada elemento y el comprobador indica que se está haciendo un uso incorrecto de los tipos. Para solucionar este problema tendremos que definir un nuevo tipo en el programa:

```
data Info = I Int | S [Char] | B Bool deriving (Show)
```

Aquí estamos definiendo un tipo `Info` que tiene tres variantes: enteros, cadenas de caracteres y booleanos. Cada opción viene marcada con un constructor que da nombre a la variante que puede pertenecer al término de tipo `Info`. De esta manera podemos corregir la lista anterior de la siguiente manera:

```
head [I 1982, S "Blade Runner", B True]
```

De esta manera, obtenemos como resultado `I 1982` en vez de `1982`, por lo que para procesar el valor en sí habrá que usar el encaje de patrones para sacar la información contenida dentro del constructor `I`. Es decir, la forma de tratar esta limitación es poner los datos en cajas (*wrapper types*) y luego sacarlos de esas cajas cuando los necesitemos. A esta complicación hay que sumar el hecho de que con el tipo auxiliar `Info` no conseguimos listas heterogéneas generales. Siendo trivial este ejemplo, podría parecer que no es un problema acuciante, pero a medida que van ganando complejidad los sistemas desarrollados se termina empleando muchos recursos en definir estructuras de tipos auxiliares para adecuarse a las necesidades de los sistemas de tipado estático. Es más, muchos lenguajes han tratado de incorporar con mayor o menor éxito mecanismos que permitieran lidiar con las limitaciones del tipado estático, siendo la orientación a objetos un ejemplo paradigmático de esta situación. Con la orientación de objetos se crean jerarquías de clases, donde las clases padres tienen interfaces de uso definidas que son sobrecargadas por las clases hijas, de este modo se pueden generalizar los algoritmos. Aunque la orientación a objetos ha llevado a otros inconvenientes en el diseño de aplicaciones,

como el tener que lidiar con conceptos como la herencia múltiple, llegando al punto de cambiar el modelo de herencia por el de composición con la arquitectura de sistemas de entidades por componentes.

Frente a esta situación, en los lenguajes de *tipado dinámico*, la comprobación de tipos se realiza en tiempo de ejecución. Aunque muchos de los lenguajes que soportan este modelo son interpretados, también los hay que se compilan a lenguajes intermedios optimizados, también conocidos como *bytecode*, que luego son interpretados por una máquina virtual permitiendo su portabilidad entre sistemas. Al realizarse la comprobación de tipos en tiempo de ejecución, no es necesario hacerlo de antemano. Esa información se guarda junto al valor creado por la expresión que es asignada a la variable. Es el programa, con los mecanismos que le ofrece el intérprete o la máquina virtual, quien se encarga—durante la ejecución—de comprobar si una variable contiene un tipo determinado o no. En caso de existir un conflicto de tipos, como pasar un número a una función que debería recibir una cadena, como en el ejemplo anterior de la función `mostrar`, el sistema se encarga de lanzar un error de ejecución, que—dependiendo del lenguaje—si es una excepción podríamos capturar para procesar el fallo y continuar con la ejecución.

Entre los lenguajes de tipado dinámico más populares se encuentran *JavaScript* [11, 27], *Python* [117], *R* [75], *Erlang* [45], *LISP* [78], *Ruby* [77], *Lua* [49] o *PHP* [58] entre otros. Siendo lenguajes más flexibles en cuanto al uso de los tipos, permiten aparentemente una mayor productividad al no obligar al programador a definir tipos de datos auxiliares, permitiendo su combinación con metodologías de desarrollo ágiles en las que el desarrollo es iterativo e incremental, para sistemas que están en constante evolución (por ejemplo, sistemas web). Pero si bien estos lenguajes nos permiten tener listas de tipos heterogéneas sin que suponga un problema, en contraposición a lo que vimos en el ejemplo de *Haskell* mostrado anteriormente, igualmente requieren el paso de sacar los datos de sus cajas para inspeccionarlos y determinar su tipo o forma. Además, se le añade otro problema bastante serio, ya que la comprobación del buen uso de los tipos solo se realiza durante la ejecución, por lo que se requiere de una especial atención en la fase de pruebas y ello supone un coste importante en los proyectos, debido—en parte—al desarrollo de baterías de pruebas que sean lo más amplias posibles para detectar el mayor número de errores que existan en el sistema. En el peor de los casos, la aplicación puede llegar a ser lanzada sin que se descubra alguno de estos errores de tipos.

A pesar del peligro de desarrollar código inseguro e inestable, los lenguajes de tipado dinámico han ido ganando fuerza durante los últimos años. Incluso se ha llegado al punto en el que algunos lenguajes de tipado estático como *C#* [111] han pasado a incorporar mecanismos de tipado dinámico como parte de su entorno, para combinar las dos filosofías. Es por ello que la industria dedica esfuerzos a investigar el desarrollo de herramientas que automaticen la búsqueda de errores en los lenguajes de tipado dinámico. Uno de estos enfoques es el análisis estático de tipos, que mediante una serie de reglas de inferencia obtiene la información relativa a los tipos que tienen las variables del programa, detectando aquellas que estén provocando un conflicto en su uso y por lo tanto que pudieran conducir a un fallo durante la ejecución.

Finalmente, queremos señalar que también se pueden concebir los sistemas de tipos para comprobar otras propiedades de un programa ajenas a la clasificación de valores, como por ejemplo a la hora de verificar el uso que se hace de la memoria por parte de las variables del programa. Este es el caso del lenguaje de programación *Rust*, que dispone de un sistema de propiedad (*ownership*) de los valores por parte de las variables, donde todos los valores tienen un único propietario y el ámbito de estos valores es el ámbito de su propietario actual. Estos mecanismos de control que implementa su sistema de tipos permiten que el lenguaje sea seguro en relación a los accesos de memoria, no permitiendo punteros nulos, punteros o referencias descolgadas, así como tampoco condiciones de carrera sobre variables.

1.2. Tipos polimórficos

Como hemos señalado en el capítulo anterior, los sistemas de tipos sirven para poder analizar las expresiones que hay dentro de un programa dado, para conocer a qué tipos pertenecen estas y si se están usando correctamente. En principio, en los lenguajes estáticos las variables de programa tienen un tipo concreto, pero algunos lenguajes permiten crear funciones genéricas donde los tipos de las expresiones están parametrizados con *variables de tipo*. Un ejemplo de esto son las plantillas en *C++*, los genéricos en *C#* y *Java*, o el polimorfismo paramétrico de *Haskell*. Si quisiéramos *aplicar* una función a todos los elementos de una lista en *Haskell* tendríamos la siguiente función:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

En la primera línea tenemos el tipo de la función, en la segunda el caso base con la lista vacía y en la tercera el caso recursivo que toma la cabeza de la lista, aplica la función `f` y procesa el resto de la lista. Aquí no tenemos un tipo concreto para los elementos de la lista de entrada (por ejemplo, números enteros), sino una lista cuyos elementos pertenecen al tipo `a`, que es una variable de tipo.Una *variable de tipo* puede ser instanciada a cualquier otro tipo del sistema de tipos. Además las variables de tipo nos permiten relacionar los tipos de la entrada y los tipos de la salida de una función. En este ejemplo, tenemos dos variables de tipo que son `a` y `b`, que nos describe que el primer parámetro es una función `a -> b` y el segundo una lista de `a`, por lo que la entrada de la función pasada como parámetro está relacionada con los elementos de la lista pasada también como argumento; y además también se describe que el resultado es una lista de `b`, relacionado la salida de la función del primer parámetro de `map` con los elementos de la lista devuelta. De ese modo, si pasamos a `map` una función con el tipo `Int -> Float`, forzosamente el segundo parámetro es de tipo `[Int]` y la el tipo del resultado es `[Float]`, ya que de no serlo el compilador nos avisaría de un error de tipos. En el caso concreto de *Haskell*, cada tipo válido obtenido tras sustituir las variables de tipo por otros tipos es una instancia posible del tipo polimórfico de la función. Por ejemplo, si tenemos la función identidad en *Haskell*:

```
id :: a -> a
id x = x
```

por un lado tenemos el tipo polimórfico `a -> a` y por otro las instancias posibles, como es el caso de `Int -> Int`, `Bool -> Bool`, etcétera. Dependiendo del sistema de tipos, las variables de tipo pueden restringirse para pertenecer a una clase[1] de tipos [43] o ser subtipo de algún tipo.

Gracias a las variables de tipo se puede realizar programación genérica dentro de los lenguajes de tipado estático, evitando tener que declarar múltiples funciones con cada instancia concreta de tipos para ofrecer la sobrecarga correspondiente. Pero también sirve para que un sistema de tipos pueda hacer las comprobaciones de tipos sin tener que generar todas las instancias posibles de un tipo polimórfico como las que hemos visto para `map` o `id`, porque si no se dispone de tipos polimórficos en un sistema de tipos, la solución más común es tener un tipo que representa a todos los posibles valores del lenguaje; pero el uso de este tipo no nos permitiría, en los ejemplos anteriores, relacionar las entradas y las salidas de las funciones.

Siendo importantes las ventajas de disponer de tipos polimórficos en un sistema de tipos, la definición de su semántica no es trivial. El polimorfismo de sistemas basados en los tipos *Hindley-Milner*, como el empleado por *Haskell*, depende directamente del algoritmo de inferencia que se cimenta sobre la generación de ecuaciones de igualdad, aplicando una mayor restricción al uso de los tipos. Hay sistemas de tipos que utilizan el *subtipado de tipos*, para poder representar mejor la semántica del lenguaje que van a analizar, añadiendo una mayor complejidad a sus sistemas a la hora de resolver las ecuaciones en la inferencia. Otro componente que aporta una mayor complejidad a la semántica son las *uniones de tipos* y que además nos obliga a tomar la decisión de si ser más conservadores o no con el análisis, porque si pasamos a `map` una función $Int \rightarrow Int$ acompañada de una lista $[Int \cup Bool]$, tendremos que determinar si rechazamos o no este uso, con sus diferentes implicaciones que pueden tener sus respectivas ventajas y desventajas. Por ejemplo, si usamos un sistema basado en restricciones de igualdad de tipos, en vez de en restricciones de subtipado, tendríamos que rechazar este último ejemplo, ya que un conjunto de valores es mayor que el otro y sería incorrecto pasar un valor de tipo $Int \cup Bool$ allá donde se espera un tipo *Int*; mientras que al usar un sistema basado en subtipado podemos llegar a considerar como válido el uso al estar uno de los tipos contenido en el otro. El combinar las uniones de tipo con el subtipado conlleva importantes complejidades técnicas, como se muestra en las reglas obtenidas en nuestra investigación a la hora de derivar tipos polimórficos. Otro aspecto semántico que hay que tener en cuenta también es si permitir o no *tipos unitarios*, que son aquellos valores literales del lenguaje que podemos representarlos como un conjunto con un único valor dentro. Si tratáramos de incorporar tipos unitarios en sistemas basados en restricciones de igualdad, nos encontraríamos que no funcionan particularmente bien con este concepto, pues tomando el último ejemplo con `map`, si la función $Int \rightarrow Int$ está acompañada de la lista [1] nos encontraremos con el mismo problema que con las uniones, lo que llevaría o bien a tipar el primer parámetro como $1 \rightarrow Int$ o

[1] Las clases de tipos de *Haskell* sirven para definir una interfaz que han de cumplir los tipos que pueden reemplazar una variable de tipo cuando se selecciona una instancia.

sobreaproximar el segundo argumento como *Int*. Como consecuencia de esta problemática, los tipos unitarios funcionan mejor en sistemas con subtipado y el sistema de tipos presentado en esta tesis dispone de ellos.

1.3. Success types

La idea de *success types* fue propuesta en [61, 62] como un enfoque adecuado para realizar análisis de tipos en lenguajes de tipado dinámico sin comprometer su flexibilidad de uso. Para entender la filosofía que hay detrás de los *success types* hay que conocer primero el lenguaje *Erlang*, un lenguaje funcional concurrente y dinámicamente tipado, que motivó la necesidad de crear este sistema de tipos.

En contraposición a la disciplina de tipado estático que ofrecen los sistemas de tipos de *Hindley-Milner* [29], la naturaleza de tipado dinámico de *Erlang* hace que sea más flexible para la tarea de programar. Sin embargo, como ya hemos mencionado en la sección anterior, esta flexibilidad tiene como coste que los errores de tipo puedan quedar ocultos hasta que el programa sea ejecutado. Se han realizado múltiples intentos para diseñar herramientas que analicen los tipos en tiempo de compilación para *Erlang* [74, 116, 53]. Entre todos los intentos realizados encontramos a *Dialyzer* [60, 62, 55], que se puede utilizar para encontrar discrepancias entre tipos en los programas, y a la herramienta relacionada *TypEr* [61], que es capaz de inferir los tipos de las funciones definidas en un programa. Ambas herramientas sirven tanto para el propósito de documentar los programas, como para detectar errores de ejecución, y constituyen una solución pragmática que fue incorporada, casi desde el momento de su creación, en la distibución oficial de *Erlang*. Son por tanto una experiencia de éxito que juega un papel práctico similar, en el caso de *Erlang*, al del sistema de tipos de *Hindley-Milner* o las clases de tipos en el caso de *Haskell*. Nuestro trabajo adopta también la filosofía de los *success types*, intentando mejorar algunos aspectos de *Dialyzer*, como se irá exponiendo más adelante.

En [53] se formaliza un sistema de *success types* en el cual los tipos funcionales incluyen restricciones que determinan cuándo una función falla o no. Si se satisfacen las restricciones, entonces la función definitivamente fallará. El enfoque que seguimos en nuestro sistema es el opuesto: aquellos entornos finales cuyas restricciones *no* se satisfacen para ninguna asignación dada de variables a valores son semánticamente vacíos. Los tipos que pueden ser derivados con [53] y los conjuntos de restricciones que determinan los fallos en ejecución están expresados de forma recursiva, dado que los autores enfocan su trabajo en aquellos errores que involucran estructuras de datos recursivas distintas de las listas. La satisfactibilidad de estos conjuntos de restricciones es no decidible en general. En [102] se propone una extensión al algoritmo de Lindahl y Sagonas [62], haciendo uso de técnicas de *slicing* [110], para encontrar y avisar de errores con el uso de variables de un programa. Esta técnica tiene como objetivo detectar expresiones dentro del código que contengan errores, aunque estas no estén afectando al resultado final de la función donde se encuentran, para eliminar o modificar dichas partes del código y evitar así futuros errores en el desarrollo posterior.

Tanto *Dialyzer* como *TypEr* están basadas en la noción de *success types*, que son una sobreaproximación a la semántica real de las expresiones. La semántica de una expresión representa el conjunto de valores a los que dicha expresión puede ser evaluada, siendo un conjunto y no un único valor debido al no determinismo que la concurrencia introduce en *Erlang*. Dentro de este contexto tenemos un tipo denominado `none()` que denota el conjunto vacío de valores, por lo que si una expresión recibe como tipo `none()` por parte del sistema, podemos asegurar que su evaluación no devolverá ningún resultado, ya sea porque no finaliza o porque la evaluación falla en algún momento. En este sentido, las expresiones que tengan a `none()` como *success type* son el análogo más cercano a una *expresión mal tipada* en los sistemas de tipo estándar. Sin embargo, a diferencia de los sistemas estándar, podemos encontrarnos con expresiones que tengan un *success type* diferente de `none()` y cuya evaluación pueda no tener éxito. Este enfoque para analizar tipos encaja dentro de la filosofía pragmática de *Erlang*. Uno de los principios de diseño de *Dialyzer* es que no se deben requerir anotaciones de tipo por parte del programador, ni producir *falsos positivos* en el análsis. Si *Dialyzer* informa de un error de tipo, dicho error está destinado a ocurrir en tiempo de ejecución.[2] Como se señala en [100], el lema *'los programas bien tipados nunca fallan'* de los tipos *Hindley-Milner* es sustituido en el enfoque usado por *Dialyzer* por *'los programas mal tipados siempre fallan'*.[3] Como se ha indicado previamente, nuestro trabajo entronca plenamente con esta filosofía.

Antes de entrar en más detalle sobre las diferencias entre los *success types* y los sistemas de tipos *Hindley-Milner*, vamos a introducir unas pinceladas del lenguaje *Erlang* para estudiar algunos ejemplos que luego analizaremos, aunque en el capítulo 2 podremos ver en más detalle la sintaxis de *Erlang*. Como viene siendo habitual en la mayoría de los lenguajes funcionales, un programa en *Erlang* consiste en una serie de definiciones de función. Como ejemplo vamos a introducir la siguiente función `halve`, que divide su valor recibido como parámetro por dos:

```
halve(X) -> X / 2.
```

Podemos incluir una *guarda* a `halve` para restringir el conjunto de valores de entrada que acepta. Por ejemplo, si queremos asegurarnos de que `halve` solo se aplica a números enteros pares, podemos reescribir la función[4] del siguiente modo:

```
halve1(X) when X rem 2 == 0 -> X div 2.
```

donde `div` y `rem` son operadores infijos que denotan la división entera y el resto de la división, respectivamente. Con esta guarda, la evaluación de una expresión como `halve1(3)` fallaría en tiempo de ejecución.

Una definición de función en *Erlang* puede tener varias *cláusulas*, que son inspeccionadas en el orden definido cada vez que una función es aplicada. Dicho esto, podemos extender el ejemplo previo con una nueva cláusula para manejar los casos que harían fallar la función anterior:

[2] Suponiendo que el programa es terminante.
[3] De nuevo, esto se aplica estrictamente solo a las ejecuciones que finalizan.
[4] Usamos un subíndice situado debajo del nombre de la función solo para fines de presentación, de modo que podamos referirnos a la definición correspondiente más adelante.

```
halve2(X) when X rem 2 == 0 -> X div 2;
halve2(X) when X rem 2 == 1 -> not_even.
```

En el caso de recibir como entrada un número impar, la función devuelve `not_even`, que es un *átomo*. En *Erlang*, los átomos son constantes simbólicas. Viendo el código, uno podría pensar que la guarda `X rem 2 == 1`—en la segunda cláusula—es redundante y que podríamos quitarla para obtener lo siguiente:

```
halve3(X) when X rem 2 == 0 -> X div 2;
halve3(X) -> not_even.
```

Sin embargo, esta definición no es equivalente a la dada para $\texttt{halve}_2$, ya que $\texttt{halve}_3$ puede ser aplicada a valores no enteros tales como el átomo `foo`. Así $\texttt{halve}_3$`(foo)` sería evaluada a `not_even` mientas que la evaluación de $\texttt{halve}_2$`(foo)` fallaría. Si queremos que $\texttt{halve}_3$ compruebe que la entrada es un número entero sin necesidad de calcular `X rem 2`, podemos añadir la siguiente guarda a la segunda cláusula:

```
halve4(X) when X rem 2 == 0 -> X div 2;
halve4(X) when is_integer(X) -> not_even.
```

en la cual la guarda `is_integer(X)` será cierta cuando `X` sea un valor entero.

Además de números y átomos, *Erlang* también soporta listas y tuplas. La sintáxis de las listas es similar a la usada en *Prolog*: `[]` denota la lista vacía y `[X|Xs]` denota la lista cuya cabeza es `X` y cuya cola es `Xs`. En cuanto a las tuplas, la expresión $\{e_1, \ldots, e_n\}$ denota una tupla de aridad n. Por ejemplo, vamos a extender $\texttt{halve}_3$ para devolver una tupla que contenga el valor con el que la función ha sido aplicada, en caso de que no sea un número par:

```
halve5(X) when X rem 2 == 0 -> X div 2;
halve5(X) -> {not_even, X}.
```

Ahora, con estos ejemplos, podremos entrar en más detalle en el concepto de *success types*. Al principio de la sección hemos señalado que los *success types* sobreaproximan el conjunto de valores a los que una expresión es evaluada, lo cual tiene como consecuencia que también sobreaproxima el conjunto de *pares entrada-salida* que definen el comportamiento de una función. Por ejemplo, dada la función $\texttt{halve}_1$ definida arriba, el tipo `(integer())` → `integer()` sería un *success type* para esta función. Este tipo indica que la función espera como argumento un número entero; de lo contrario fallará o no terminará. En caso de ejecutarse con éxito, devolverá un número entero. Desde un punto de vista semántico, este tipo representa todas las funciones cuyos grafos (es decir, conjuntos de pares entrada-salida) son de la forma (n, m), donde n y m son números enteros. Por lo tanto, el tipo anterior es una sobreaproximación del comportamiento de $\texttt{halve}_1$, porque para que fuera una aproximación

exacta no debería permitir, por ejemplo, la tupla $(5,0)$ ya que esta no pertenece al grafo de $\texttt{halve}_1$. En cambio, sabemos que cualquier par entrada-salida que tenga valores no enteros como primera componente no puede pertenecer al grafo de ninguna función con el tipo $(\texttt{integer()}) \to \texttt{integer()}$ y por consiguiente tampoco puede pertenecer al grafo de $\texttt{halve}_1$. Por lo tanto podemos decir que una aplicación de función como $\texttt{halve}_1\texttt{(foo)}$ fallaría en tiempo de ejecución sin duda alguna.

Bajo este sistema, cada expresión literal (tales como un entero o un átomo) constituye por sí sola un *tipo unitario* que representa dicho literal. En particular, `not_even` es un tipo que representa únicamente al átomo `not_even`. Por lo tanto, un posible *success type* para la función $\texttt{halve}_2$ sería $(\texttt{integer()}) \to \texttt{integer()} \cup \texttt{not\_even}$. El lado derecho de la flecha contiene un *tipo unión* que indica que $\texttt{halve}_2$ puede devolver un número entero o el átomo `not_even`.

Los *success types* pueden ser en ocasiones poco intuitivos en comparación con los tipos *Hindley-Milner*. A primera vista, uno podría pensar que el *success type* obtenido para $\texttt{halve}_2$ también podría ser utilizado para $\texttt{halve}_3$, pero sería incorrecto hacerlo. Lo cierto es que la expresión $\texttt{halve}_3\texttt{(foo)}$ se evalúa a `not_even`, por lo que el par $(\texttt{foo},\texttt{not\_even})$ está contenido en el grafo de $\texttt{halve}_3$, pero no está representado por $(\texttt{integer()}) \to \texttt{integer()} \cup \texttt{not\_even}$. Para incluir este par necesitaríamos un tipo más genérico para $\texttt{halve}_3$; algo como $(\texttt{any()}) \to \texttt{integer()} \cup \texttt{not\_even}$, donde el tipo `any()` denota todos los valores de *Erlang*. El problema es que una función con este tipo permite cualquier valor de entrada, por lo que si todavía quisiéramos obtener el tipo `integer()`, tendríamos que añadir una guarda con `is_integer` a la segunda cláusula, como hicimos con $\texttt{halve}_4$. No obstante, suponiendo que el programador elige *no incluir* semejante guarda (porque quisiera, por ejemplo, que $\texttt{halve}_3$ aceptara cualquier valor de entrada), todavía podríamos describir con mayor precisión el comportamiento de $\texttt{halve}_3$ usando los *success types sobrecargados*. Es decir, el tipo funcional sobrecargado[5] $(\texttt{integer()}) \to \texttt{integer()} \sqcup (\texttt{any()}) \to \texttt{not\_even}$ denotaría todas las funciones que pueden devolver un entero cuando reciben un entero como entrada y que pueden devolver `not_even` en cualquier caso.

El tipo $\{\tau_1,\ldots\tau_n\}$ denota el conjunto de tuplas de aridad n tales que, para cada $i \in \{1..n\}$, la componente i-ésima tiene el tipo τ_i, por lo que la función $\texttt{halve}_5$ acepta el siguiente *success type* sobrecargado: $(\texttt{integer()}) \to \texttt{integer()} \sqcup (\texttt{any()}) \to \{\texttt{not\_even}, \texttt{any()}\}$. Suponiendo que la única información que tuviéramos sobre $\texttt{halve}_5$ fuese este *success type*, el *success type* más preciso que podríamos llegar a inferir para la expresión $\texttt{halve}_5\texttt{(foo)}$ sería $\{\texttt{not\_even}, \texttt{any()}\}$. La limitación que percibimos es que un tipo monomórfico como $(\texttt{any()}) \to \{\texttt{not\_even}, \texttt{any()}\}$ no establece ninguna conexión entre las dos apariciones de la variable `X` en la cláusula $\texttt{halve}_5\texttt{(X) -> \{not\_even, X\}}$. En esta tesis—y esta es su contribución esencial—se presentan los *success types polimórficos*, que nos permiten indicar una relación entre la entrada y la salida de una función, la cual se realiza mediante las *variables de tipo polimórficas*. Por ejemplo, el tipo $(\texttt{integer()}) \to \texttt{integer()} \sqcup \forall\alpha.(\alpha) \to \{\texttt{not\_even}, \alpha\}$ es un *success type* de $\texttt{halve}_5$, que nos permite derivar el tipo $\{\texttt{not\_even}, \texttt{foo}\}$ para la expresión $\texttt{halve}_5\texttt{(foo)}$.

[5] Usamos el operador $\sqcup$ en lugar de $\cup$ por cuestiones técnicas que hacen que la semántica de $\cup$ sea más restrictiva a la hora de manejar los valores a los que se instancian las variables de tipo. Esta restricción no es necesaria a la hora de manejar tipos funcionales y de ahí la necesidad de realizar esta diferenciación entre uniones de tipos.

Además de lo que acabamos de ver, las variables polimórficas pueden ser restringidas. Es decir, si hubiéramos añadido la guarda `when is_integer(X)` a la segunda cláusula de $\texttt{halve}_5$, un posible *success type sobrecargado* para esta función sería el siguiente: $(\texttt{integer()}) \rightarrow \texttt{integer()} \sqcup \forall\alpha.(\alpha) \rightarrow \{\texttt{not\_even}, \alpha\}$ **when** $\alpha \subseteq \texttt{integer()}$.

Los sistemas con *success types* conllevan varias particularidades que no existen en los sistemas estándar *Hindley-Milner*. Por ello vamos a analizar algunas de ellas un poco más:

- En un sentido estricto no existen expresiones mal tipadas—en el sentido de no aceptar ningún tipo—en un sistema basado en *success types*, pues toda expresión tiene como mínimo un *success type*: `any()`.
- No existe ningún algoritmo que pueda, en general, calcular todos los *success types* para una expresión. Esto se debe a que la noción de *success type* es semántica, en lugar de estar definida por un conjunto de reglas. Por ejemplo, supongamos la siguiente expresión:

  ```
  case b of
     true -> 1;
     false -> 2
  end
  ```

  en la que *b* es una expresión compleja que siempre se evalúa a `true`. Resulta que `1` sería un *success type* para esta expresión, pero un algoritmo que pudiera inferir tal tipo necesitaría saber que *b* es evaluada siempre a `true`, lo cual es un problema indecidible.
- Las expresiones analizadas no siempre tienen un "mejor" *success type*, que sería el *success type* mínimo. Por ejemplo, suponemos la siguiente función:

  ```
  g() -> case rand:uniform(2) of
            1 -> 0;
            2 -> {ok, g()}
         end.
  ```

  donde `rand:uniform(2)` puede ser evaluada a `1` o `2`. Existe una cadena decreciente infinita de *success types* para la expresión `g()`, usando el operador $\supseteq$ para representar la relación de subtipado entre los tipos de la cadena, consistente en:

  $$\texttt{any()} \supseteq 0 \cup \{\texttt{ok}, \texttt{any()}\} \supseteq 0 \cup \{\texttt{ok}, 0 \cup \{\texttt{ok}, \texttt{any()}\}\} \supseteq 0 \cup \{\texttt{ok}, 0 \cup \{\texttt{ok}, 0 \cup \{\texttt{ok}, \texttt{any()}\}\}\} \supseteq \ldots$$

  Esto contrasta con lo que sucede con *Hindley-Milner*, donde toda expresión bien tipada tiene un mejor tipo—en el sentido de más general, que es lo adecuado en ese contexto—que es el denominado *tipo principal*.

- Existe una relación de subtipado entre tipos, que también está definida de forma semántica. Sin embargo, esta relación es covariante tanto en los argumentos como en el resultado de un tipo funcional. Según [55], el tipo $(\tau) \rightarrow \tau'$ es un *success type* para una función f si y solo si para cada par de valores v y v':

$$f(v) \text{ es evaluada a } v' \Longrightarrow v \text{ es de tipo } \tau \wedge v' \text{ es de tipo } \tau'$$

  Por ello, de cara a probar que $(\tau_1) \rightarrow \tau'_1 \subseteq (\tau_2) \rightarrow \tau'_2$ es suficiente demostrar que $\tau_1 \subseteq \tau'_1$ y $\tau_2 \subseteq \tau'_2$. Por el contrario, en el contexto de un sistema estándar *Hindley-Milner*, resulta que si $(\tau) \rightarrow \tau'$ es un tipo para f, entonces:

$$f(v) \text{ es evaluada a } v' \wedge v \text{ es de tipo } \tau \Longrightarrow v' \text{ es de tipo } \tau'$$

  Dado que la condición de que v es de tipo τ está en el lado izquierdo de la implicación, la condición suficiente de covarianza $\tau_1 \subseteq \tau'_1$ mostrada antes se convierte en una relación contravariante: $\tau'_1 \subseteq \tau_1$.

- La noción de polimorfismo es más sutil en el contexto de los *success types* que en los sistemas de tipo de *Hindley-Milner*. En *Hindley-Milner*, cualquier instancia de un esquema de tipo polimórfico válido para una expresión es un tipo válido para dicha expresión. Por ejemplo, si $\forall\alpha.(\alpha) \rightarrow \alpha$ es un tipo *Hindley-Milner* válido para la función identidad, entonces también lo han de ser (`bool()`) → `bool()` e (`integer()`) → `integer()`. Esto no es verdad cuando usamos *success types*, dado que estos dos *success types* monomórficos son incompatibles para una misma expresión (los tipos corresponden con grafos de función disjuntos). De hecho, el primer tipo monomórfico prohibiría la aplicación de la función identidad a un número entero, que sin embargo es una expresión que siempre tendría éxito.

1.4. *Dialyzer*: un analizador de discrepancias para *Erlang*

En la sección anterior hemos introducido al lector en los *success types* y señalado que *Erlang* dispone de la herramienta *Dialyzer* [99] para analizar estáticamente los tipos en los programas escritos en este lenguaje. Siendo *Dialyzer* una gran herramienta para encontrar errores en tiempo de ejecución, trae consigo también algunos defectos y debilidades. El primero de ellos es que no es capaz de inferir tipos polimórficos para las funciones.[6] Los programadores pueden incluir especificaciones de tipo polimórficas en sus programas [55], pero el algoritmo de inferencia no está capacitado para manejar las variables de tipo polimórficas, por lo que estas son transformadas en tipos monomórficos antes de realizarse el análisis. La herramienta tampoco está diseñada para inferir por sí sola especificaciones de tipos funcionales sobrecargados, aunque se permite a los usuarios introducir estos tipos sobrecargados para etiquetar sus funciones. Recordemos que un tipo funcional sobrecargado representa una

[6] Sin embargo, esto sí es posible en los sistemas de tipado estático, como se muestra en el caso de *Haskell*.

colección de tipos funcionales que representan a su vez las diferentes ramas de ejecución de una función, donde los tipos de entrada y de salida de una rama pueden diferir de los que hay en las otras ramas.

Como ya se mencionó en la sección 1.3, *Dialyzer* es una herramienta que infiere *success types* para toda expresión en un programa y detecta cuándo una subexpresión admite el tipo `none()`, lo cual significa que su evaluación está destinada a fallar en tiempo de ejecución. Con los ejemplos de las funciones `halve`, de la sección 1.3, *Dialyzer* infiere para $\texttt{halve}_1$ el tipo (`integer()`) → `integer()` y para $\texttt{halve}_2$ el tipo (`integer()`) → `integer()` ∪ `not_even`. En cuanto a $\texttt{halve}_3$, se infiere el *success type* (`any()`) → `integer()` ∪ `not_even` para la función, que difiere del tipo sobrecargado (`integer()`) → `integer()` ⊔ (`any()`) → `not_even`, que sería lo deseable. A pesar de que *Dialyzer* no es capaz de inferir especificaciones de tipo sobrecargadas, el usuario puede utilizarlas en sus programas y permitir así que *Dialyzer* haga uso de ellas para obtener información adicional, obteniendo así resultados más precisos. Para ilustrar esta problemática, veamos el siguiente ejemplo:

```
-spec f(0) -> 1; (1) -> 2.
f(0) -> 1;
f(1) -> 2.
g() -> f(f(1)).
```

Téngase en cuenta que el punto y coma de la especificación representa al operador ⊔, dentro de la sintaxis que *Dialyzer* utiliza junto el lenguaje *Erlang*. Con esta especificación sobrecargada de `f`, *Dialyzer* nos informará de que la evaluación de `g()` fallará durante la ejecución, dado que la expresión `f(1)` tiene el tipo `2` como *success type*, el cual no pertenece al dominio de `f`, por lo que no se puede usar como parámetro de entrada en la aplicación externa de `f`. Sin la especificación sobrecargada dada por el usuario para `f`, *Dialyzer* habría obtenido como *success type* (`0` ∪ `1`) → `1` ∪ `2`. Como resultado, esto provocaría que la expresión `f(1)` obtuviera el tipo `1` ∪ `2`, y también lo haría `f(f(1))`, por lo que no se informaría de ninguna colisión entre tipos incompatibles.

Dialyzer no dispone de especificaciones sobrecargadas para funciones en las que los tipos de los dominios sean no disjuntos. Tales especificaciones son ignoradas. En nuestro ejemplo de $\texttt{halve}_3$, la especificación (`integer()`) → `integer()` ⊔ (`any()`) → `not_even` sería ignorada, ya que `integer()` y `any()` no denotan conjuntos de elementos disjuntos (de hecho, hay una relación de subtipado entre ambos tipos). Sin embargo tal apreciación en las especificaciones podría ser de utilidad a la hora de obtener resultados más precisos en determinados casos. Por ejemplo, supongamos la siguiente definición:

```
halve_plus_one(X) -> halve3(X) + 1.
```

Un *success type* correcto para `halve_plus_one` es (`integer()`) → `integer()`. El hecho de que la subexpresión $\texttt{halve}_3\texttt{(X)}$ aparezca como argumento del operador + fuerza que la rama con el tipo

(`any()`) → `not_even` en la especificación sobrecargada de $\texttt{halve}_3$ quede descartada, y por ello `X` estaría obligada a tener tipo `integer()`. Sin un tipo sobrecargado, el tipo que se habría inferido para `X` sería `any()`. No obstante, las reglas para derivar tipos que presentamos en esta tesis nos permitirán derivar el tipo (`integer()`) → `integer()` para `halve_plus_one`.

Respecto al polimorfismo, *Dialyzer* permite especificaciones dadas por el usuario con variables de tipo. Por ejemplo, para $\texttt{halve}_5$ el usuario podría especificar el tipo (`integer()`) → `integer()` ⊔ $\forall\alpha.(\alpha) \rightarrow \{\texttt{not\_even}, \alpha\}$. Desafortunadamente, aunque se permita al usuario incorporar esta especificación a la función, la información adicional—que ofrecería el polimorfismo—se pierde cuando las aplicaciones a esta función sean analizadas. Todas las apariciones de α en el tipo de $\texttt{halve}_5$ son colapsadas en el tipo `any()`, lo que supone que el tipo de $\texttt{halve}_5$ sería en realidad: (`integer()`) → `integer()` ⊔ (`any()`) → {`not_even`, `any()`}, perdiendo así la conexión entre la entrada y la salida.

Veamos ahora cómo se comporta *Dialyzer* con la función `map`, ya que se trata de un ejemplo relevante a la hora de programar en lenguajes funcionales. La implementación básica de la función `map` en *Erlang* es la siguiente:

```
map(_, []) -> [];
map(F, [X|XS]) -> [F(X)|map(F, XS)].
```

El tipo inferido que *TypEr* nos ofrece es:

```
-spec map(any(), [any()]) -> [any()].
```

Como podemos ver, el tipo inferido usa el tipo `any()` prácticamente para cada elemento en el tipo funcional. El tipo `any()` representa todos los posibles valores que pueden ser representados con la semántica del lenguaje. La única información que podemos obtener de nuestra función `map` es que el segundo parámetro de entrada y el resultado son listas. Sin embargo, la función admite el siguiente tipo funcional sobrecargado:

$$(\texttt{any()}, \texttt{[]}) \rightarrow \texttt{[]} \sqcup \forall\alpha,\beta,\alpha',\beta'.\left(\alpha \rightarrow \beta, \texttt{[}\alpha'\texttt{]}\right) \rightarrow \texttt{[}\beta'\texttt{]} \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta$$

Podemos ver en el ejemplo que ni las variables de tipo polimórficas, ni el tipo funcional sobrecargado, han sido inferidos por *TypEr*. Este ejemplo nos sirve también para mostrar una vez más las diferencias con sistemas *Hindley-Milner*. En estos el tipo inferido para `map` sería $\forall\alpha,\beta.\left((\alpha) \rightarrow \beta, \texttt{[}\alpha\texttt{]}\right) \rightarrow \texttt{[}\beta\texttt{]}$, que es el tipo más general posible que, de acuerdo con el principio *'los programas bien tipados nunca fallan'*, asegura ausencia de errores de tipo en tiempo de ejecución.

Sin embargo, ese no es un *success type* correcto, pues el programa no exige que para que una aplicación de `map` sea exitosa el primer argumento de `map` deba ser una función. Eso solo ocurre si el segundo argumento de `map` es una lista no vacía, y de ahí el tipo sobrecargado polimórfico que hemos mostrado arriba.

Tener en cuenta el polimorfismo permite a una herramienta de inferencia de tipos anticipar durante la compilación algunos errores del programa que habrían pasado desapercibidos si la herramienta hubiera colapsado las variables de tipo al tipo `any()`. Este es precisamente uno de los objetivos principales de la tesis. Como ejemplo, vamos a considerar la función identidad `id`, la cual tendrá el tipo $\forall\alpha.(\alpha) \to \alpha$. Este tipo es más preciso que usar `(any()) -> any()`, dado que el primero nos permitiría derivar que la evaluación de `id(true) + 3` no tendría éxito. De un modo similar, las variables de tipo polimórficas nos permiten detectar si los elementos de una lista pasada como argumento a la función `map` son disjuntos del dominio de la función recibida como parámetro por `map`. En [66] se introduce una transformación de programas para que *Dialyzer*, cuando se ejecuta sobre el programa transformado, use una instancia monomórfica adecuada para cada aplicación de función. Sin embargo, derivar esquemas de tipo polimórfico para funciones no es trivial, ya que el significado de las variables de tipo polimórficas no está tan bien estudiado en el contexto de los *success types* como en el contexto de los sistemas de tipo *Hindley-Milner*. Debido a la dificultad para derivar e inferir tipos polimórficos, conseguir con los *success types* polimórficos un tratamiento preciso y sólidamente fundamentado ha resultado ser un trabajo de gran complejidad técnica para esta tesis.

1.5. Alternativas actuales disponibles

En esta sección vamos a explorar las alternativas disponibles que existen para realizar análisis estático de tipos sobre lenguajes de tipado dinámico. Primero, en la sección 1.5.1, vamos a introducir una visión general de las alternativas que existen a nivel teórico. En la sección 1.5.2 hablaremos de sistemas de tipos existentes para trabajar con otros lenguajes de tipado dinámico: *JavaScript* y *Python*. Hay que señalar que algunos de estos sistemas de tipos se basan en el enfoque denominado *gradual typing* [107, 108]. Por ejemplo: *JavaScript* tiene *TypeScript* [13] y *Flow* [19], y *Python* tiene *MyPy* [103]. Por último, en la sección 1.5.3 veremos la relación que las herramientas de la sección anterior tienen entre las anotaciones y los falsos positivos. Los ejemplos que veremos en estas secciones estarán escritos en los lenguajes *Erlang*, *TypeScript* y *Python*.

1.5.1. Trabajos relacionados

La comunidad científica, desde hace algunas décadas, ha llevado a cabo numerosas líneas de investigación para aplicar análisis estático basado en tipos a lenguajes de tipado dinámico. Algunos ejemplos bien conocidos han sido aplicadas al lenguaje *JavaScript* [8], donde encontramos herramientas como *Typescript* [13] y *Flow* [19], así como con *Python* también tenemos las herramientas *RPython* [7] y *MyPy* [103]. No obstante, también encontramos que estas técnicas han sido aplicadas a otros lenguajes, como por ejemplo el caso de *Ruby* [36, 6], *Scheme* [122] y *Smalltalk* [114]. A pesar de que *RPython* está orientado a la traducción de *Python* a instrucciones primitivas de la JVM[7] y del CLI[8] (en lugar de

[7] Java Virtual Machine, que es la máquina virtual de la plataforma Java.
[8] Common Language Infrastructure, que es la máquina virtual de la plataforma .NET de Microsoft.

emular el modelo de *Python* usando la máquina virtual correspondiente), estos sistemas permiten al programador capturar errores de tipo en tiempo de compilación. Sin embargo, estos trabajos siguen un enfoque más tradicional de asegurar la ausencia de errores de tipo en tiempo de ejecución, incluso si ello conlleva dar falsos positivos. El sistema de tipos de nuestro trabajo sigue el objetivo opuesto introducido por los *success types* [62], es decir, evitar los falsos positivos. Nuestro objetivo es asistir al programador en la detección del mayor número posible de errores seguros. Aunque puedan quedar otros errores de tipo más sutiles sin señalar, este enfoque puede combinarse con una variedad de mecanismos que *Erlang* dispone (tales como árboles de supervisión) para informar de fallos y restaurar el estado de un programa en caso de que este falle.

Soft Typing

El *soft typing* [16] es otro enfoque a la hora de aplicar análisis estático basado en tipos, que consiste en encontrar aquellas localizaciones en un programa donde la consistencia de tipos no está garantizada, para insertar en esas localizaciones comprobaciones de tipo en tiempo de ejecución. Este enfoque comparte con *Dialyzer* una filosofía similar en la medida en que no se requieren anotaciones de tipo por parte del programador. La comprobación de *soft types* no rechaza programas con errores de tipo potenciales, pero al contrario que con los *success types*, es más conservador en el sentido de que inserta comprobaciones de tipo siempre que la ausencia de errores de tipo no esté garantizada. Algunas implementaciones de los sistemas de *soft typing* han sido desarrolladas para *Scheme* [122] y *Erlang* [86], en este último caso introduciendo un lenguaje de especificación para especificar la interfaz de los módulos de *Erlang*. Como ha sido reconocido por su autor, este último sistema podría producir falsos positivos como ocurriría por ejemplo con la función `lists:nth/2` cuando no existe garantía alguna de que se acceda a la lista dentro de sus límites. Otra diferencia con nuestro trabajo respecto al *soft typing* es la incorporación en nuestro trabajo de entornos de tipos como parte del resultado final, capturando así las condiciones necesarias para la evaluación de la expresión. Además, las restricciones que acompañan a los tipos en nuestro sistema es otra diferencia que mejora los tipos funcionales polimórficos.

Gradual Typing

Otra área de investigación relacionada con la integración del tipado estático en lenguajes de tipado dinámico son los sistemas basados en *gradual typing* [107, 108]. Al contrario que el enfoque del todo-o-nada que ofrecen los lenguajes tradicionales, un sistema de *gradual types* permite a los programadores anotar parcialmente sus programas con tipos, mientras que las partes sin anotar de los programas tienen tipos dinámicos de forma implícita, lo cual correspondería aproximadamente con el tipo `any()` que usamos en nuestro trabajo. Hay dos propiedades que definen la base de los *gradual types*. La primera nos dice que se capturarán todos los fallos de tipo cuando un programa esté completamente anotado. La segunda es que, tras analizar y transformar cualquier programa, todos los errores

de tipo han de ser capturados, ya sea de forma estática o mediante excepciones capturadas en tiempo de ejecución. Se ha estudiado la aplicación del polimorfismo sobre sistemas basados en *gradual typing* [50], tomando como partida el *Sistema F* [96, 39] para extenderlo en dos nuevos sistemas que permiten *gradual types polimórficos* con subtipado. Otro enfoque consiste en estudiar cómo refinar los *gradual types* [57, 109] para mejorar los resultados obtenidos, pero sin romper con las propiedades que definen a los *gradual types*. Los sistemas con *gradual types* han sido también estudiados en el contexto de los lenguajes imperativos [95, 119]. Existen algunos algoritmos de inferencia para *gradual types* [94] cuyo funcionamiento consiste en ir restringiendo los tipos de variables a partir de sus definiciones y asignaciones (*entradas*) y a partir de los contextos donde van apareciendo (*salidas*). Lo último limita el conjunto de valores que una variable puede contener en tiempo de ejecución, de un modo similar a como en nuestros sistema los entornos de tipos representan un límite superior a los valores de todas las variables que están en ámbito. Sin embargo, el objetivo de la inferencia con *gradual types* no es detectar discrepancias de tipos, sino la realización de optimizaciones de rendimiento [41], de un modo similar a lo que hacen los sistemas de *soft types*. En este sentido, podemos decir que nuestro sistema de tipos es más cercano a la noción de *success types* que los *gradual types*.

La evolución de los *gradual types* llevó a crear *Gradualizer* [23], como una propuesta de algoritmo que genera un sistema de *gradual types* partiendo de un sistema de tipos bien formado y que genera un compilador de *cast calculus* [106], cuya semántica describe las comprobaciones de tipos que se realizan al hacer un *casting* de tipos, dado que una de las aplicaciones que tienen estos sistemas es dar soporte a los lenguajes de programación para que puedan combinar el tipado estático con el tipado dinámico. Sobre este sistema se estudió la relación de subtipado que hay entre los tipos estáticos y los tipos dinámicos [85]. Estudiar esta relación permite a los autores formalizar cuándo se puede hacer un *casting* entre tipos de forma segura al analizar expresiones. Sin embargo, para no depender del *cast calculus* propuesto antes, en [5] se propone el concepto de *Confined Gradual Typing* para mejorar el rendimiento a la hora de realizar los *casting* de tipos, para que luego en [38] se proponga el *Abstracting Gradual Typing* (AGT), en el que, dado un juicio de tipo estático guiado por la sintaxis del lenguaje [21], se induce su correspondiente juicio de *gradual typing*, apoyándose en las derivaciones realizadas para garantizar la seguridad de los tipos, en lugar de usar la semántica del *cast calculus*. En cuanto a los trabajos más recientes sobre *gradual types*, en [105] podemos ver cómo tomar un lenguaje de tipado estático para añadir tipado dinámico de forma segura, poniendo como ejemplo el lenguaje de transacciones financieras usado dentro de la empresa *Bloomberg*. En [104] podemos ver una evolución de los AGT para mejorar su eficiencia en cuanto a las semánticas usadas para la comprobación de tipos. En [51] se discute sobre el enfoque original de Siek [107, 108] de cómo adaptar los *gradual types* a lenguajes de tipado estático previos. En ese trabajo se propone que los sistemas de *gradual types* sean paramétricos con respecto a los lenguajes de tipado estático en lugar de amoldarse a la semántica concreta de cada uno. En [81] se propone usar técnicas de verificación de contratos para mejorar la eficiencia, disminuyendo la sobrecarga que conllevan las comprobaciones innecesarias, dado que mediante el análisis propuesto se puede garantizar si es seguro prescindir de una comprobación de tipos en un código con tipos dinámicos o no.

Hindley-Milner

Dentro de los sistemas del estilo *Hindley-Milner* ha habido diversas líneas de investigación buscando incorporar la noción de subtipado como parte del sistema. Los *principal typings* [54] es otro enfoque propuesto, como evolución al trabajo de Damas [28], para aplicar análisis estático basado en tipos sobre el lenguaje *ML* [79, 80] incorporando el subtipado como parte del mismo. En dicho trabajo se señala que los sistemas de tipos propuestos por Damas tienen dos propiedades, una primera definida como la existencia de *principal types* y la segunda como la existencia de *principal typings*. La propiedad de *principal types* analiza qué tipo puede representar todos los tipos posibles para una expresión bajo un entorno, mientras que la propiedad de *principal typings* analiza qué juicio de tipado representa todos los posibles tipos para una expresión. Con la propuesta del autor, añadiendo tipos intersección[9] y la pertinente operación de subtipado para seleccionar cada tipo en dichas intersecciones, se busca complementar trabajos anteriores para realizar inferencias incrementales, así como inferir tipos polimórficos recursivos, de modo que se puedan inferir todos los tipos posibles para una expresión. Este enfoque no es práctico cuando se tienen tipos unitarios, como ocurre en nuestro sistema basado en *success types*. En el trabajo [32] se propone el lenguaje *MLsub* como una versión de *ML* donde el sistema de tipos ofrece tipos con intersecciones y uniones, para poder introducir el subtipado como parte del sistema. En la tesis desarrollada por Petrucciani [90] se extiende la teoría de tipos conjuntistas para incorporar la noción de subtipado, junto al polimorfismo y los tipos unión, dentro de un marco teórico que define su sistema con el que poder analizar lenguajes basados en el cálculo lambda. También en este trabajo se muestra cómo aplicar su sistema de tipos sobre sistemas de *gradual types* que analicen lenguajes funcionales. Al estar formalizado sobre la teoría de tipos conjuntistas, también permite disponer de tipos intersección y tipos negación. Dentro del ámbito de *Erlang*, en el trabajo [87], se propone un sistema de tipos basado en subtipado con polimorfismo que garantice la seguridad del código, por encima de evitar posibles falsos positivos como es el caso de los *success types*.

Tipos basados en restricciones

A pesar de que los sistemas de tipo que sobreaproximan los comportamientos en tiempo de ejecución son raramente utilizados con lenguajes funcionales (a excepción de los *success types* en *Erlang*), esta idea ha sido ampliamente estudiada en el contexto de la programación lógica con *Prolog* [26]. Heintze y Jaffar [46] introdujeron la transformación de programas lógicos en conjuntos de restricciones que infieren una aproximación de su comportamiento, además de un método para simplificar dichos conjuntos. Otro enfoque para analizar y capturar el significado de un programa lógico es el basado en *regular types* [31, 123], que pueden ser expresados con programas lógicos con predicados unarios.[10] La comprobación de tipos es decidible para los *regular types* y la inferencia es factible mediante el uso

[9] El concepto de intersección de tipos, que se maneja en este trabajo, se puede relacionar con los tipos unión que manejamos en nuestro trabajo, ya que ambos representan la unión de todos los tipos que representan a una expresión.

[10] Esto es, programas cuyos predicados solo admiten un parámetro.

de técnicas de interpretación abstracta ascendente con un operador de *widening* adecuado [37, 118]. En [42] se propone una implementación de un sistema con *gradual types polimórficos* con subtipado, basado en el trabajo de Mycroft-O'Keefe [82], para incorporar anotaciones de tipo en *Prolog* y poder realizar comprobaciones de tipos en tiempo de ejecución. Además de lo anterior, en [92, 93] se introduce una semántica de *aplicaciones exitosas* para la Programación Lógica con Restricciones (CLP[11])—con un sistema de tipos definido—y se proporcionan condiciones de verificación para dicha semántica, con el fin de detectar errores en los programas mediante análisis estáticos.

Los enfoques relacionados con la programación lógica que hemos mencionado hasta ahora involucran únicamente tipos monomórficos. Barbuti y Giacobazzi [12] introducen en su sistema el polimorfismo, usando interpretación abstracta para realizar la inferencia, dada una definición de predicado, de un conjunto de objetivos exitosos abstractos para dicho predicado que puede contener variables de tipo. Sin embargo, este conjunto no sobreaproxima el conjunto de objetivos exitosos para el predicado que ha sido definido, dado que algunos objetivos refutables podrían estar mal tipados. Los autores además describen de forma breve cómo introducir los tipos unión en su sistema. Su enfoque es posteriormente refinado por Lu [69, 70], quien diseña un análisis para inferir dependencias entre los parámetros de un predicado lógico. Dichas dependencias están expresadas con reglas que involucran variables polimórficas, añadiendo en [72, 71, 73] los tipos unión como parte de la evolución de su sistema. Al contrario que en [12], los resultados demuestran que se sobreaproxima el conjunto de objetivos exitosos en los predicados de los programas. De un modo similar a nuestro sistema, la información de los tipos es propagada en los símbolos funcionales de los argumentos al resultado y viceversa. Esta propagación se establece de forma explícita por las dependencias que se infieren mediante el análisis, mientras que en nuestro sistema de tipos esta información se encuentra implícita en los entornos de tipos. Una diferencia sustancial de nuestro sistema de tipos con respecto a estos enfoques, es que las restricciones sobre las variables de tipo no se pueden expresar en estos sistemas últimos que acabamos de mencionar.

Una variante del trabajo de Heintze y Jaffar es el presentado por Aiken [3, 1, 2] para realizar análisis de programas basados en restricciones conjuntistas, pero en lugar de aplicarlos sobre lenguajes de programación lógica, el autor lo hace sobre el cálculo lambda. Entre los trabajos que han usado este enfoque, en [89] tenemos un trabajo que realiza análisis sobre el uso de los punteros en programas escritos en *C*. Recordemos que un sistema de tipos puede diseñarse para detectar situaciones concretas que se quieran considerar erróneas, en este caso controlar las asignaciones que puedan recibir los punteros.

[11] Constraint Logic Programming

Optimización

En trabajos como [44] se prefiere aplicar un enfoque donde se analiza el código a ejecutar, en este caso servidores genéricos[12] de *Erlang*, para generar código de apoyo que gestione la comprobación de tipos en tiempo de ejecución y hacer segura la ejecución de las partes analizadas. Esta metodología es similar a la vista en los *gradual types*, aunque aquí no se trata de optimizar el código eliminando comprobaciones innecesarias de tipos sino añadir las que son necesarias para evitar que la aplicación falle por un mal uso de los tipos de cara a los mensajes que recibe un proceso servidor.

Técnicas basadas en *Model Checking*

El *model checking* [24] es otro tipo de técnica automática utilizada para verificar si una propiedad, expresada en lógica modal, se cumple en un sistema concurrente. El objetivo de esta técnica es encontrar errores durante la ejecución de un sistema, simulando dicha ejecución con semánticas que definen las transiciones de estado de los lenguajes a analizar. En [52] Jakob y Thiemann proponen, para los *success types*, una alternativa al algoritmo de Lindahl y Sagonas para obtener resultados más precisos mediante el uso de *model checking* [24] sobre autómatas basados en árboles, teniendo en cuenta esquemas recursivos de encaje de patrones. En [84] se aporta un enfoque que aplica esta técnica implementando la semántica de *Core Erlang* en el lenguaje *Maude* [25]. El motivo de tomar *Core Erlang*, en lugar de *Erlang*, se debe a que su semántica está mejor documentada [15]. Sin embargo, ha habido propuestas para dar una semántica a *Erlang* más comprensible, como se muestra en [115]. A pesar de las dificultades, existe una herramienta llamada *McErlang* [35, 17, 33], preparada para realizar comprobación de modelos sobre programas en lenguaje *Erlang*. Esta herramienta está centrada en verificar programas concurrentes, buscando errores durante la ejecución de los mismos. Para lograr esta tarea, *McErlang* sustituye componentes de la plataforma *Erlang/OTP* (distribución, concurrencia y comunicación) con una versión propia que simula los procesos dentro del comprobador de modelos. Este enfoque permite acceder al estado del programa en ejecución de forma sencilla. En [113], un trabajo previo del autor de esta tesis, se propuso unas bases genéricas para comprobar modelos en *Maude* para analizar lenguajes concurrentes, tomando como caso de estudio el lenguaje *Erlang*. La idea era poder definir fórmulas en *Lógica Temporal Lineal* (LTL) para verificar programas concurrentes traducidos a un lenguaje abstracto genérico. Esta técnica—al tener un coste computacional alto, ya que necesita de árboles de búsqueda—se reserva para analizar sistemas concurrentes, donde no se puede determinar de antemano el resultado de la ejecución debido a la complejidad de los sistemas a analizar. Por ello, ya que lenguajes como *Erlang* tienen un componente muy importante de programación concurrente, el uso de *model checking* como herramienta a la hora de analizar las interacciones entre procesos es una alternativa a considerar, pudiendo acompañarla con el uso de sistemas de tipos para complementar dichos análisis.

[12] Este tipo de servidores son aquellos módulos que heredan como comportamiento el módulo `gen_server` de la biblioteca estándar de *Erlang*.

1.5.2. Analizando programas sin anotaciones de tipo

Dado que *TypeScript* y *MyPy* son herramientas basadas en *gradual types*, permiten programas sin anotaciones de tipos. El mecanismo de comprobación de tipos puede encontrar la existencia de problemas potenciales en dichos programas sin anotaciones de tipos, pero existen algunas limitaciones a la hora de hacerlo. Un ejemplo básico en *JavaScript* es el siguiente:

```
let y = 16 - "15"; // y = 1
```

En *TypeScript* obtenemos el siguiente error:

```
The right-hand side of an arithmetic operation must be of type
'any', 'number', 'bigint' or an enum type.
```

que dice que el lado derecho de una operación aritmética debe ser de tipo `'any'`, `'number'`, `'bigint'` o un tipo enumeración. Este error asume que el operador de la resta en *JavaScript* no acepta otro tipo que no sea numérico, lo cual no es cierto dentro del propio estándar del lenguaje, pero *TypeScript*, asumiendo un enfoque conservador, nos da un falso positivo para dicha sentencia. Dado que esto tiene más que ver con el comportamiento—un tanto esotérico—de los operadores en *JavaScript*, pasemos a ver un ejemplo similar en *Python*:

```
x = 8 - "4"
```

En *MyPy* obtenemos el siguiente error para esta asignación:

```
error: Unsupported operand types for - ("int" and "str")
```

que dice que los tipos de los operandos no están soportados para el operador de la resta y nos muestra los tipos de los operandos que hemos usado. En este caso, si tratamos de restar una cadena de texto a un número en *Python*, el lenguaje lanzará una excepción y el programa fallará. Tanto *TypeScript* como *MyPy* pueden determinar un tipo para valores literales y comprobar si este tipo encaja en los tipos esperados por los parámetros de un operador o función. Sin embargo, si usamos una función para encapsular el código que hemos mostrado antes:

```
function sub(a, b) {
    return a - b;
}

let x = sub(26354, "deckard");
```

TypeScript infiere para `sub` el tipo `(a:any,b:any):number`, para `x` el tipo `number` y no obtenemos ningún error como en el ejemplo anterior. No obtenemos ningún error con la llamada a `sub` usando un `string` como segundo argumento, porque cualquier valor puede encajar dentro del tipo `any`. Aparte de esto, las variables pasadas como argumentos a una función no ganan información adicional de tipo, que podría ser útil para capturar errores cuando esas variables de programa estén siendo aplicadas a otra función. Por ejemplo, si `sub` hubiera sido definida como `sub(a:number,b:number):number` y tuviéramos la siguiente función:

```
function fail(x, y) {
    x = "nietzsche";
    return sub(x, y);
}
```

El tipo inferido para `fail` es `(x:any,x:any):number`, porque con la llamada `sub(x,y)` el sistema no es capaz de deducir el tipo `number` para las variables `x` e `y`. Además tampoco es capaz de inferir que `x` es un `string`, con la asignación en la primera sentencia de la función, a la hora de analizar la llamada `sub(x,y)`. Esto es así porque *TypeScript* solo tiene en cuenta, o bien el tipo con el que etiquetamos la variable (ya sea en la definición de los parámetros de una función o en las sentencias `let`), o bien el tipo de la expresión con la que se inicializa la variable en un `let` si no hemos etiquetado la variable. Esto es un comportamiento tomado de lenguajes con tipado estático como *Java* o *C++*, lo cual impide detectar el error cometido en `fail` al asignar una cadena a la variable `x` por accidente. Sin embargo, en nuestro trabajo, las reglas del sistema sí permiten la obtención de información adicional para las variables de programa que son aplicadas a una función, pudiendo así capturar más errores en el uso de estas variables.

Con la herramienta *Flow* los resultados son similares a *TypeScript*, pero obtenemos un mensaje de error que culpa a la expresión `a - b` dentro de `sub` cuando realizamos llamadas a la función con diferentes tipos, como por ejemplo, `sub(42, 23)` y `sub(19, "junio")`.

En *Python* tenemos una situación similar:

```
def sub(a, b):
    return a - b

x = sub(2501, "motoko")
```

La función `sub` en *MyPy* tiene como tipo `(a:Any,b:Any) -> Any`, así que estamos ante la misma situación donde una cadena de texto encaja dentro del tipo `Any`. El motivo por el que esto ocurre es que en *Python* la mayoría de los operadores (incluyendo el de resta) pueden ser sobrecargados por las clases que programemos.

En nuestro sistema y en *Dialyzer*, las variables de programa pueden tener un tipo especificado por el usuario, pero estas especificaciones no son requeridas, por lo que podemos tener programas sin

anotaciones de tipo e igualmente derivar tipos para las funciones del mismo. Por lo tanto, los errores mostrados en los ejemplos de esta sección habrían sido detectados. Dado que nuestras reglas para tipar expresiones extraen información de las aplicaciones de función para modificar las variables de programa usadas como argumentos en la aplicación, podemos inferir información de tipo adicional para una abstracción cualquiera, y por consiguiente comprobar errores de tipo cuando se aplican dichas abstracciones.

1.5.3. Anotaciones y falsos positivos

En este apartado examinamos algunos ejemplos que muestran cómo los enfoques conservadores de, por ejemplo, *TypeScript*, *Flow* o *MyPy*, dan lugar a errores de tipo que corresponden a falsos positivos, lo que no sucede con el enfoque de los *success types*.

En el siguiente ejemplo en *TypeScript* vamos a tener una función que toma una lista de valores y devuelve una lista con solo aquellos que pueden ser transformados en número, construyendo para cada uno, un objeto que contenga dos atributos: uno con el elemento y otro con su sucesor.

```
interface NextInt<T> {
  value: T;
  next: number;
}

function getNexts<T>(values: Array<T>): Array<NextInt<T>> {
  return values.map(x => {
    let temp = Number(x);
    if (isNaN(temp)) {
      return undefined;
    } else {
      return { value: x, next: Math.trunc(temp) + 1 };
    }
  })
  .filter(x => x !== undefined);
}
```

En *TypeScript* obtenemos el siguiente error al compilar el programa:

```
Type '({ value: T; next: number; } | undefined)[]' is not assignable
to type 'NextInt<T>[]'.
```

El error indica que no puede asignar un array con una unión de tipos a un array con la interfaz que hemos definido. El problema que encontramos aquí es que—después de ejecutar la función `map`—tenemos un array cuyo tipo es `NextInt<T> | undefined` en lugar de `NextInt<T>`. Por ese motivo

ejecutamos la función `filter` para eliminar todos aquellos valores `undefined` que se encuentren en el array, pero a pesar de ello el tipo unión todavía se mantiene después de aplicar `filter`. Una primera solución sería ejecutar primero `filter` usando la conversión a número para comprobar aquellos elementos en el array para los que la conversión tenga éxito, y después ejecutar `map` usando de nuevo la conversión para construir el resultado final:

```
function getNexts<T>(values: Array<T>): Array<NextInt<T>> {
  return values.filter(x => !isNaN(Number(x)))
    .map(x => ({ value: x, next: Math.trunc(Number(x)) + 1 }));
}
```

Esto nos lleva a ejecutar la función de conversión a número en dos ocasiones, lo cual sería ineficiente e innecesario. Para solventar esto último, otra opción—que nos evita el mensaje de alerta del principio—es ejecutar un `map` adicional para aplicar un *casting* a `NextInt<T>` para cada elemento en el array, pero esto también es innecesario, porque podemos usar un *casting* de tipo para todo el resultado final antes de devolverlo:

```
function getNexts<T>(values: Array<T>): Array<NextInt<T>> {
  return <Array<NextInt<T>>>values.map(...)
    .filter(x => x !== undefined);
}
```

En *Flow* el mensaje de error que nos devuelve la herramienta es similar al de *TypeScript*, pero el problema no se puede solventar con un simple *casting* y por lo tanto el código de la función debe ser modificado siguiendo el patrón de la primera solución que hemos visto para este ejemplo.

Ahora veamos, a continuación, un ejemplo similar en *Python* usando las anotaciones de tipo de *MyPy*:

```
def toInteger(victim:Any) -> Optional[int]:
    try:
        return int(victim)
    except:
        return None

def makeTuple(value:T) -> Optional[Tuple[T, int]]:
    aux = toInteger(value)
    if type(aux) == int:
        return (value, aux + 1)
    else:
        return None
```

```
def getNexts(values:Iterable[T]) -> Iterable[Tuple[T, int]]:
    aux1 = map(makeTuple, values)
    aux2 = list(filter(lambda x: x != None, aux1))
    return aux2
```

Dentro de la función `makeTuple` obtenemos el siguiente error para la expresión `aux + 1`:

```
error: Unsupported operand types for + ("None" and "int")
note: Left operand is of type "Optional[int]"
```

Y dentro de la función `getNexts` obtenemos el siguiente error en la sentencia `return`:

```
error: Argument 1 to "list" has incompatible type
"Iterator[Optional[Tuple[T, int]]]"; expected
"Iterable[Tuple[T, int]]"
```

El primer error nos avisa de que `aux` no encaja en los tipos esperados por el operador suma, dado que el valor `None` no se puede sumar a ningún número. Incluso si usáramos la condición `aux != None` como parte de la sentencia `if`, seguiríamos obteniendo un mensaje de error lanzado por la herramienta. Para poder resolver esta situación, podemos declarar que la variable `aux` tiene tipo `Any`:

```
aux:Any = toInteger(value)
```

Al declarar `aux` de tipo `Any`, *MyPy* registra la variable de programa como una variable dinámica y no comprueba si esta encaja o no en el tipo del operador suma. Después de resolver el primer mensaje de error, el segundo se puede solventar realizando un *casting* de tipo con el resultado final que vamos a devolver:

```
return cast(Iterable[Tuple[T, int]], aux2)
```

También podríamos haber ejecutado otro `map` y realizar un *casting* a `Tuple[T, int]` sobre cada elemento, pero igual que con el ejemplo en *TypeScript*, habría sido ineficiente e innecesario.

En *Erlang*, también podemos escribir una función similar a `getNexts` de la siguiente manera:

```
getNexts(V) ->
  V2 = map(fun(X) ->
       case to_integer(X) of
         undefined -> undefined;
         N -> {X, N + 1}
       end
     end, V),
  filter(fun(X) -> X =/= undefined end, V2).
```

El tipo ([α]) → [{α,integer()} ∪ 'undefined'] se puede derivar en nuestro sistema para la función getNexts, como ocurre con *TypeScript* and *MyPy*. Sin embargo, como no es vacía la intersección entre los tipos [{α,integer()} ∪ 'undefined'] y [{α,integer()}], un sistema basado en *success types* no marcaría como un error si el usuario etiqueta la función getNexts con el tipo ([α]) → [{α,integer()}].

En contraposición, tanto *TypeScript* como *MyPy* dan falsos positivos para programas seguros, porque el objetivo final de estas herramientas es adaptar la disciplina de tipado estático a los lenguajes que analizan. Esto puede forzar al usuario a añadir información de tipo extra y *castings* para encajar las piezas en los tipos esperados para los resultados, o crear una jerarquía compleja de tipos para evitar estos errores de tipo. En el peor escenario, la adopción de este enfoque puede terminar quitando flexibilidad a la hora de desarrollar programas—a menos que el programador elija ignorar las advertencias dadas por la herramienta de comprobación de tipos—y en otros casos el programador será forzado a añadir pistas adicionales para ayudar al mecanismo de comprobación de tipos. Con los *success types* estos ejemplos no habrían sido rechazados, dando la opción al usuario de definir los tipos de entrada para una abstracción determinada si así lo desea, sin tener que preocuparse de los falsos positivos.

Las funciones sobrecargadas—es decir, grupos de funciones que tienen el mismo nombre y el mismo número de argumentos, pero con diferentes tipos para los parámetros de entrada—pueden ser definidas en lenguajes como *Java*, pero en lenguajes de tipado dinámico no se pueden definir. En lenguajes como *Erlang*, *JavaScript* o *Python*, podemos simular este comportamiento en tiempo de ejecución mediante la comprobación dinámica de los tipos de los argumentos, para poder seleccionar una rama de ejecución u otra. En nuestro sistema, y en *Dialyzer*, se pueden usar tipos funcionales sobrecargados en especificaciones de tipo para representar las diferentes ramas de ejecución de la función etiquetada. Sin embargo, *TypeScript* y *MyPy* no toman en consideración los tipos funcionales sobrecargados y solo podremos tener un tipo funcional abarcando todas las ramas de ejecución de la función.

1.6. Objetivos y contribuciones

En esta sección conectamos los apartados anteriores con los objetivos y contribuciones de la tesis, haciendo algunas menciones a los trabajos que se han ido generando en el camino.

Hasta ahora hemos visto los diferentes modelos de sistemas de tipos propuestos para abordar la cuestión de cómo analizar los lenguajes con tipado dinámico. Por un lado hay enfoques como el de los *gradual types*, que están orientados a usar dicho análisis para determinar dónde hay que introducir una comprobación de tipos de cara a garantizar la seguridad de los programas analizados, pero estos enfoques implican perder eficiencia con la sobrecarga de llevar a cabo dichas comprobaciones en tiempo de ejecución. Por otro lado están los enfoques que realizan dichas comprobaciones de tipo mediante herramientas externas para avisar de errores potenciales, pero que muchas veces son tan

conservadores como los sistemas *Hindley-Milner*, llevando a rechazar programas que en ocasiones se comportan correctamente. En estos enfoques se ha intentado incorporar el subtipado a los sistemas, para aumentar la flexibilidad en los análisis, pero al final se gira en torno a imponer la filosofía del tipado estático sobre el dinámico para garantizar la seguridad del código a ejecutar. El ejemplo más paradigmático de este enfoque es *TypeScript*, que introduce programación fuertemente tipada al lenguaje *JavaScript*, con los problemas que ello conlleva para el programador, que ha de pelearse para encajar los tipos y añadir anotaciones innecesarias. Al final, buena parte del problema versa sobre la disputa que hay entre partidarios de los lenguajes de tipado estático o los de tipado dinámico. Cada uno tiene sus ventajas y desventajas. Por ello, no resulta del todo comprensible imponer una metodología de tipos a un lenguaje concebido inicialmente como un lenguaje de tipado dinámico, con una metodología propia, perdiendo la flexibilidad que aporta. Por ello los *success types* se enfocaron en evitar que *Erlang* perdiera su naturaleza dinámica, pero pudiendo aún obtener resultados útiles con los análisis realizados con la herramienta *Dialyzer*.

En la sección 1.4 hemos analizado con más detalle la herramienta *Dialyzer*, incluidas algunas de sus carencias. Esta herramienta ha demostrado obtener resultados útiles en la práctica [101, 22], pero al carecer de tipos polimórficos reales pierde mucha precisión con todas las funciones polimórficas públicas que contiene un módulo en *Erlang*. En [66] se propuso una primera contribución para afrontar esta limitación, donde dado un programa *Erlang* con especificaciones de tipo polimórficas dadas por el usuario, se transformaba dicho programa en uno nuevo tal que *Dialyzer*, cuando se ejecutara sobre el programa transformado, infiriera tipos más precisos para las expresiones que utilizaran dichas funciones polimórficas. Sin embargo, este enfoque está limitado por la estrecha dependencia que tiene con *Dialyzer*. Cualquier cambio realizado en la herramienta podría afectar e incluso invalidar la transformación propuesta. Además, demostrar cualquier resultado teórico implicaría tener que confiar en las propiedades que *Dialyzer* tiene, las cuales no han sido demostradas con suficiente rigurosidad. Esta es una segunda debilidad relevante de *Dialyzer*: la falta de una formalización rigurosa y un marco teórico bien desarrollado sobre el que se pueda justificar la corrección técnica de las propuestas que se cimenten sobre dicha herramienta.

Estos escollos encontrados con *Dialyzer* nos han llevado a tener que plantear un nuevo sistema basado en *success types* para abordar el problema de analizar e inferir tipos polimórficos en el lenguaje *Erlang*. Por ello los objetivos de este trabajo son:

- La construcción de un *sistema de tipos* que, siguiendo el enfoque de *success types*, incorpore polimorfismo y disponga de un marco teórico sólido.
- El desarrollo de una *herramienta de análisis* estático basada en el sistema de tipos.
- La demostración de los *resultados de corrección* pertinentes del sistema de tipos.

Para afrontar la tarea de construir un sistema de tipos basado en los *success types*, es un requisito indispensable disponer de una semántica [98] para el lenguaje *Erlang*, de cara a poder definir también cuál

es la semántica de los tipos que tendrían que sobreaproximar la semántica de las expresiones. Esto se vio reflejado en un primer trabajo [64] donde formalizamos un sistema de tipos monomórfico con sus resultados de corrección. En [67, 65] introdujimos un sistema de tipos polimórficos como siguiente paso hacia nuestro objetivo. Concretamente, establecimos un conjunto de reglas de tipado para derivar *success types polimórficos* en programas escritos en una versión desazucarada de *Erlang*. Además, demostramos que los tipos obtenidos eran correctos con respecto a una semántica adecuada para las expresiones de los programas. En [68] extendimos el sistema de tipos polimórficos previo de diversas maneras. La primera de ellas consiste en añadir la posibilidad de tener tipos funcionales sobrecargados. Esto nos permite obtener resultados más precisos no solo para describir el resultado de una función, sino también para derivar tipos más precisos en las aplicaciones de dichas funciones. Una consecuencia notable es que tratamos las llamadas a funciones que involucran la comprobación de tipos en tiempo de ejecución (tales como `is_integer`, `is_atom`, etcétera) como si fueran llamadas a funciones ordinarias, sin la necesidad de reglas de tipado específicas que reflejen su comportamiento. Otra diferencia, con respecto a los trabajos anteriores, es la forma en que tratamos los tipos no lineales (es decir, los tipos funcionales en los que una variable de tipo aparece más de una vez). La semántica de los tipos y restricciones dada en [67] fue un tanto artificial debido a la necesidad de consolidar la información correspondiente a las diferentes apariciones de una variable de tipo, haciendo uso de operadores de unión propios no estándar que dificultaban el desarrollo técnico del sistema. Por lo que, en [68], adoptamos un enfoque explícito mediante la introducción de nuevos tipos de restricciones sobre los tipos, que nos permitían poder prescindir de los tipos no lineales, para simplificar el desarrollo técnico. Finalmente se pasó a desarrollar reglas de inferencia y un algoritmo para obtener tipos polimórficos sobrecargados. Sobre dicha base se programó un prototipo de herramienta para analizar expresiones escritas en un subconjunto del lenguaje *Erlang*.

También cabe destacar que las contribuciones que este trabajo ha realizado consisten en ofrecer una definición semántica de tipos polimórficos que, combinada con un conjunto de reglas de tipado nos permite, por un lado, tener en cuenta la información polimórfica a la hora de derivar tipos para expresiones y, por otro lado, derivar esquemas de tipos polimórficos sobrecargados para definiciones de funciones. Esto está acompañado a su vez por un algoritmo de inferencia que genera una serie de restricciones en una primera instancia y luego son transformadas para presentar un tipo más legible para usuarios finales. Nuestro sistema de tipos además viene acompañado con sus resultados de corrección, demostrando que en efecto los tipos obtenidos para las expresiones dadas son *success types*.

1.7. Estructura de la tesis

Tras la introducción anterior, la estructura del resto de la tesis es la siguiente:

- En el capítulo 2 vamos a introducir al lector en el lenguaje de programación *Erlang* y presentar

Mini Erlang, el subconjunto del lenguaje escogido para afrontar la investigación, definiendo su sintaxis completa y su semántica.

- En el capítulo 3 presentaremos el sistema para derivar tipos monomórficos que creamos como paso inicial para arrancar nuestro trabajo de investigación, acompañado de algunos ejemplos para estudiar su funcionamiento y de sus resultados de corrección.
- En el capítulo 4 mostraremos el sistema para derivar tipos polimórficos sobrecargados que desarrollamos como evolución del monomórfico, también acompañado de varios ejemplos que ilustran cómo funciona el sistema, así como de sus resultados de corrección.
- En el capítulo 5 detallaremos el algoritmo de inferencia que nos permite obtener tipos polimórficos sobrecargados que pueden ser derivados con el sistema del capítulo anterior, junto a sus resultados de corrección. También mostraremos una serie de ejemplos y tipos obtenidos con la herramienta implementada para analizarlos. Dicha herramienta se encuentra en el siguiente repositorio público: `https://github.com/gorkinovich/meta`
- Finalmente, en el capítulo 6 expondremos las conclusiones sobre los resultados obtenidos y además explicaremos cuáles son las posibles líneas de trabajo futuro que podrían derivar de esta investigación.

Capítulo 2

Preliminares

A lo largo del siguiente capítulo introduciremos el lenguaje de programación con el que hemos estado trabajando, que consiste en un subconjunto del lenguaje *Erlang*, al que llamamos *Mini Erlang*. En la sección 2.1 haremos una introducción al lenguaje *Erlang* para aquellos que no estén familiarizados con el mismo. En la sección 2.2 presentaremos la sintaxis de *Mini Erlang*, que es una versión simplificada de *Core Erlang*, el lenguaje intermedio al que todo programa *Erlang* es convertido durante su compilación. Por último en la sección 2.3 presentaremos la semántica de *Mini Erlang*.

2.1. Lenguaje *Erlang*

El lenguaje *Erlang* [9] es un lenguaje de programación funcional, que conforma el núcleo de la plataforma *Erlang/OTP*, creado en la empresa de telecomunicaciones *Ericsson* en los años ochenta del siglo XX por *Armstrong*, *Virding* y *Williams*. Esta plataforma la componen el entorno de ejecución de *Erlang* (una máquina virtual que ejecuta código *bytecode*) y un conjunto de bibliotecas. Se trata de un lenguaje declarativo que fue diseñado para afrontar los retos de desarrollar aplicaciones distribuidas en tiempo real [10], tolerantes a fallos, con alta disponibilidad y que permitan la capacidad de cambiar código en caliente[1] sin parar los sistemas. Entre sus características destacables encontramos: computación concurrente, tipado dinámico, encaje de patrones para estructuras de datos, variables inmutables[2] y evaluación impaciente.[3]

Es importante señalar que los programas de *Erlang* suelen estar orientados a tener uno o más procesos en ejecución. Esto es gracias al bajo coste que supone lanzar procesos y gestionarlos en su entorno de

[1] Cambiar código en caliente permite a los desarrolladores cambiar el código de la aplicación mientras se está ejecutando.
[2] También conocidas como variables de asignación única; estas no podrán cambiar de valor durante la ejecución de aquel que le ha sido asignado.
[3] Al contrario que la evaluación perezosa, que se ejecuta bajo demanda, la evaluación impaciente ha de calcular los valores completos que se van a pasar como valores a la llamada de una función.

ejecución. Estos procesos tienen mecanismos para comunicarse entre sí de forma sencilla. El lenguaje dispone también de mecanismos para vigilar procesos desde otro proceso, pudiendo volver a lanzar aquel proceso que haya caído para continuar prestando sus servicios al resto.

Gracias a su filosofía de diseño pragmática, con los años *Erlang* ha ido recibiendo una mayor atención por parte de la industria y el mundo académico, debido a su fortaleza a la hora de producir sistemas escalables y robustos, que son fáciles de construir y mantener. Es por ello que encontramos diversas tecnologías siendo desarrolladas con *Erlang* [20, 18], tales como: software de bróker de mensajería (*RabbitMQ* y *VerneMQ*), servidores de mensajería instantánea con XMPP (*ejabberd*) o bases de datos distribuidas (*Riak*, *Apache CouchDB*, *Amazon SimpleDB*), incluyendo también el servidor de mensajes de *Whatsapp*.

Todo programa *Erlang* está compuesto por una serie de *módulos* con un nombre asignado, que contienen *funciones* que pueden ser públicas o no, definiendo así la interfaz de uso del módulo. Por ejemplo:

```
-module(numbers).
-export([one/0, two/0]).

zero() -> 0.
one() -> zero() + 1.
two() -> one() + 1.
```

El ejemplo contiene un módulo llamado `numbers` que tiene tres funciones de cero parámetros, de las cuales solo `one` y `two` son accesibles desde el exterior. La función `zero` devuelve el valor cero, `one` el valor uno y `two` el valor dos. Podemos apreciar en `one` y `two`, que cada una de estas funciones utiliza la función declarada previamente, aplicándola y sumando una unidad al resultado con el operador suma.

2.1.1. Valores literales

El lenguaje *Erlang* dispone de los siguientes tipos de valores literales: *números enteros*, *números de coma flotante* y *átomos*. Los *átomos* consisten en cadenas alfanuméricas que tienen valor propio por sí solas; en este sentido pueden considerarse constantes simbólicas.

La forma de definir literales numéricos es similar a la de otros lenguajes conocidos. Por ello nos vamos a centrar en cómo definimos los *átomos*. Para definir uno debemos empezar con una letra minúscula del alfabeto inglés, seguido de más letras minúsculas o mayúsculas, números, guion bajo (`_`) y el símbolo arroba (`@`). También, alternativamente, podemos usar las comillas simples para utilizar otros caracteres especiales o poder empezar por una letra mayúscula si fuera necesario. Por ejemplo:

```
plastic_love@mariya_takeuchi
'Ride on Time (Tatsurô Yamashita)'
```

El lenguaje también dispone de *cadenas de texto*, que escribimos usando comillas dobles, de forma similar a otros lenguajes conocidos. Estos valores literales son azúcar sintáctico que se traducen en listas de números enteros (sección 2.1.3) que representan dichas cadenas de texto en los programas.

La *lista vacía*, representada con `[]`, también se trata de un valor literal especial que se utiliza como valor de terminación en las *listas* del lenguaje, explicadas en la sección 2.1.3.

2.1.2. Variables

Las variables en el lenguaje *Erlang* representan valores inmutables asignados como el resultado de una expresión o como los parámetros de entrada a una función. Que sean inmutables quiere decir que, a diferencia de lenguajes como C++ o Java, solo les podemos asignar valor en su inicialización. Por ejemplo, no se permite lo siguiente:

```
X = 0, X = X + 1.
```

No se permite porque, una vez se inicializa la variable, el operador igual en la segunda expresión pasa a cumplir con otro propósito en el lenguaje que es el encajar patrones, en vez de enlazar una expresión con una variable como ocurre con la primera expresión. En este caso el encaje de patrones provoca un fallo de ejecución al ser valores distintos los que hay a cada lado del operador. En la sección 2.1.4 veremos más información sobre el encaje de patrones.

Para dar nombre a una variable usaremos caracteres alfanuméricos (`A-Z`, `a-z`, `0-9`), guiones bajos (`_`) y el carácter arroba (`@`); pero estos nombres identificadores han de empezar por una letra mayúscula o guion bajo[4] para que sean válidos.

2.1.3. Estructuras de datos

El lenguaje permite definir diferentes estructuras de datos con las que trabajar. Entre ellas encontramos: *listas*, *tuplas*, *mapas* (o diccionarios) y *cadenas de bits*. Existen otras estructuras a nivel de implementación como pueden ser arrays o conjuntos, pero solo son accesibles usando la biblioteca estándar de *Erlang* donde se accede a la implementación nativa del lenguaje para crear instancias y poder trabajar con estas estructuras de datos especiales. Para este trabajo nos vamos a centrar únicamente en las *listas* y las *tuplas*.

Las *listas* consisten en secuencias de valores en un orden determinado. Las listas se crean con el *constructor de listas* `[_|_]`, donde la expresión en la parte izquierda es el valor de la *cabeza* de la lista y en la derecha está la *cola* de la misma. Cuando la cola no es de la forma `[_|_]` se denomina *terminación*. Es habitual que la terminación sea la lista vacía (`[]`), pero puede ser cualquier otro valor al no impedirlo la semántica de *Erlang*. Algunos ejemplos de listas son:

[4] Por convención en *Erlang* los nombres que empiezan por guion bajo son los de variables cuyo contenido no interesa.

```
[a|[b|[c|[]]]]
[1|[2|[3|[4|[5|[]]]]]]
[one|[23|[2.3|ok]]]
```

El primer ejemplo es una lista de átomos que termina con la lista vacía. El segundo es una lista de números que también termina con la lista vacía. El tercero es una lista heterogénea que termina con el átomo `ok`.

Para construir las listas en los ejemplos anidamos los constructores de listas. Como esta forma de construir listas es farragosa, el lenguaje dispone de azúcar sintáctico que permite alternativas mejores para hacer lo mismo:

$$[x_1,\ldots,x_n \mid x] = [x_1 \mid \cdots [x_n \mid x] \cdots]$$
$$[x_1,\ldots,x_n] = [x_1,\ldots,x_n \mid []]$$

Con esto, los ejemplos anteriores pasaríamos a poder escribirlos de la siguiente manera:

```
[a,b,c]
[1,2,3,4,5]
[one,23,2.3|ok]
```

Las *tuplas* son secuencias de valores con una longitud fija. El contenido de las tuplas está delimitado por el símbolo de las llaves y separado por comas. Algunos ejemplos de tuplas son:

```
{left, right}
{one, 23, 2.34}
```

También podemos escribir una tupla vacía con `{}` si así lo quisiéramos. En principio podríamos hacer con listas lo que hacemos con tuplas, pero cuando queremos usar una colección de datos con un tamaño fijo—y no variable—resulta más cómodo usar tuplas.

Tanto las *listas* como las *tuplas* se pueden anidar. Por ejemplo:

```
{[{1983, may, 24}, ongaku], 3.42}
[[2, 11], {23, 31}]
```

2.1.4. Encaje de patrones

Al igual que podemos construir estructuras de datos para componer información compleja, podemos deconstruirlas para obtener parte de la información contenida en ellas. Para hacer esto, al igual que en otros lenguajes funcionales, usaremos el encaje de patrones. Para ello usamos el operador = que sirve para enlazar variables y patrones con el valor resultante de una expresión:

```
Value = {numbers, [1, 2, 3]},
{_, [First | _]} = Value.
```

En este ejemplo se construye un valor que es una tupla, cuya primera componente es el átomo `numbers` y la segunda una lista con tres números. A continuación, queremos obtener de dicha estructura de datos la cabeza de la lista. Para ello construimos un patrón donde indicamos que hay una tupla de dos componentes y que estamos interesados en asignar a la variable `First` el valor que se encuentre en la cabeza de la lista que hay en la segunda componente de la tupla. Como resultado de estos dos encajes de patrones, la variable `First` habrá adquirido el valor 1.

2.1.5. Operadores

El lenguaje dispone de operadores de comparación, aritméticos, lógicos y para manipular listas. La mayor parte de los operadores son similares a sus equivalentes en otros lenguajes. Los operadores de comparación y los lógicos devuelven como resultado los átomos `true` y `false`, que ejercen como los valores *booleanos* del lenguaje. De los operadores para manipular listas, el operador de concatenación (++) devuelve una lista donde el lado derecho pasa a ser la terminación de la lista en la izquierda, con lo que es perfectamente legal que el lado derecho sea cualquier valor posible.

2.1.6. Estructuras de control

De las estructuras de control del lenguaje *Erlang*, solo dos de ellas son relevantes para este trabajo, que son `case` y `receive`. Aunque también disponemos de sentencias de control para gestionar excepciones lanzadas en tiempo de ejecución en caso de error, la filosofía de *Erlang* está basada en el eslogan *"let it crash"* [9], que consiste en dejar que los procesos se "estrellen" si se produce algún error durante su ejecución y se lance un nuevo proceso para sustituir al que acaba de morir. Es por esta filosofía peculiar que no se recomienda un uso profuso de las excepciones, pasando estas a un segundo plano en nuestra investigación.

La estructura `case` recibe un valor llamado *discriminante* y selecciona de una serie de *cláusulas* cuál es la primera en la que el valor puede encajar, para así ejecutar el cuerpo de la cláusula. Esto permite ramificar la ejecución de las funciones. A continuación mostramos un ejemplo típico de su uso:

```
case L of
    [] -> nothing;
    [X|_] -> {something, X}
end.
```
(2.1)

Aquí se espera que la variable `L` sea algún tipo de lista. Si es una lista vacía se devuelve el átomo `nothing`, pero si no está vacía se devuelve una tupla con el átomo `something` y la cabeza de la misma. Si `L` no fuera una lista, al no haber ninguna cláusula disponible, el programa lanzaría una excepción.

En las *cláusulas* tenemos un patrón de encaje que liga las variables que aparecen en el mismo con las subestructuras correspondientes en el discriminante del `case`. Cada patrón puede ir seguido de una *guarda* donde indicar las condiciones lógicas que se deben cumplir para poder ejecutar las expresiones en el cuerpo de la cláusula. El orden en el que definimos las cláusulas es importante, ya que se usa dicho orden para seleccionarlas, de modo que cuando se selecciona la primera que permite el encaje del discriminante con el patrón, las siguientes donde el discriminante también encaja con el patrón son ignoradas. Por ejemplo:

```
case X of
    N when 0 < N, N =< 10 -> gaff;
    N when N < 0; 10 < N -> deckard;
    0 -> rachael;
    _ -> tyrell
end.
```
(2.2)

El ejemplo tiene cuatro cláusulas donde la última recoge cualquier posible valor con la *variable anónima*.[5] La primera cláusula recibe un valor en `N` y tiene que cumplir como condición que sea mayor que cero y menor o igual que diez. Para ello utilizamos la coma a modo de operador `and`. La segunda cláusula recibe un valor en otra `N`—distinta a la anterior por estar en un ámbito diferente—para comprobar que sea menor que cero o mayor que diez, usando el punto y coma como operador `or`. Finalmente la tercera comprueba si el valor de `X` es cero.

Otro detalle sobre las *guardas* es que se permite solo el uso de operadores y un conjunto reducido de *funciones predefinidas* en el lenguaje. En particular, algunas de estas funciones sirven para comprobar si un valor es de un tipo concreto, como por ejemplo `is_integer` que devuelve si un valor es entero o no. Estas funciones predefinidas permitidas están recogidas en la documentación oficial del lenguaje. También cabe señalar que podremos usar guardas en las definiciones de las funciones, como indicamos en la sección 2.1.7.

La otra estructura fundamental es `receive`, que contiene una serie de *cláusulas* de las que se seleccionará la primera que pueda encajar con el valor actual en la cabecera de la lista de mensajes recibidos por parte del proceso. Esto permite ramificar la ejecución a la hora de recibir información desde otros procesos. Por ejemplo:

```
receive
    jonathan -> dio;
    X -> X
end.
```
(2.3)

En el ejemplo hay dos cláusulas, donde la primera devuelve el átomo `dio` si el proceso que ejecuta el `receive` recibe un mensaje con el átomo `jonathan` por parte de otro proceso. La segunda cláusula devolverá cualquier valor que reciba sin más. Igual que ocurre con `case`, el orden en el que declaramos

[5] En este caso también podemos conseguir el mismo efecto con una variable fresca cualquiera que no se usará.

las cláusulas es importante, porque si se recibe `jonathan` se ejecutará la primera clásula solamente, a pesar de que el valor puede encajar en la segunda. Hay que tener en cuenta que `receive` bloquea la ejecución a la espera de que se reciba algún valor en la cola de mensajes del proceso, lo cual puede suponer un problema en ocasiones y es por ello que la estructura dispone de la cláusula `after`:

```
receive
    ymo -> rydeen;
    casiopea -> asayake
after
    1000 -> computer_love
end.
```
(2.4)

En el ejemplo hay una sección llamada `after` donde el lado izquierdo tiene que ser un número entero (o una variable que contenga un valor entero) que representa el número de milisegundos que el proceso tendrá como tiempo límite mientras espera la llegada de un mensaje que encaje con los patrones anteriores en el `receive`. Si se supera el tiempo límite de espera, se ejecutará el cuerpo del `after`. Podemos también usar en el lado izquierdo el átomo `infinity`, que sería lo mismo que no poner la sección `after`. Por otro lado, si ponemos como valor de espera un cero, si hubiera un valor en la cola de mensajes, se procesará para ver si este puede encajar con alguna de las cláusulas; de lo contrario se saltará inmediatamente a la sección `after`.

2.1.7. Funciones

Al principio de la sección 2.1, explicamos que los *módulos* en *Erlang* contienen *funciones*. Para definir una *función* tenemos que darle un nombre con un átomo, definir una serie de patrones de encaje como parámetros, una guarda como condición y finalmente un cuerpo de la función con las expresiones a evaluar. Además, de forma similar a un `case`, podemos tener diferentes cláusulas para una misma función, organizando las diferentes ramas de ejecución de la misma. Por ejemplo:

```
test() ->
    foo([1,2,3]).

foo(L) when is_list(L) ->
    lists:map(fun(X) -> X + 1 end, L);
foo(_) ->
    bukowski.
```
(2.5)

El ejemplo ilustra diferentes aspectos de las funciones en el lenguaje. Primero, hay una función `test` que llama a una función llamada `foo` con una lista de números. Segundo, hay una función `foo` que tiene dos ramas de ejecución. En la primera rama se recibe un valor en `L` al que se pone como con-

dición que se trate de una lista con la *función predefinida*[6] `is_list`, para luego llamar a la función `map` que está dentro del módulo `lists` pasándole como parámetros una *λ-expresión* y la lista `L`. La *λ*-expresión del ejemplo recibe un elemento en `X`, le suma el valor uno y lo devuelve. La segunda rama de `foo` se ejecutará cuando el valor pasado no encaje con la primera cláusula, y simplemente devolverá un valor literal.

En el ejemplo vemos cómo podemos llamar a las funciones, ya sea una dentro del mismo módulo, una función predefinida del lenguaje o una función en otro módulo exterior. Además de las anteriores maneras podemos enlazar una variable de programa con una función, para luego poder llamar a la función que representa. Para hacer referencia a las funciones como valores usaremos `fun`, seguido del módulo donde está la función, su nombre y su aridad.[7] Si queremos hacer referencia a una función del mismo módulo, podemos obviar el nombre del módulo. Por ejemplo:

```
id(X) -> X.

test(L) ->                                    (2.6)
    F = fun lists:map/2,
    F(fun id/1, L).
```

El ejemplo tiene la función identidad, seguido de una función de prueba que asigna a `F` el valor de la función `map` (que tiene dos parámetros y está en el módulo `lists`), para realizar una llamada a función con `F`. Cabe señalar que la implementación de `map` recibe como primer parámetro una función y en este último ejemplo estamos pasando la función identidad del módulo actual.

2.2. Sintaxis de *Mini Erlang*

La semántica del lenguaje *Erlang* en ocasiones no es sencilla. Por ello, el equipo encargado de desarrollarlo se vio en la necesidad de definir un lenguaje intermedio llamado *Core Erlang* [15] que tuviera una semántica clara y simple con el que poder trabajar a la hora de analizar programas. Cualquier programa escrito en *Erlang* se puede traducir a *Core Erlang* y viceversa. Por ello, para trabajar en nuestra investigación partimos de la base de usar un subconjunto del lenguaje *Core Erlang* al que hemos llamado *Mini Erlang* y cuya sintaxis puede ser vista en la figura 2.1.

El motivo principal que llevó a tomar un subconjunto de *Core Erlang* fue tener un lenguaje que nos permitiera simplificar las reglas de tipado en nuestro sistema sin perder capacidad de expresividad con el lenguaje. Hay que señalar que el lenguaje utilizado en esta tesis coincide con el utilizado en el sistema de tipos polimófico desarrollado [67, 68], pero que las diferencias con versiones anteriores [64] a nivel de sintaxis son apenas perceptibles.

[6] Las funciones predefinidas están dentro del módulo `erlang`, sin embargo el lenguaje nos permite obviar hacer referencia a dicho módulo para facilitar su uso.

[7] La aridad es el número de argumentos que tiene una función.

$$
\begin{array}{ll}
var ::= \text{VARIABLENAME} & exp ::= var \mid lit \mid fun \mid [exp_1 \mid exp_2] \mid \{\overline{exp_i}^{\,n}\} \\
lit ::= \text{ATOM} \mid \text{INTEGER} \mid \text{FLOAT} \mid [] & \quad \mid var(\overline{var_i}^{\,n}) \mid \textbf{let}\ var = exp_1\ \textbf{in}\ exp_2 \\
pat ::= var \mid lit \mid [pat_1 \mid pat_2] \mid \{\overline{pat_i}^{\,n}\} & \quad \mid \textbf{letrec}\ \overline{var_i = fun_i}^{\,n}\ \textbf{in}\ exp \\
cls ::= pat\ \textbf{when}\ exp_1 \rightarrow exp_2 & \quad \mid \textbf{case}\ var\ \textbf{of}\ \overline{cls_i}^{\,n}\ \textbf{end} \\
fun ::= \textbf{fun}(\overline{var_i}^{\,n}) \rightarrow exp & \quad \mid \textbf{receive}\ \overline{cls_i}^{\,n}\ \textbf{after}\ var \rightarrow exp
\end{array}
$$

Figura 2.1: Sintaxis de *Mini Erlang*, el subconjunto usado de Core Erlang

En *Mini Erlang* los programas los podemos expresar como un conjunto de funciones recursivas que se encuentran en una expresión **letrec**, donde la expresión que iría en la parte **in** sería la llamada a la función principal del programa. Esto, aplicado al ejemplo del módulo `numbers` de la sección 2.1, quedaría del siguiente modo:

```
letrec Zero = fun() → 0
       One = fun() → let K = 1 in
                     let N = Zero() in N + K
       Two = fun() → let K = 1 in
                     let N = One() in N + K
in 'ok'
```

Tal como muestra la figura 2.1, disponemos de *valores literales*, *variables*, *listas*, *tuplas*, *λ-expresiones*, la sentencia **let** para introducir nuevas variables, la sentencia **letrec** para introducir nuevas funciones recursivas, la sentencia **case** para ramificar la ejecución, la sentencia **receive** para ramificar la ejecución cuando se recibe un mensaje en el proceso actual, y llamadas a función. En la cláusula **after**—de **receive**—se utiliza una variable para almacenar el tiempo de espera límite, pudiendo ser un valor numérico entero para indicar el número de milisegundos de la espera o el átomo `'infinity'` si no queremos ejecutar la cláusula.

Los valores literales que tiene el lenguaje son: *átomos*, *números enteros*, *números de coma flotante* y la *lista vacía*. Aunque *Core Erlang* incluye cadenas de texto como literales, han sido omitidas ya que esencialmente son listas de números enteros. Los valores literales permitidos los podemos definir igual que en *Erlang* con una salvedad: los átomos no podrán definirse sin estar delimitados por comillas simples.

Los nombres de variables se rigen por las mismas reglas que *Erlang*, pero para poder hacer encaje de

patrones necesitamos usar la estructura de control **case**. Tomando el ejemplo de la sección 2.1.4:

```
let Value = {'numbers',[1|[2|[3|[]]]]}
in case Value of
    {_0,[First|_1]} when 'true' → First
end
```

El ejemplo usa la variable `Value` y luego con el **case**, que solo tiene una cláusula, obtenemos con `First` el fragmento deseado.[8]

En cuanto a estructuras de datos, *Mini Erlang* tiene *listas* y *tuplas*; que son idénticas a las de *Erlang*, con la salvedad de que con las listas solo podremos utilizar el *constructor de listas* para su construcción. Ya vimos en los ejemplos anteriores de la sección 2.1.3 cómo definíamos una lista sin el azúcar sintáctico. No obstante, por motivos de legibilidad, utilizaremos las sintaxis azucarada en algunos de los ejemplos. Una peculiaridad de *Core Erlang* es la incorporación de las llamadas *secuencias*, que son definidas usando <.^{n}.>. Como estas son muy similares a las tuplas y no son más que un mecanismo de optimización en la implementación de *Erlang*, en *Mini Erlang* usaremos tuplas en lugar de secuencias al ser prácticamente idénticas desde un punto de vista de los tipos, pues lo único que varía es el nombre de la estructura.

Otra diferencia relevante entre *Mini Erlang* y *Core Erlang* es que unificamos los tres sistemas para invocar funciones que tiene *Core Erlang* en uno solo. Esto se hace porque—desde el punto de vista del tipado—las tres son equivalentes, así que dando un trato uniforme a las llamadas a función eliminamos ruido de las reglas en el sistema de tipos.

En *Mini Erlang* podemos observar que, tanto en los parámetros de llamada a una función como en el discriminante del **case**, solo están permitidas variables de programa de cara a simplificar las reglas de tipado. Esto permite asociar información de tipos al discriminante cuando vayamos a tipar las diferentes ramas de ejecución del programa analizado. También asumiremos—dentro del contexto de las aplicaciones de función—que tanto la función aplicada como los argumentos de la misma van a ser variables de programa, pues con ello sus tipos podrán ser almacenados dentro de un entorno de tipos cuando se analice la aplicación de dicha función. No perderemos expresividad al introducir esta limitación en la sintaxis, ya que podemos sustituir las expresiones complejas usando sentencias **let**, que introducirán variables de programa en las que se almacenarán los valores de las subexpresiones que componen la expresión compleja inicial.

Tomando algunos de los ejemplos de la sección 2.1.6, para el ejemplo 2.1 de la expresión **case**, su

[8] *Core Erlang* requiere que siempre haya una cláusula a ejecutar en tiempo de ejecución, por ello—si no tiene una cláusula por defecto para todo valor—generará una cláusula sencilla que llama a una función primitiva del intérprete para lanzar un error en tiempo de ejecución. Al no ser el tratamiento de excepciones un objetivo principal de la investigación, en *Mini Erlang* prescindimos de este comportamiento.

equivalente en *Mini Erlang* sería el siguiente:

```
case L of
    [] when 'true' → 'nothing'
    [X|_0] when 'true' → {'something',X}
end
```

Entre las diferencias apreciables, la guarda de las cláusulas siempre es obligatoria en el lenguaje. También se renombran las variables anónimas al pasar de *Erlang* a *Core Erlang*, usando una variable fresca para cada aparición de las mismas. Luego para la expresión **receive**, tomando el ejemplo 2.3, su equivalente en *Mini Erlang* sería por ejemplo:

```
receive
    'jonathan' when 'true' → 'dio'
    X when 'true' → X
after
    'infinity' → 'true'
```

Hay que señalar que en *Core Erlang* la cláusula **after** es obligatoria. Por ello, cuando no declaremos en *Erlang*, se genera la que mostramos en el ejemplo anterior. Cuando usemos el valor `'infinity'` en la cláusula **after**, el resultado de la expresión del cuerpo de la cláusula es irrelevante porque la semántica no devolverá nunca ese valor. Al traducir a *Mini Erlang*, siguiendo el modelo de *Core Erlang*, cuando no exista una cláusula **after** también se generará y su cuerpo será el átomo `'true'`, aunque como ya hemos señalado sería irrelevante el valor escogido para el cuerpo del **after**.

Para declarar *funciones*—como vimos en los ejemplos de la sección 2.1.7—se usa en *Mini Erlang* una combinación de λ-expresiones con sentencias **case**, sin olvidar las sentencias **letrec** o **let** para dar nombre a la función. Tomando el ejemplo 2.5 de la función `foo`, su versión en *Mini Erlang* sería:

```
letrec Foo = fun(L) →
    case L of
        V when is_list(V) →
            let _1 = fun(X) → let _0 = 1 in X + _0
            in lists:map(_1,L)
        _2 when 'true' → 'bukowski'
    end
in let _3 = [1,2,3] in Foo(_3)
```

Como podemos ver, en este y los sucesivos ejemplos de la tesis, usaremos la notación de guion bajo más un número como nombre de las variables frescas que se van a generar al convertir un programa de *Erlang* a *Mini Erlang*. Otro detalle—importante para la comprensión de los ejemplos—es que si bien la sintaxis requiere el uso de variables para almacenar las funciones que vamos a utilizar, para fa-

cilitar el entendimiento de los ejemplos—al usar funciones predefinidas o de la biblioteca estándar de *Erlang*—se utilizarán los nombres identificadores de estas funciones directamente como si se trataran de variables en *Mini Erlang*, como es el caso de `is_list` en el ejemplo.

En cuanto a las expresiones de *Erlang*—e incluso *Core Erlang*—que son azúcar sintáctico, vamos a asumir en este trabajo que el código que vamos a recibir siempre estará sin dicho azúcar, aunque en algunos de los ejemplos de la tesis hagamos uso del mismo. Por ejemplo, *Core Erlang* tiene **do** como sentencia de control, cuya traducción final consiste en una expresión **let**, que sería la que recibiríamos al trabajar con *Mini Erlang*.

Por último, en este trabajo decidimos ignorar el tratamiento de excepciones, por lo que en *Mini Erlang* no hay disponible ninguna estructura de control para manejarlas. Como ya mencionamos, en la sección 2.1.6, su uso no está recomendado y por ello escogimos centrarnos en las sentencias de control más relevantes del lenguaje, dejando para trabajo futuro el tratamiento de excepciones.

2.3. Semántica de *Mini Erlang*

En trabajos previos [66, 67, 68] y en esta tesis definimos la *semántica* de una *expresión cerrada* como un subconjunto de **DVal**, donde **DVal** representa el conjunto de todos los posibles valores que podemos expresar con *Mini Erlang*. Podemos encontrar una definición formal de **DVal** en [66].

Las *funciones* en **DVal** están representadas como *grafos*, que consisten en un conjunto de pares con la siguiente forma $((\overline{args}), value)$, en donde cada par relaciona una tupla de valores $(\overline{args})$ de entrada con el valor final del resultado. Además, debido a la naturaleza indeterminista de la *concurrencia* en *Erlang*, una misma tupla de argumentos $(\overline{args})$ puede estar relacionada con más de un resultado dentro del grafo de la función, por lo que la *semántica* de una *función* es una relación matemática más que una función en términos estrictos.

Para representar las *estructuras de datos* dentro de **DVal** también utilizaremos tuplas con la forma $(ctor, \overline{args})$, donde *ctor* es el constructor de la estructura y $\overline{args}$ la secuencia de valores tomados por el constructor. En *Mini Erlang* los constructores disponibles son:

- $\{\cdot^{n}\cdot\}$ para tuplas de *Erlang*, donde n es el número de elementos que tiene y cuyos valores finales serán $\overline{args}$.
- `[_|_]` para listas de *Erlang*, donde el primer valor de $\overline{args}$ es el valor de la *cabeza* de la lista y el segundo la *cola* de la misma. Además por comodidad usaremos la notación $(\texttt{[\_|\_]}, v_1, \ldots, v_{n-1}, v_n)$ y $\texttt{[}v_1, \ldots, v_{n-1} \texttt{|} v_n\texttt{]}$ para indicar $n-1$ constructores de lista anidados, con v_n como continuación.

La semántica de una expresión cerrada la podemos definir como el conjunto de valores a los que podemos evaluar dicha expresión. No obstante, para dar semántica a expresiones que contienen variables libres, necesitamos tener en consideración *sustituciones* que den valor a estas variables. Por

ello definimos una sustitución θ como una función total **Var** → **DVal**, donde **Var** es el conjunto de todas las variables de programa posibles. La notación $e\theta$ representa la expresión cerrada resultante de aplicar la sustitución θ a la expresión e. Para denotar el conjunto de todas las sustituciones usaremos **Subst**. Usaremos la notación $[]$ para expresar la sustitución que asigna el valor por defecto a todas las variables. Usaremos el número cero como valor por defecto en las sustituciones, pero cualquier otro valor dentro de **DVal** nos hubiera podido servir para el mismo propósito. Por otro lado, la notación $[x_1/v_1,\ldots,x_n/v_n]$ será usada para representar aquella sustitución que asigna el valor v_i a la variable de programa x_i y cero al resto de variables. Usaremos también la notación $\theta[x_1/v_1,\ldots,x_n/v_n]$ para denotar una θ' tal que:

$$\theta'(x) = \begin{cases} v_i & \text{si } x = x_i \\ \theta(x) & \text{en otro caso} \end{cases}$$

La semántica $\mathscr{E}[\![e]\!]$ de una expresión e está definida como una relación $\mathscr{E}[\![e]\!] \subseteq \textbf{Subst} \times \textbf{DVal}$. Esto quiere decir que si $(\theta, v) \in \mathscr{E}[\![e]\!]$ entonces v es uno de los posibles valores a los que la expresión cerrada $e\theta$ puede ser reducida. La definición completa de $\mathscr{E}[\![e]\!]$ está recogida en la figura 2.2.

La semántica de un valor literal c devuelve tuplas (θ, c) para cualquier valor de θ, con lo que el resultado de evaluar una constante c es—precisamente—c, para cualquier sustitución θ. Con una variable de programa x, la semántica es parecida a la de los literales pero el valor a devolver en la segunda componente es el valor de x dentro de θ.

Es interesante observar cómo se realiza la selección de las sustituciones entre las diferentes partes de una expresión, para agrupar todos aquellos valores que contengan una sustitución determinada común a dichas subexpresiones. Esta selección la podemos ver con claridad en la semántica de las estructuras de datos. Por ejemplo, con la expresión `{X,Y}` cuando se devuelve el valor `{1,2}`, deducimos que la semántica devolverá para ese resultado sustituciones θ donde $\theta(\texttt{X}) = 1$ y $\theta(\texttt{Y}) = 2$.

La semántica de las λ-expresiones devuelve el grafo explicado al principio en la sección y la sustitución devuelta—junto al valor que será el resultado final de la tupla del grafo—tiene restringidos los valores en las variables de entrada, que han de ser iguales a los argumentos de la función. En el caso de las aplicaciones de función buscaremos dentro del grafo de la función asignada a la variable f, aquella tupla cuyos argumentos correspondan con la secuencia de valores obtenidos en la sustitución para las variables x_i utilizadas en la aplicación. Para la expresión **let**, por un lado, tenemos la expresión e_1 que va a ser enlazada con la variable x, con lo que para poder evaluar la expresión interior e_2 tenemos que considerar solo las sustituciones θ que liguen v_1 a x.

Antes de abordar las expresiones **case** y **receive**, vamos a explicar la función *matches*. Esta función recibe una sustitución θ, un valor v y una cláusula p **when** $e_g \to e$, para darnos un conjunto de sustituciones donde las variables de programa $\overline{x_i}$—que encontraremos dentro de la expresión patrón p—están asignadas a ciertos valores $\overline{v_i}$, tales que una nueva sustitución $\theta[\overline{x_i/v_i}]$ acompañada del valor v existiría en la semántica del patrón p y esa misma sustitución $\theta[\overline{x_i/v_i}]$ acompañada del valor `'true'` existiría en la semántica de la guarda e_g. Esto significa que la función devuelve las sustitucio-

$$\mathscr{E}[\![c]\!] = \{(\theta, c) \mid \theta \in \mathbf{Subst}\}$$

$$\mathscr{E}[\![x]\!] = \{(\theta, \theta(x)) \mid \theta \in \mathbf{Subst}\}$$

$$\mathscr{E}[\![\{\overline{e_i}^n\}]\!] = \left\{\left(\theta, \left(\{\cdot^n_{\cdot}\}, \overline{v_i}^n\right)\right) \mid \forall i \in \{1..n\} : (\theta, v_i) \in \mathscr{E}[\![e_i]\!]\right\}$$

$$\mathscr{E}[\![[e_1 \mid e_2]]\!] = \{(\theta, ([\_\mid\_], v_1, v_2)) \mid (\theta, v_1) \in \mathscr{E}[\![e_1]\!], (\theta, v_2) \in \mathscr{E}[\![e_2]\!]\}$$

$$\mathscr{E}[\![\mathbf{fun}(\overline{x_i}^n) \to e]\!] = \left\{\left(\theta, \left\{\left(\left(\overline{v_i}^n\right), v\right) \mid \left(\theta\left[\overline{x_i / v_i}^n\right], v\right) \in \mathscr{E}[\![e]\!]\right\}\right) \mid \theta \in \mathbf{Subst}\right\}$$

$$\mathscr{E}[\![f(\overline{x_i}^n)]\!] = \left\{(\theta, v) \mid \theta \in \mathbf{Subst}, \left(\left(\overline{\theta(x_i)}^n\right), v\right) \in \theta(f)\right\}$$

$$\mathscr{E}[\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!] = \{(\theta, v) \mid (\theta, v_1) \in \mathscr{E}[\![e_1]\!], (\theta[x_1 / v_1], v) \in \mathscr{E}[\![e_2]\!]\}$$

$$\mathscr{E}\left[\!\left[\mathbf{case}\ x\ \mathbf{of}\ \overline{cls_i}^n\ \mathbf{end}\right]\!\right] = \bigcup_{i=1}^{n} \{(\theta, v) \mid \theta \in \mathbf{Subst}, \theta' \in matches(\theta, \theta(x), cls_i), (\theta', v) \in \mathscr{E}[\![e'_i]\!], (\forall k < i : matches(\theta, \theta(x), cls_k) = \emptyset)\}$$

donde $\forall i \in \{1..n\} : cls_i = (p_i\ \mathbf{when}\ e_i \to e'_i)$

$$\mathscr{E}\left[\!\left[\mathbf{receive}\ \overline{cls_i}^n\ \mathbf{after}\ x_t \to e\right]\!\right] = \bigcup_{i=1}^{n} \{(\theta, v) \mid \theta \in \mathbf{Subst}, \theta(x_t) \in \texttt{integer()} \cup \{\texttt{'infinity'}\}, v' \in \mathbf{DVal}, \theta' \in matches(\theta, v', cls_i), (\theta', v) \in \mathscr{E}[\![e'_i]\!], (\forall k < i : matches(\theta, v', cls_k) = \emptyset)\} \cup \{(\theta, v) \mid (\theta, v) \in \mathscr{E}[\![e]\!], \theta(x_t) \in \texttt{integer()}\}$$

donde $\forall i \in \{1..n\} : cls_i = (p_i\ \mathbf{when}\ e_i \to e'_i)$

$$\mathscr{E}[\![\mathbf{letrec}\ \overline{x_i = e_i}^n\ \mathbf{in}\ e]\!] = \left\{(\theta, v) \mid \theta \in \mathbf{Subst}, \left(\overline{v_i}^n\right) = lfp\ F_\theta, \left(\theta\left[\overline{x_i / v_i}^n\right], v\right) \in \mathscr{E}[\![e]\!]\right\}$$

donde $F_\theta\left(\overline{v_i}^n\right) = \left(\overline{v'_i}^n\right)$

y $\forall k \in \{1..n\} . \{v'_k\} = \left\{v \mid \left(\theta\left[\overline{x_i / v_i}^n\right], v\right) \in \mathscr{E}[\![e_k]\!]\right\}$

$$matches(\theta, v, p\ \mathbf{when}\ e_g \to e) = \left\{\theta\left[\overline{x_i / v_i}\right] \mid \overline{v_i} \in \mathbf{DVal}, \left(\theta\left[\overline{x_i / v_i}\right], v\right) \in \mathscr{E}[\![p]\!], \left(\theta\left[\overline{x_i / v_i}\right], \texttt{'true'}\right) \in \mathscr{E}[\![e_g]\!]\right\}$$

donde $vars(p) = \{\overline{x_i}\}$

Figura 2.2: Semántica denotacional de las expresiones

nes modificadas de θ que encajen con la semántica del patrón y satisfagan la guarda de la cláusula.

Usando la función *matches*, la semántica de las sentencias **case** y **receive** elimina aquellos valores correspondientes a las cláusulas que no son ejecutadas debido a que el discriminante ha encajado con el patrón y guarda de una cláusula anterior. Tomemos el siguiente ejemplo:

case X **of**
    Y **when** `'true'` → `'first'`
    Z **when** `'true'` → `'second'`
end

Si no usáramos *matches*, la semántica de la expresión terminaría siendo un conjunto con los valores `'first'` y `'second'`. Pero al usar esta función, todas las posibles sustituciones que tendría la segunda cláusula están contenidas en el conjunto de sustituciones de la primera, con lo que esa es la razón por la que no podemos alcanzar durante la ejecución la segunda cláusula.

Así que, teniendo en cuenta lo anterior, la semántica de una expresión **case** contendrá los valores obtenidos para los cuerpos de las cláusulas, siempre y cuando la sustitución actual para el patrón no esté recogida por alguna cláusula anterior. Esto mismo también valdría para **receive**, con el añadido de tener que comprobar la condición de si x_t es un número entero o el átomo `'infinity'` para las cláusulas de la sentencia, o si se trata de un número entero para evaluar el cuerpo del **after**. Recuérdese que si x_t fuera `'infinity'` nunca alcanzaremos la ejecución de la expresión e asociada a la cláusula **after**.

En la semántica de **letrec** buscaremos los puntos fijos de las funciones F_θ definidas a partir de la semántica de los diferentes $\overline{e_i}$, obteniendo una serie de valores $\overline{v_i}$ que son funciones, pasando luego a restringir en la sustitución θ esos grafos de función obtenidos en cada $\overline{x_i}$ para poder evaluar la expresión e.

Por último, tenemos que señalar que inicialmente en [64] la semántica de una expresión cerrada e era denotada con la función $\mathscr{E}[\![e]\!]$, pero—como se ha visto previamente en esta sección—la semántica actual de la función $\mathscr{E}[\![e]\!]$ está definida como $\mathscr{E}[\![e]\!] \subseteq \mathbf{Subst} \times \mathbf{DVal}$. Para evitar la confusión usaremos la notación $\mathscr{E}[\![e]\!]|_2$ para la semántica de las expresiones cerradas, que definimos como $\mathscr{E}[\![e]\!]|_2 \subseteq \mathbf{DVal}$, y para las expresiones no cerradas usaremos la función $\mathscr{E}[\![e]\!]$.

Capítulo 3

Success types monomórficos

En el capítulo 1 hemos explicado que los objetivos de la tesis consistían en realizar un sistema de tipos para analizar lenguajes de tipado dinámico como es el caso de *Erlang*. Para ello, en última instancia necesitamos un sistema de tipos polimórfico, pero en el transcurso de la investigación, como paso intermedio, desarrollamos primero uno de tipos monomórficos con el que sentar unas bases sobre las que poder progresar. Es por ello que a lo largo del siguiente capítulo vamos a introducir al lector nuestro sistema para derivar *success types* monomórficos [64].

En la sección 3.1 presentaremos la sintaxis de los tipos monomóficos que usa el sistema. En la sección 3.2 detallaremos un poco más sobre la relevancia del uso de *entornos* dentro de los tipos funcionales de la investigación. En la sección 3.3 presentaremos la semántica de los tipos monomóficos. En la sección 3.4 presentaremos las reglas de tipado que proponemos. En la sección 3.5 mostraremos ejemplos de derivación de tipos. Por último en la sección 3.6 presentaremos los resultados de corrección del sistema de tipos.

3.1. Sintaxis de los tipos monomórficos

Cada tipo en el sistema está pensado para ser una sobreaproximación a un conjunto de valores. Asumimos la existencia de un conjunto $\mathbb{B}$ de *tipos básicos* donde encontramos `integer()`, `float()`, `number()`, `atom()`, etcétera, representado cada uno de ellos un conjunto de valores en *Erlang*, es decir, un subconjunto de **DVal**.

Denotamos con **Type** el conjunto de tipos generados por la siguiente gramática:

$$\tau ::= \texttt{none()} \mid \texttt{any()} \mid B \mid c \mid \{\tau_1, \ldots, \tau_n\} \mid \texttt{nelist}(\tau_1, \tau_2) \mid \tau_1 \cup \tau_2 \mid (\tau_1, \ldots, \tau_n) \xrightarrow{\Gamma} \tau$$

donde $B \in \mathbb{B}$ y $c \in$ **DVal** es una expresión literal del lenguaje.

El tipo `none()` denota la ausencia de valores. Si una expresión tiene `none()` como *success type*, su evaluación no tendrá valor alguno, lo que quiere decir que su evaluación fallará o será divergente. Al contrario, el tipo `any()` denota el conjunto **DVal** que comprende todos los valores del lenguaje. Por lo tanto, `any()` siempre sobreaproxima el conjunto de valores a los que una expresión pudiera ser evaluada. En otras palabras, `any()` es un *success type* para toda expresión posible. Por otro lado, el sistema de tipos dispone de *tipos unitarios*, que corresponde con las expresiones literales del lenguaje. Esto quiere decir que para cada valor $c \in$ **DVal** representable por un literal existe un tipo c que denota el conjunto $\{c\}$.

El tipo $\{\tau_1,\ldots,\tau_n\}$ denota aquellas tuplas cuya i-ésima componente está contenida dentro de los valores representados por τ_i, para cada $i \in \{1..n\}$. El tipo `nelist(`τ_1`,`τ_2`)` representa aquellas listas cuyos elementos están dentro del tipo τ_1 y sus continuaciones (es decir, aquellas colas que encontremos en la parte más profunda) pertenecen al conjunto denotado por el tipo τ_2. En *Erlang* podemos hacer distinción entre listas *propias* e *impropias*. Una lista será propia si la cola más profunda es la lista vacía; de lo contrario se tratará de una lista impropia. Por ejemplo, `[1|[2|[3|[]]]]` es una lista propia, mientras que `[a|[b|[c|d]]]` no lo es, ya que su cola más profunda (el átomo `d`) no es la lista vacía. Una lista no vacía propia cuyos elementos tienen el tipo τ la podemos representar con el tipo `nelist(`τ`,[])`, donde `[]` es el tipo unitario que denota la lista vacía.

El tipo unión $\tau_1 \cup \tau_2$ denota el conjunto de valores que están contenidos en τ_1 o lo están en τ_2. Por ejemplo, el tipo `nelist(`τ`,[])` $\cup$ `[]` representa aquellas listas propias (incluidas las vacías) cuyos elementos tienen a τ como tipo. Para abreviar el futuro uso de este tipo, usaremos la notación `[`τ`]`.

El tipo $(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau$ denota el conjunto de funciones de n parámetros que aceptan valores de entrada contenidos en $(\tau_1,\ldots,\tau_n)$ y devuelven como resultado un valor contenido en τ. La Γ que encontramos sobre la flecha es un *entorno de tipos* que podría contener restricciones sobre las variables libres que se encuentran en el cuerpo de la función. Esas restricciones representan condiciones necesarias para la correcta evaluación de la función. El rol que desempeña el entorno Γ lo explicaremos en la siguiente sección.

3.2. Entornos dentro de los tipos funcionales

Un entorno de tipos es una función total que relaciona variables con tipos, es decir, Γ : **Var** $\rightarrow$ **Type**. Nótese que, a diferencia de los sistemas de tipos de *Hindley-Milner*, la definición de los entornos de tipos y los tipos son mutuamente recursivas en nuestro sistema monomórfico. Esta diferencia se debe a que los entornos contienen tipos y los tipos funcionales contienen entornos. Veremos en el siguiente capítulo cómo esta recursión mutua puede ser eliminada gracias al polimorfismo.

En este trabajo utilizamos la notación $[]$ para denotar un entorno donde toda variable está ligada al tipo `any()`, mientras que con $\bot$ denotamos el entorno *bottom* donde ligamos toda variable al tipo `none()`. Además, con la notación $[x_1 : \tau_1,\ldots,x_n : \tau_n]$ indicamos que cada variable x_i está ligada al tipo τ_i, para cada $i \in \{1..n\}$.

Dado un entorno Γ y un conjunto X de variables, la notación $\Gamma\backslash X$ representa el entorno de tipos resultante de reemplazar los tipos de las variables contenidas en X por el tipo `any()`. Esta operación incluye los enlaces existentes en el entorno Γ, pero también se aplica recursivamente a los entornos adjuntos a los tipos funcionales que existan en Γ. De forma similar, para un tipo τ, tenemos la notación $\tau\backslash X$ para un conjunto X de variables, que reemplaza en todos los entornos de tipos que hay dentro de τ los tipos de las variables contenidas en X por el tipo `any()`. Siendo más formales, $\Gamma\backslash X$ denota un entorno Γ' tal que:

$$\Gamma'(y) = \begin{cases} \texttt{any()} & \text{si } y \in X \\ \Gamma(y)\backslash X & \text{en otro caso} \end{cases}$$

donde $\Gamma(y)\backslash X$ es el tipo que daría como resultado de sustituir cada tipo funcional $(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau$ por el tipo $(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma\backslash X} \tau$. Con el fin de hacer más concisa la notación, en el caso de que X sea el conjunto unitario $\{x\}$, dejaremos fuera las llaves para poder usar como notación $\Gamma\backslash x$ y $\tau\backslash x$.

Diremos que una variable está *sin restringir* en Γ si dicha variable tiene como tipo `any()` en ese entorno y en los entornos que estuvieran contenidos en los tipos que contiene Γ. Esto quiere decir, que x está sin restringir en Γ, si y solo si $\Gamma\backslash x = \Gamma$.

Como ejemplo, consideremos la expresión **fun**(`X`) → `X` + 1. Aplicando las reglas que veremos en la sección 3.4 obtendríamos (`number()`) $\xrightarrow{[]}$ `number()` como *success type* para la λ-expresión anterior. Esto significa que si la función recibe como argumento un valor no numérico, la ejecución de esta fallará. Por otra parte, la ejecución de la función podría tener éxito o no, pero si lo tuviera devolverá un valor numérico. El entorno `[]`, que encontramos encima de la flecha del tipo obtenido, indica que la ejecución de la función no requiere de condiciones adicionales sobre variable alguna distinta a `X`. Ahora tomemos en consideración la expresión **fun**(`X`) → `X` + `Y`, donde `Y` es una variable libre. La ejecución de la función, cuando es aplicada a un valor v, no solo depende del hecho de que v sea un número, sino que además dependerá del valor asignado a `Y`. De hecho, `Y` deberá contener un valor numérico, o la evaluación del cuerpo de la función fallaría en otro caso. Sin embargo, no hay forma de reflejar este detalle relativo a la variable `Y` en los tipos asociados a los parámetros de entrada de la función, ni tampoco en el tipo del resultado de la misma. Esto lleva a indicar, en el entorno del tipo funcional, que `Y` ha de tener tipo `number()`. En particular, la expresión **fun**(`X`) → `X` + `Y` aceptará el siguiente *success type*:

$$(\texttt{number()}) \xrightarrow{[\texttt{Y:number()}]} \texttt{number()} \tag{3.1}$$

que significa que, el hecho de que la variable `Y` tenga como tipo a `number()`, es una condición necesaria para la evaluación con éxito de `X` + `Y`.

Por último, para simplificar los ejemplos del capítulo, en aquellos tipos funcionales en los que el entorno sea `[]` también aceptaremos como notación que la flecha no venga acompañada de dicho entorno.

3.3. Semántica de los tipos monomórficos

A continuación procederemos a explicar la semántica de los tipos vistos en la sección 3.1, pero primero hay que señalar que para cada *tipo básico* B usaremos la notación $\mathscr{B}[\![B]\!]$ para representar el conjunto de valores que contiene B. Por ejemplo, $\mathscr{B}[\![\texttt{integer()}]\!]$ incluye el conjunto de los números enteros, mientras que $\mathscr{B}[\![\texttt{atom()}]\!]$ denota el conjunto de los *átomos* de *Erlang* (sección 2.1.1). Los tipos básicos deben cumplir la siguiente propiedad:

$$\forall B_1, B_2 \in \mathbb{B}. \quad \mathscr{B}[\![B_1]\!] \subseteq \mathscr{B}[\![B_2]\!] \quad \vee \quad \mathscr{B}[\![B_2]\!] \subseteq \mathscr{B}[\![B_1]\!] \quad \vee \quad \mathscr{B}[\![B_1]\!] \cap \mathscr{B}[\![B_2]\!] = \emptyset \tag{3.2}$$

porque de lo contrario no podríamos ordenar correctamente estos conjuntos. Los tipos básicos $B \in \mathbb{B}$ deben cumplir también la propiedad de que su semántica $\mathbb{B}$ solo contiene valores literales.

En principio, un tipo denota un conjunto de valores, por lo que necesitaremos una función $\mathscr{T}[\![\_]\!]$ que, dado un tipo τ, devuelva un conjunto $\mathscr{T}[\![\tau]\!] \subseteq \mathbf{DVal}$ que contenga los valores del lenguaje abstraídos por τ. Sin embargo, el ejemplo de la sección anterior muestra que existen tipos cuya semántica no depende exclusivamente del tipo por sí solo, sino que también depende de los tipos asignados a las variables libres que existan. Por ejemplo, si τ es el tipo mostrado en (3.1) e `Y` contiene un valor numérico, este tipo denotaría un conjunto de funciones de números a números. Teniendo en cuenta que las funciones son representadas como grafos, la semántica de τ sería algo como lo siguiente:

$$\left\{F \mid F \subseteq \left\{\left((v), v'\right) \mid v, v' \in \mathscr{B}[\![\texttt{number()}]\!]\right\}\right\}$$

No obstante, si `Y` estuviera ligado a un valor no numérico, entonces la semántica prevista para (3.1) es la función que siempre falla. Esta función la representamos con el grafo vacío, por ello la semántica de τ consistirá en una única función que es el grafo vacío, es decir, $\{\emptyset\}$. Este ejemplo demuestra la necesidad de una semántica que también sea paramétrica en el *contexto* en el que ocurre un tipo. Este contexto es dado por una sustitución θ que contiene el valor asignado a cada variable. Por lo tanto, en lugar de $\mathscr{T}[\![\tau]\!]$ usaremos $\mathscr{T}_\theta[\![\tau]\!]$ para denotar la semántica del tipo τ bajo una sustitución θ. La definición de $\mathscr{T}_\theta[\![\tau]\!]$ la podemos encontrar en la figura 3.1. Esta definición es mutuamente recursiva con la función $\mathscr{T}_{Env}[\![\_]\!]$ que devuelve la semántica de un entorno de tipos. Teniendo en cuenta que un entorno de tipos Γ es una relación de variables a tipos, es natural definir su semántica como el conjunto de sustituciones θ tal que $\theta(x)$ está contenida dentro de la semántica de $\Gamma(x)$ para cada variable x.

Como se ha mencionado en la sección 2.3 del capítulo anterior, la semántica de las expresiones cerradas está representada con la función $\mathscr{E}[\![e]\!]|_2 \subseteq \mathbf{DVal}$ y para expresiones abiertas usamos la función $\mathscr{E}[\![e]\!] \subseteq \mathbf{Subst} \times \mathbf{DVal}$. Por ello, de forma similar que en el caso de las expresiones, adaptaremos la semántica de los tipos para que también esté definida como relación entre sustituciones y valores. Este trabajo usa la notación $\mathscr{T}[\![\tau]\!]$ para denotar la semántica de τ como relación entre sustituciones y va-

$$
\begin{aligned}
\mathscr{T}_\theta[\![\texttt{none}()]\!] &= \emptyset \\
\mathscr{T}_\theta[\![\texttt{any}()]\!] &= \mathbf{DVal} \\
\mathscr{T}_\theta[\![B]\!] &= \mathscr{B}[\![B]\!] \\
\mathscr{T}_\theta[\![c]\!] &= \{c\} \\
\mathscr{T}_\theta[\![\{\tau_1,\ldots,\tau_n\}]\!] &= \left\{\left(\{\overset{n}{\cdots}\}, v_1,\ldots,v_n\right) \mid \forall i \in \{1..n\}.\, v_i \in \mathscr{T}_\theta[\![\tau_i]\!]\right\} \\
\mathscr{T}_\theta[\![\texttt{nelist}(\tau_1,\tau_2)]\!] &= \left\{\left([\_\mid\_], \overline{v_i}^n, v'\right) \mid n \geq 1, \forall i \in \{1..n\}.\, v_i \in \mathscr{T}_\theta[\![\tau_1]\!], v' \in \mathscr{T}_\theta[\![\tau_2]\!]\right\} \\
\mathscr{T}_\theta[\![\tau_1 \cup \tau_2]\!] &= \mathscr{T}_\theta[\![\tau_1]\!] \cup \mathscr{T}_\theta[\![\tau_2]\!] \\
\mathscr{T}_\theta\left[\!\!\left[(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau\right]\!\!\right] &= \begin{cases} \{\emptyset\} & \text{si } \theta \notin \mathscr{T}_{Env}[\![\Gamma]\!] \\ \left\{F \mid F \subseteq \left\{((v_1,\ldots,v_n), v) \mid \forall i. v_i \in \mathscr{T}_\theta[\![\tau_i]\!], v \in \mathscr{T}_\theta[\![\tau]\!]\right\}\right\} & \text{en otro caso} \end{cases}
\end{aligned}
$$

$$
\mathscr{T}_{Env}[\![\Gamma]\!] = \left\{\theta \in \mathbf{Subst} \mid \forall x \in \mathbf{Var}.\theta(x) \in \mathscr{T}_\theta[\![\Gamma(x)]\!]\right\}
$$

Figura 3.1: Semántica de los tipos y entornos de tipos

lores, donde cada sustitución θ está relacionada con cada valor en $\mathscr{T}_\theta[\![\tau]\!]$. Es decir,

$$
\mathscr{T}[\![\tau]\!] \overset{def}{=} \left\{(\theta, v) \mid \theta \in \mathbf{Subst}, v \in \mathscr{T}_\theta[\![\tau]\!]\right\}
$$

En el contexto de las expresiones cerradas, la sustitución θ es irrelevante para la semántica del tipo, por lo que definimos $\mathscr{T}[\![\tau]\!]|_2 = \{v \mid (\theta, v) \in \mathscr{T}[\![\tau]\!], \theta \in \mathbf{Subst}\}$.

Con estas definiciones podemos decir que τ es un *success type* para una expresión e si, para toda sustitución θ, el conjunto de valores a los que $e\theta$ puede ser evaluada está contenida dentro de la semántica de τ bajo la misma sustitución. Siendo más precisos, tendremos la siguiente definición de *success type*:

Definición 1. El tipo τ es un *success type* para la expresión e si y solo si $\mathscr{E}[\![e]\!] \subseteq \mathscr{T}[\![\tau]\!]$.

Esta definición es una generalización de la que fue dada previamente en [66], porque cuando e es cerrada y todos los entornos en τ están vacíos, es equivalente a $\mathscr{E}[\![e]\!]|_2 \subseteq \mathscr{T}[\![\tau]\!]|_2$.

3.3.1. Relación de subtipado

De cara a establecer resultados de corrección semántica del sistema de tipos que presentamos en 3.4, necesitamos algunas nociones adicionales, comenzando por una noción de subtipado, tanto para tipos como para entornos de tipos, que determinará una estructura de retículo de tipos.

Dentro de la semántica de los tipos, será necesario demostrar una serie de proposiciones para poder trabajar luego con las mismas en la corrección. En concreto, es necesario poder definir la relación de subtipado entre diferentes tipos y entornos, así como también poder demostrar la existencia de un retículo de tipos.

Primero vamos a definir la relación de subtipado para tipos de la siguiente manera:

Definición 2 (Relación $\subseteq$ de subtipado para tipos)**.** Dados dos tipos τ_1 y τ_2, decimos que $\tau_1 \subseteq \tau_2$ si y solo si $\mathcal{T}_\theta [\![\tau_1]\!] \subseteq \mathcal{T}_\theta [\![\tau_2]\!]$ para toda sustitución $\theta \in$ **Subst**.

También podemos, de modo equivalente, utilizar la siguiente proposición:

Proposición 1. *Para cada par de tipos τ_1 y τ_2, es cierto que $\tau_1 \subseteq \tau_2$ si y solo si $\mathcal{T} [\![\tau_1]\!] \subseteq \mathcal{T} [\![\tau_2]\!]$.*

Demostración. Asumiremos que $\tau_1 \subseteq \tau_2$. Por la definición, obtenemos que $\forall \theta, v. v \in \mathcal{T}_\theta [\![\tau_1]\!] \Rightarrow v \in \mathcal{T}_\theta [\![\tau_2]\!]$, que podemos reescribir como $\forall (\theta, v).(\theta, v) \in \mathcal{T} [\![\tau_1]\!] \Rightarrow (\theta, v) \in \mathcal{T} [\![\tau_2]\!]$, obteniendo como resultado que $\mathcal{T} [\![\tau_1]\!] \subseteq \mathcal{T} [\![\tau_2]\!]$.

Ahora asumiendo que $\mathcal{T} [\![\tau_1]\!] \subseteq \mathcal{T} [\![\tau_2]\!]$, por la definición, obtenemos que $\forall (\theta, v).(\theta, v) \in \mathcal{T} [\![\tau_1]\!] \Rightarrow (\theta, v) \in \mathcal{T} [\![\tau_2]\!]$, que puede ser reescrito como $\forall \theta, v. v \in \mathcal{T}_\theta [\![\tau_1]\!] \Rightarrow v \in \mathcal{T}_\theta [\![\tau_2]\!]$, y por lo tanto $\tau_1 \subseteq \tau_2$. □

Extenderemos la relación de subtipado a entornos de tipos de la siguiente forma:

Definición 3 (Relación $\subseteq$ de subtipado para entornos)**.** Dados dos entornos Γ_1 y Γ_2, decimos que $\Gamma_1 \subseteq \Gamma_2$ si y solo si $\mathcal{T}_{Env} [\![\Gamma_1]\!] \subseteq \mathcal{T}_{Env} [\![\Gamma_2]\!]$.

De modo análogo al de los tipos, tendremos la siguiente proposición:

Proposición 2. *Si $\Gamma_1(x) \subseteq \Gamma_2(x)$ para toda variable $x \in$ **Var**, entonces $\Gamma_1 \subseteq \Gamma_2$.*

Demostración. Asumiremos una sustitución θ tal que $\theta \in \mathcal{T}_{Env} [\![\Gamma_1]\!]$. Esto significa que, para toda variable x se tiene que $\theta(x) \in \mathcal{T}_\theta [\![\Gamma_1(x)]\!] \subseteq \mathcal{T}_\theta [\![\Gamma_2(x)]\!]$. Por lo tanto, $\theta \in \mathcal{T}_{Env} [\![\Gamma_2]\!]$. □

Sin embargo, lo recíproco no es cierto de forma general. Por ejemplo, si tenemos los entornos $\Gamma_1 =$ [`X : integer(),Z : none()`] y $\Gamma_2 =$ [`X : atom(),Z : none()`], se cumple que $\mathcal{T}_{Env} [\![\Gamma_1]\!] = \mathcal{T}_{Env} [\![\Gamma_2]\!] = \emptyset$, pero $\Gamma_1(X)$ y $\Gamma_2(X)$ resultan ser incomparables entre sí. Esto se debe a que la relación de subtipado no es, en sentido estricto, una relación de orden, porque no es antisimétrica. Por ejemplo, los tipos $any() \cup 0$ y $any()$ son subtipo uno de otro. Lo mismo pasa con los entornos Γ_1 y Γ_2. Para lidiar con esta situación, definimos una relación de equivalencia en la que, dados dos tipos τ y τ', diremos que son *equivalentes* si y solo si su semántica es igual. De forma análoga, definiremos la misma relación de equivalencia para dos entornos Γ y Γ'. Es decir:

$$\tau \approx \tau' \iff \mathcal{T} [\![\tau]\!] = \mathcal{T} [\![\tau]\!] \qquad\qquad \Gamma \approx \Gamma' \iff \mathcal{T}_{Env} [\![\Gamma]\!] = \mathcal{T}_{Env} [\![\Gamma]\!]$$

Por abuso de notación, podemos describir la igualdad entre tipos como la igualdad entre las respectivas clases de equivalencia. De este modo pasaríamos de tener un preorden en nuestro conjunto de tipos a tener un retículo de tipos ordenado.

3.3.2. Ínfimo de tipos y entornos

Dados dos tipos τ y τ', usaremos la notación $\tau \sqcap \tau'$ para representar el ínfimo entre tipos, que definiremos de la siguiente manera:

1. $\tau \sqcap \tau = \tau$
2. $\tau \sqcap \texttt{any}() = \tau$
3. $\tau \sqcap \texttt{none}() = \texttt{none}()$
4. $c \sqcap B = \begin{cases} c & c \in \mathscr{B}[\![B]\!] \\ \texttt{none}() & c \notin \mathscr{B}[\![B]\!] \end{cases}$
5. $B \sqcap B' = \begin{cases} B & \mathscr{B}[\![B]\!] \subseteq \mathscr{B}[\![B']\!] \\ B' & \mathscr{B}[\![B']\!] \subseteq \mathscr{B}[\![B]\!] \\ \texttt{none}() & \text{en otro caso} \end{cases}$
6. $\tau \sqcap (\tau' \cup \tau'') = (\tau \sqcap \tau') \cup (\tau \sqcap \tau'')$
7. $\left((\overline{\tau_i}^n) \xrightarrow{\Gamma} \tau\right) \sqcap \left((\overline{\tau'_i}^n) \xrightarrow{\Gamma'} \tau'\right) = \left(\overline{\tau_i \sqcap \tau'_i}^n\right) \xrightarrow{\Gamma \sqcap \Gamma'} \tau \sqcap \tau'$
8. $$\begin{aligned} \texttt{nelist}(\tau_1,\tau_2) \sqcap \texttt{nelist}(\tau_3,\tau_4) &= \texttt{nelist}(\tau_1 \sqcap \tau_3, \tau_2 \sqcap \tau_4) \\ &\cup \texttt{nelist}(\tau_1 \sqcap \tau_3, \texttt{nelist}(\tau_1,\tau_2) \sqcap \tau_4) \\ &\cup \texttt{nelist}(\tau_1 \sqcap \tau_3, \texttt{nelist}(\tau_3,\tau_4) \sqcap \tau_2) \end{aligned}$$
9. $\{\overline{\tau_i}^n\} \sqcap \{\overline{\tau'_i}^n\} = \{\overline{\tau_i \sqcap \tau'_i}^n\}$
10. En otro caso: $\tau \sqcap \tau' = \texttt{none}()$

Para evitar duplicidades en los casos, hemos de tener en cuenta que el caso $\tau \sqcap \tau'$ también engloba el caso $\tau' \sqcap \tau$, ya que la operación del ínfimo es conmutativa. Como podemos observar también se realiza, dentro de los tipos funcionales, el ínfimo entre dos entornos. Así que dados dos entornos de tipos Γ y Γ', la notación $\Gamma \sqcap \Gamma'$ denota aquel entorno Γ'' tal que $\Gamma''(x) = \Gamma(x) \sqcap \Gamma'(x)$, para todo $x \in \mathbf{Var}$.

Teniendo en cuenta lo anterior, se cumple la siguiente proposición para el ínfimo entre tipos y entornos:

Proposición 3. *Para todo $\tau_1, \tau_2, \Gamma_1, \Gamma_2$ y θ, es cierto que:*

1. $\mathscr{T}_\theta[\![\tau_1 \sqcap \tau_2]\!] = \mathscr{T}_\theta[\![\tau_1]\!] \cap \mathscr{T}_\theta[\![\tau_2]\!]$, *y por tanto* $\mathscr{T}[\![\tau_1 \sqcap \tau_2]\!] = \mathscr{T}[\![\tau_1]\!] \cap \mathscr{T}[\![\tau_2]\!]$.

2. $\mathscr{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!] = \mathscr{T}_{Env}[\![\Gamma_1]\!] \cap \mathscr{T}_{Env}[\![\Gamma_2]\!]$.

Demostración. Por inducción sobre las estructuras de τ_1, τ_2, Γ_1 y Γ_2. Empezaremos primero demostrando (1) distinguiendo cada caso de la operación.

- **Caso** $\tau_1 = \tau_2 = \tau$

  Cuando ambos operandos son el mismo tipo, tenemos que:

$$\begin{aligned} \mathscr{T}_\theta[\![\tau \sqcap \tau]\!] &= \mathscr{T}_\theta[\![\tau]\!] \\ &= \mathscr{T}_\theta[\![\tau]\!] \cap \mathscr{T}_\theta[\![\tau]\!] \end{aligned}$$

- **Caso** $\tau_2 = \texttt{any()}$

  Cuando uno de los dos operandos es el tipo `any()`, tenemos que:

$$\begin{aligned} \mathscr{T}_\theta[\![\tau_1 \sqcap \texttt{any()}]\!] &= \mathscr{T}_\theta[\![\tau_1]\!] \\ &= \mathscr{T}_\theta[\![\tau_1]\!] \cap \mathbf{DVal} && \text{porque } \mathscr{T}_\theta[\![\tau_1]\!] \subseteq \mathbf{DVal} \\ &= \mathscr{T}_\theta[\![\tau_1]\!] \cap \mathscr{T}_\theta[\![\texttt{any()}]\!] \end{aligned}$$

- **Caso** $\tau_2 = \texttt{none()}$

  Cuando uno de los dos operandos es el tipo `none()`, tenemos que:

$$\begin{aligned} \mathscr{T}_\theta[\![\tau_1 \sqcap \texttt{none()}]\!] &= \mathscr{T}_\theta[\![\texttt{none()}]\!] \\ &= \emptyset \\ &= \mathscr{T}_\theta[\![\tau_1]\!] \cap \emptyset \\ &= \mathscr{T}_\theta[\![\tau_1]\!] \cap \mathscr{T}_\theta[\![\texttt{none()}]\!] \end{aligned}$$

- **Caso** $\tau_2 = B_2$

  Cuando uno los operandos es un tipo básico, tenemos que distinguir entre varios subcasos. El primero es cuando $\tau_1 = c$, donde si $c \in \mathscr{B}[\![B_2]\!]$ tenemos que:

$$\begin{aligned} \mathscr{T}_\theta[\![c \sqcap B_2]\!] &= \mathscr{T}_\theta[\![c]\!] \\ &= \{c\} \\ &= \{c\} \cap \mathscr{B}[\![B_2]\!] && \text{porque } c \in \mathscr{B}[\![B_2]\!] \\ &= \mathscr{T}_\theta[\![c]\!] \cap \mathscr{T}_\theta[\![B_2]\!] \end{aligned}$$

Pero si $c \notin \mathscr{B}[\![B_2]\!]$ obtendremos que:

$$\begin{aligned}
\mathscr{T}_\theta[\![c \sqcap B_2]\!] &= \mathscr{T}_\theta[\![\texttt{none()}]\!] \\
&= \emptyset \\
&= \{c\} \cap \mathscr{B}[\![B_2]\!] \qquad \text{porque } c \notin \mathscr{B}[\![B_2]\!] \\
&= \mathscr{T}_\theta[\![c]\!] \cap \mathscr{T}_\theta[\![B_2]\!]
\end{aligned}$$

El segundo subcaso es cuando $\tau_1 = B_1$. Teniendo en cuenta la propiedad 3.2, sabemos que: (A) $\mathscr{B}[\![B_1]\!] \subseteq \mathscr{B}[\![B_2]\!]$, o bien (B) $\mathscr{B}[\![B_2]\!] \subseteq \mathscr{B}[\![B_1]\!]$, o bien (C) $\mathscr{B}[\![B_1]\!] \cap \mathscr{B}[\![B_2]\!] = \emptyset$. Con el caso (A), tenemos que:

$$\begin{aligned}
\mathscr{T}_\theta[\![B_1 \sqcap B_2]\!] &= \mathscr{T}_\theta[\![B_1]\!] \\
&= \mathscr{B}[\![B_1]\!] \\
&= \mathscr{B}[\![B_1]\!] \cap \mathscr{B}[\![B_2]\!] \qquad \text{porque } \mathscr{B}[\![B_1]\!] \subseteq \mathscr{B}[\![B_2]\!] \\
&= \mathscr{T}_\theta[\![B_1]\!] \cap \mathscr{T}_\theta[\![B_2]\!]
\end{aligned}$$

Con el caso (B), tenemos lo siguiente:

$$\begin{aligned}
\mathscr{T}_\theta[\![B_1 \sqcap B_2]\!] &= \mathscr{T}_\theta[\![B_2]\!] \\
&= \mathscr{B}[\![B_2]\!] \\
&= \mathscr{B}[\![B_1]\!] \cap \mathscr{B}[\![B_2]\!] \qquad \text{porque } \mathscr{B}[\![B_2]\!] \subseteq \mathscr{B}[\![B_1]\!] \\
&= \mathscr{T}_\theta[\![B_1]\!] \cap \mathscr{T}_\theta[\![B_2]\!]
\end{aligned}$$

Por último con el caso (C), nos encontramos con:

$$\begin{aligned}
\mathscr{T}_\theta[\![B_1 \sqcap B_2]\!] &= \mathscr{T}_\theta[\![\texttt{none()}]\!] \\
&= \emptyset \\
&= \mathscr{B}[\![B_1]\!] \cap \mathscr{B}[\![B_2]\!] \qquad \text{porque } \mathscr{B}[\![B_1]\!] \cap \mathscr{B}[\![B_2]\!] = \emptyset \\
&= \mathscr{T}_\theta[\![B_1]\!] \cap \mathscr{T}_\theta[\![B_2]\!]
\end{aligned}$$

Para los subcasos donde τ_1 no es un literal o un tipo básico, si se trata `any()` o `none()`, utilizaremos los casos anteriores. Una de las propiedades de los tipos básicos nos dice que $\mathscr{B}[\![B_2]\!]$ solo contendrá valores que sean literales. Teniendo presente lo anterior, si τ_1 tiene la forma $\{\overline{\tau'_i}^{\,n}\}$, $\texttt{nelist}(\tau'_1, \tau''_1)$ o $\left(\overline{\tau'_i}^{\,n}\right) \xrightarrow{\Gamma'} \tau'$, sabemos que $\mathscr{B}[\![B_2]\!] \not\subseteq \mathscr{T}_\theta[\![\tau_1]\!]$. Por lo tanto:

$$\begin{aligned}
\mathscr{T}_\theta[\![\tau_1 \sqcap B_2]\!] &= \mathscr{T}_\theta[\![\texttt{none()}]\!] \\
&= \emptyset \\
&= \mathscr{T}_\theta[\![\tau_1]\!] \cap \mathscr{B}[\![B_2]\!] \qquad \text{porque } \mathscr{B}[\![B_2]\!] \text{ solo contiene literales} \\
&= \mathscr{T}_\theta[\![\tau_1]\!] \cap \mathscr{T}_\theta[\![B_2]\!]
\end{aligned}$$

- **Caso** $\tau_2 = \tau'_1 \cup \tau'_2$

Cuando uno de los dos operandos es un tipo unión, tenemos que:

$$\begin{aligned}
\mathcal{T}_\theta[\![\tau_1 \sqcap (\tau'_1 \cup \tau'_2)]\!] &= \mathcal{T}_\theta[\![(\tau_1 \sqcap \tau'_1) \cup (\tau_1 \sqcap \tau'_2)]\!] \\
&= \mathcal{T}_\theta[\![\tau_1 \sqcap \tau'_1]\!] \cup \mathcal{T}_\theta[\![\tau_1 \sqcap \tau'_2]\!] \\
&= (\mathcal{T}_\theta[\![\tau_1]\!] \cap \mathcal{T}_\theta[\![\tau'_1]\!]) \cup (\mathcal{T}_\theta[\![\tau_1]\!] \cap \mathcal{T}_\theta[\![\tau'_2]\!]) \quad \text{por h.i.} \\
&= \mathcal{T}_\theta[\![\tau_1]\!] \cap (\mathcal{T}_\theta[\![\tau'_1]\!] \cup \mathcal{T}_\theta[\![\tau'_2]\!]) \\
&= \mathcal{T}_\theta[\![\tau_1]\!] \cap \mathcal{T}_\theta[\![\tau'_1 \cup \tau'_2]\!]
\end{aligned}$$

- **Caso** $\tau_2 = \{\overline{\tau''_i}^n\}$

Cuando uno los operandos es un tipo tupla, tenemos que distinguir entre varios subcasos. Asumamos que τ_1 es una tupla de la forma $\{\overline{\tau'_i}^n\}$, tenemos que:

$$\begin{aligned}
& \mathcal{T}_\theta\left[\!\!\left[\{\overline{\tau'_i}^n\} \sqcap \{\overline{\tau''_i}^n\}\right]\!\!\right] \\
= \; & \mathcal{T}_\theta\left[\!\!\left[\{\overline{\tau'_i \sqcap \tau''_i}^n\}\right]\!\!\right] \\
= \; & \left\{\left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right) \mid \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta[\![\tau'_i \sqcap \tau''_i]\!]\right\} \\
= \; & \left\{\left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right) \mid \forall i \in \{1..n\} : v_i \in \mathcal{T}_\theta[\![\tau'_i]\!], v_i \in \mathcal{T}_\theta[\![\tau''_i]\!]\right\} \quad \text{por h.i.} \\
= \; & \left\{\left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right) \mid \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta[\![\tau'_i]\!]\right\} \cap \\
& \left\{\left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right) \mid \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta[\![\tau''_i]\!]\right\} \\
= \; & \mathcal{T}_\theta\left[\!\!\left[\{\overline{\tau'_i}^n\}\right]\!\!\right] \cap \mathcal{T}_\theta\left[\!\!\left[\{\overline{\tau''_i}^n\}\right]\!\!\right]
\end{aligned}$$

Para aquellos subcasos donde τ_1 no es una tupla, si tiene la forma $\tau'_1 \cup \tau'_2$ o $\texttt{any}()$, ya han sido tratados en los casos anteriores. Para los subcasos restantes, $\mathcal{T}_\theta[\![\tau_1]\!]$ no contiene tuplas de n componentes, por lo que:

$$\begin{aligned}
\mathcal{T}_\theta\left[\!\!\left[\tau_1 \sqcap \{\overline{\tau''_i}^n\}\right]\!\!\right] &= \mathcal{T}_\theta[\![\texttt{none}()]\!] \\
&= \emptyset \\
&= \mathcal{T}_\theta[\![\tau_1]\!] \cap \mathcal{T}_\theta\left[\!\!\left[\{\overline{\tau''_i}^n\}\right]\!\!\right]
\end{aligned}$$

- **Caso** $\tau_2 = \texttt{nelist}(\tau'_2, \tau''_2)$

Cuando uno los operandos es un tipo lista no vacía, tenemos que distinguir entre varios subcasos. Si τ_1 tiene la forma $\texttt{nelist}(\tau'_1, \tau''_1)$, para demostrar la igualdad $\mathcal{T}_\theta[\![\tau_1 \sqcap \tau_2]\!] = \mathcal{T}_\theta[\![\tau_1]\!] \cap \mathcal{T}_\theta[\![\tau_2]\!]$ probaremos cada inclusión:

1. $\mathcal{T}_\theta[\![\tau_1 \sqcap \tau_2]\!] \subseteq \mathcal{T}_\theta[\![\tau_1]\!] \cap \mathcal{T}_\theta[\![\tau_2]\!]$

   Sea un valor $v \in \mathcal{T}_\theta[\![\tau_1 \sqcap \tau_2]\!]$. Entonces, tenemos que demostrar que:

$$\begin{aligned}
v \in \; & \mathcal{T}_\theta[\![\texttt{nelist}(\tau'_1 \sqcap \tau'_2, \tau''_1 \sqcap \tau''_2)]\!] \\
\cup \; & \mathcal{T}_\theta[\![\texttt{nelist}(\tau'_1 \sqcap \tau'_2, \texttt{nelist}(\tau'_1, \tau''_1) \sqcap \tau''_2)]\!] \\
\cup \; & \mathcal{T}_\theta[\![\texttt{nelist}(\tau'_1 \sqcap \tau'_2, \texttt{nelist}(\tau'_2, \tau''_2) \sqcap \tau''_1)]\!]
\end{aligned}$$

Razonamos distinguiendo casos según a qué conjunto de esa unión pertenezca v. En cualquier caso, sabemos que $v = [v_1, \ldots, v_n \mid v']$, para ciertos $v_1, \ldots, v_n, v'$.

a) Con $v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1 \sqcap \tau'_2, \tau''_1 \sqcap \tau''_2) \rrbracket$, tendremos por hipótesis de inducción que:

$$\begin{array}{rcl} \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \sqcap \tau'_2 \rrbracket & \Longrightarrow & v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket \cap \mathcal{T}_\theta \llbracket \tau'_2 \rrbracket \\ v' \in \mathcal{T}_\theta \llbracket \tau''_1 \sqcap \tau''_2 \rrbracket & \Longrightarrow & v' \in \mathcal{T}_\theta \llbracket \tau''_1 \rrbracket \cap \mathcal{T}_\theta \llbracket \tau''_2 \rrbracket \end{array}$$

Tomando cada elemento de la intersección para los v_i y v', podremos volver a construir v con la forma $[v_1, \ldots, v_n \mid v']$, para obtener lo siguiente:

$$\begin{array}{rcl} \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket \wedge v' \in \mathcal{T}_\theta \llbracket \tau''_1 \rrbracket & \Longrightarrow & v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \rrbracket \\ \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_2 \rrbracket \wedge v' \in \mathcal{T}_\theta \llbracket \tau''_2 \rrbracket & \Longrightarrow & v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_2, \tau''_2) \rrbracket \end{array}$$

Todo ello junto nos da como resultado que:

$$v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \rrbracket \cap \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_2, \tau''_2) \rrbracket$$

b) Con $v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1 \sqcap \tau'_2, \texttt{nelist}(\tau'_1, \tau''_1) \sqcap \tau''_2) \rrbracket$, tendremos por hipótesis de inducción que:

$$\begin{array}{rcl} \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \sqcap \tau'_2 \rrbracket & \Longrightarrow & v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket \cap \mathcal{T}_\theta \llbracket \tau'_2 \rrbracket \\ v' \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \sqcap \tau''_2 \rrbracket & \Longrightarrow & v' \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \rrbracket \cap \mathcal{T}_\theta \llbracket \tau''_2 \rrbracket \end{array}$$

Con $v' \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \rrbracket$ podemos descomponer el valor v' con la siguiente forma $[v'_1, \ldots, v'_m \mid v'']$, de modo que $v'_j \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket$ para cada $j \in \{1..m\}$ y $v'' \in \mathcal{T}_\theta \llbracket \tau''_1 \rrbracket$. Por lo que v pasa de la forma $[v_1, \ldots, v_n \mid v']$ a la siguiente $[v_1, \ldots, v_n, v'_1, \ldots, v'_m \mid v'']$. Desde aquí, obtendremos que:

$$\left.\begin{array}{l} \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket \\ \forall j \in \{1..m\}.\, v'_j \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket \\ v'' \in \mathcal{T}_\theta \llbracket \tau''_1 \rrbracket \end{array}\right\} \Longrightarrow v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \rrbracket$$

$$\left.\begin{array}{l} \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_2 \rrbracket \\ v' \in \mathcal{T}_\theta \llbracket \tau''_2 \rrbracket \end{array}\right\} \Longrightarrow v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_2, \tau''_2) \rrbracket$$

Todo ello junto nos da como resultado que:

$$v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \rrbracket \cap \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_2, \tau''_2) \rrbracket$$

c) Con $v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1 \sqcap \tau'_2, \texttt{nelist}(\tau'_2, \tau''_2) \sqcap \tau''_1) \rrbracket$, la demostración es análoga a la vista en el caso (1.b).

2. $\mathcal{T}_\theta \llbracket \tau_1 \sqcap \tau_2 \rrbracket \supseteq \mathcal{T}_\theta \llbracket \tau_1 \rrbracket \cap \mathcal{T}_\theta \llbracket \tau_2 \rrbracket$

Asumimos un valor v tal que $v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \rrbracket$ y $v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_2, \tau''_2) \rrbracket$. Con ello obtenemos, por un lado, que v tiene la forma $[v_1, \ldots, v_n \mid v']$, donde $v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket$ para cada $i \in \{1..n\}$ y $v' \in \mathcal{T}_\theta \llbracket \tau''_1 \rrbracket$. Y por el otro lado, tenemos que v tiene la forma $[v'_1, \ldots, v'_m \mid v'']$, donde $v'_j \in \mathcal{T}_\theta \llbracket \tau'_2 \rrbracket$ para cada $j \in \{1..m\}$ y $v'' \in \mathcal{T}_\theta \llbracket \tau''_2 \rrbracket$. Desde aquí tendremos que distinguir los siguientes casos:

a) **Caso** $n = m$

En este caso tenemos que $v' = v''$ y $v_i = v'_i$ para cada $i \in \{1..n\}$, por lo tanto:

$$\begin{array}{rcl} \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket \cap \mathcal{T}_\theta \llbracket \tau'_2 \rrbracket & \wedge & v' \in \mathcal{T}_\theta \llbracket \tau''_1 \rrbracket \cap \mathcal{T}_\theta \llbracket \tau''_2 \rrbracket \\ \Downarrow & & \{ \text{por h.i.} \} \\ \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \sqcap \tau'_2 \rrbracket & \wedge & v' \in \mathcal{T}_\theta \llbracket \tau''_1 \sqcap \tau''_2 \rrbracket \end{array}$$

Con esto obtenemos que v tiene la forma $[v_1, \ldots, v_n \mid v']$ y que se cumple que:

$$v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1 \sqcap \tau'_2, \tau''_1 \sqcap \tau''_2) \rrbracket \implies v \in \mathcal{T}_\theta \llbracket \tau_1 \sqcap \tau_2 \rrbracket$$

b) **Caso** $n < m$

Tenemos que $v = [v_1, \ldots, v_n \mid v']$ y $v = [v'_1, \ldots, v'_n, \ldots, v'_m \mid v'']$, por lo que para todo $i \in \{1..n\}$ tenemos que $v_i = v'_i$ y que $v' = [v'_{n+1}, \ldots, v'_m \mid v'']$. Sabemos que los valores $v'_{n+1}, \ldots, v'_m$ pertenecen $\mathcal{T}_\theta \llbracket \tau'_2 \rrbracket$ y que v'' pertenece a $\mathcal{T}_\theta \llbracket \tau''_2 \rrbracket$, por lo tanto $v' \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_2, \tau''_2) \rrbracket$. Además, como $v' \in \mathcal{T}_\theta \llbracket \tau''_1 \rrbracket$, por hipótesis de inducción obtenemos que $v' \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_2, \tau''_2) \sqcap \tau''_1 \rrbracket$. Por otro lado, como para todo $i \in \{1..n\}$ tenemos que $v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket$ y que como $v_i = v'_i$ entonces $v_i \in \mathcal{T}_\theta \llbracket \tau'_2 \rrbracket$. Con esto, por hipótesis de inducción tenemos que $v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \sqcap \tau'_2 \rrbracket$. Por consiguiente:

$$\begin{array}{c} \forall i \in \{1..n\}.\, v_i \in \mathcal{T}_\theta \llbracket \tau'_1 \sqcap \tau'_2 \rrbracket \quad \wedge \quad v' \in \mathcal{T}_\theta \llbracket \tau''_1 \rrbracket \cap \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_2, \tau''_2) \sqcap \tau''_1 \rrbracket \\ \Downarrow \\ v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1 \sqcap \tau'_2, \texttt{nelist}(\tau'_2, \tau''_2) \sqcap \tau''_1) \rrbracket \end{array}$$

Y con ello se demuestra que $v \in \mathcal{T}_\theta \llbracket \tau_1 \sqcap \tau_2 \rrbracket$ para este caso.

c) **Caso** $n > m$

Tenemos que $v = [v_1, \ldots, v_m, \ldots, v_n \mid v']$ y $v = [v'_1, \ldots, v'_m \mid v'']$, por lo que para todo $i \in \{1..m\}$ tenemos que $v_i = v'_i$ y que $v' = [v'_{m+1}, \ldots, v'_n \mid v']$. Sabemos que los valores $v_{m+1}, \ldots, v_n$ pertenecen $\mathcal{T}_\theta \llbracket \tau'_1 \rrbracket$ y que v' pertenece a $\mathcal{T}_\theta \llbracket \tau''_1 \rrbracket$, por lo tanto $v'' \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \rrbracket$. Además, como $v'' \in \mathcal{T}_\theta \llbracket \tau''_2 \rrbracket$, por hipótesis de inducción obtenemos que $v'' \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \sqcap \tau''_2 \rrbracket$. Por otro lado, como para todo $i \in \{1..m\}$ tenemos que $v'_i \in \mathcal{T}_\theta \llbracket \tau'_2 \rrbracket$ y que como $v_i = v'_i$ entonces $v'_i \in \mathcal{T}_\theta \llbracket \tau'_1 \rrbracket$. Con esto, por

hipótesis de inducción tenemos que $v'_i \in \mathcal{T}_\theta \llbracket \tau'_1 \sqcap \tau'_2 \rrbracket$. Por consiguiente:

$$\begin{array}{c} \forall i \in \{1..m\} . v'_i \in \mathcal{T}_\theta \llbracket \tau'_1 \sqcap \tau'_2 \rrbracket \quad \wedge \quad v'' \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1, \tau''_1) \sqcap \tau''_2 \rrbracket \\ \Downarrow \\ v \in \mathcal{T}_\theta \llbracket \texttt{nelist}(\tau'_1 \sqcap \tau'_2, \texttt{nelist}(\tau'_1, \tau''_1) \sqcap \tau''_2) \rrbracket \end{array}$$

Y con ello se demuestra que $v \in \mathcal{T}_\theta \llbracket \tau_1 \sqcap \tau_2 \rrbracket$ para este caso.

Para aquellos subcasos donde τ_1 no es una lista vacía, la demostración es similar a la que hemos visto en la tupla, siendo el resultado de $\tau_1 \sqcap \tau_2 = \texttt{none}()$ salvo para el caso de $\texttt{any}()$ o una unión de tipos.

- **Caso** $\tau_2 = \left(\overline{\tau''_i}^n\right) \xrightarrow{\Gamma''} \tau''$

  Cuando uno los operandos es un tipo funcional, tenemos que distinguir entre varios subcasos. Si τ_1 tiene la forma $\left(\overline{\tau'_i}^n\right) \xrightarrow{\Gamma'} \tau'$, para demostrar la igualdad $\mathcal{T}_\theta \llbracket \tau_1 \sqcap \tau_2 \rrbracket = \mathcal{T}_\theta \llbracket \tau_1 \rrbracket \cap \mathcal{T}_\theta \llbracket \tau_2 \rrbracket$, tendremos que distinguir entre los casos del resultado de realizar el ínfimo entre Γ' y Γ'':

  1. **Caso** $\theta \in \mathcal{T}_{Env} \llbracket \Gamma' \rrbracket \wedge \theta \in \mathcal{T}_{Env} \llbracket \Gamma'' \rrbracket$

     Sea F una función que pertenece a $\mathcal{T}_\theta \left\llbracket \left(\overline{\tau'_i}^n\right) \xrightarrow{\Gamma'} \tau' \right\rrbracket$ y $\mathcal{T}_\theta \left\llbracket \left(\overline{\tau''_i}^n\right) \xrightarrow{\Gamma''} \tau'' \right\rrbracket$, sabemos que F es un grafo compuesto de pares w con la forma $((v_1, \ldots, v_n), v)$, donde tenemos que:

$$\begin{array}{c} \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta \llbracket \tau'_i \rrbracket \wedge v_i \in \mathcal{T}_\theta \llbracket \tau''_i \rrbracket \quad \wedge \quad v \in \mathcal{T}_\theta \llbracket \tau' \rrbracket \wedge v \in \mathcal{T}_\theta \llbracket \tau'' \rrbracket \\ \Downarrow \qquad \{ \text{por h.i.} \} \\ \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta \llbracket \tau'_i \sqcap \tau''_i \rrbracket \quad \wedge \quad v \in \mathcal{T}_\theta \llbracket \tau' \sqcap \tau'' \rrbracket \\ \Downarrow \\ F \subseteq \left\{ ((v_1, \ldots, v_n), v) \mid \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta \llbracket \tau'_i \sqcap \tau''_i \rrbracket , v \in \mathcal{T}_\theta \llbracket \tau' \sqcap \tau'' \rrbracket \right\} \\ \Downarrow \\ F \in \mathcal{T}_\theta \left\llbracket \left(\overline{\tau'_i \sqcap \tau''_i}^n\right) \xrightarrow{\Gamma' \sqcap \Gamma''} \tau' \sqcap \tau'' \right\rrbracket \end{array}$$

     Esto último se cumple porque como tenemos $\theta \in \mathcal{T}_{Env} \llbracket \Gamma' \rrbracket$ y $\theta \in \mathcal{T}_{Env} \llbracket \Gamma'' \rrbracket$, por hipótesis de inducción obtenemos $\theta \in \mathcal{T}_{Env} \llbracket \Gamma' \sqcap \Gamma'' \rrbracket$.

     Ahora supongamos que $F \in \mathcal{T}_\theta \left\llbracket \left(\overline{\tau'_i \sqcap \tau''_i}^n\right) \xrightarrow{\Gamma' \sqcap \Gamma''} \tau' \sqcap \tau'' \right\rrbracket$. Por lo tanto F representa un grafo y sabemos que cada $w \in F$ tiene la forma $((v_1, \ldots, v_n), v)$, entonces obtenemos que:

$$\begin{array}{c} \forall w \in F \quad \text{donde} \quad \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta \llbracket \tau'_i \sqcap \tau''_i \rrbracket , v \in \mathcal{T}_\theta \llbracket \tau' \sqcap \tau'' \rrbracket \\ \Downarrow \qquad \{ \text{por h.i.} \} \\ \forall w \in F \quad \text{donde} \quad \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta \llbracket \tau'_i \rrbracket , \\ \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta \llbracket \tau''_i \rrbracket , v \in \mathcal{T}_\theta \llbracket \tau' \rrbracket , v \in \mathcal{T}_\theta \llbracket \tau'' \rrbracket \\ \Downarrow \\ \forall w \in F \quad \text{donde} \quad \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta \llbracket \tau'_i \rrbracket , v \in \mathcal{T}_\theta \llbracket \tau' \rrbracket , \\ \wedge \quad \forall w \in F \quad \text{donde} \quad \forall i \in \{1..n\} . v_i \in \mathcal{T}_\theta \llbracket \tau''_i \rrbracket , v \in \mathcal{T}_\theta \llbracket \tau'' \rrbracket \end{array}$$

Porque se cumple que $\theta \in \mathcal{T}_{Env}[\![\Gamma']\!]$ y $\theta \in \mathcal{T}_{Env}[\![\Gamma'']\!]$, tenemos que la función F pertenece a $\mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i'}^n\right) \xrightarrow{\Gamma'} \tau'\right]\!\right]$ y $\mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i''}^n\right) \xrightarrow{\Gamma''} \tau''\right]\!\right]$. Por consiguiente, obtenemos que:

$$F \in \mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i'}^n\right) \xrightarrow{\Gamma'} \tau'\right]\!\right] \sqcap \mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i''}^n\right) \xrightarrow{\Gamma''} \tau''\right]\!\right]$$

2. **Caso** $\theta \notin \mathcal{T}_{Env}[\![\Gamma']\!] \vee \theta \notin \mathcal{T}_{Env}[\![\Gamma'']\!]$

   Sabemos entonces que $\theta \notin \mathcal{T}_{Env}[\![\Gamma']\!] \cap \mathcal{T}_{Env}[\![\Gamma'']\!]$, por lo que aplicando la hipótesis de inducción tenemos que $\theta \notin \mathcal{T}_{Env}[\![\Gamma' \sqcap \Gamma'']\!]$. Por lo tanto, en este caso:

   $$\mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i'}^n\right) \xrightarrow{\Gamma'} \tau'\right]\!\right] \cap \mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i''}^n\right) \xrightarrow{\Gamma''} \tau''\right]\!\right] = \emptyset$$

   porque tenemos $\mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i'}^n\right) \xrightarrow{\Gamma'} \tau'\right]\!\right] = \emptyset$ o $\mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i''}^n\right) \xrightarrow{\Gamma''} \tau''\right]\!\right] = \emptyset$. Entonces, porque sabemos que $\theta \notin \mathcal{T}_{Env}[\![\Gamma' \sqcap \Gamma'']\!]$, obtenemos que $\mathcal{T}_\theta\left[\!\left[\left(\overline{\tau_i' \sqcap \tau_i''}^n\right) \xrightarrow{\Gamma' \sqcap \Gamma''} \tau' \sqcap \tau''\right]\!\right]$.

Para aquellos subcasos donde τ_1 no es un tipo funcional, la demostración es similar a la que hemos visto en la tupla y las listas no vacías, siendo el resultado de $\tau_1 \sqcap \tau_2 = \texttt{none()}$ salvo para el caso de `any()` o una unión de tipos.

Por último demostraremos (2) de la siguiente manera:

$$\begin{aligned}
& \theta \in \mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!] \\
\iff\ & \theta \in \mathcal{T}_{Env}[\![\Gamma_1]\!] \wedge \theta \in \mathcal{T}_{Env}[\![\Gamma_2]\!] \\
\iff\ & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_1(x)]\!] \wedge \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_2(x)]\!] \\
\iff\ & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_1(x)]\!] \cap \mathcal{T}_\theta[\![\Gamma_2(x)]\!] \\
\iff\ & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_1(x) \sqcap \Gamma_2(x)]\!] && \text{por h.i.} \\
\iff\ & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![(\Gamma_1 \sqcap \Gamma_2)(x)]\!] \\
\iff\ & \theta \in \mathcal{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!]
\end{aligned}$$

□

3.3.3. Supremo de tipos y entornos

Dados dos tipos τ y τ', usaremos la notación $\tau \sqcup \tau'$ para representar el supremo entre tipos, que definiremos como:

$$\tau \sqcup \tau' = \tau \cup \tau'$$

De forma análoga, dados dos entornos de tipos Γ y Γ', la notación $\Gamma \sqcup \Gamma'$ denota aquel entorno Γ'' tal que $\Gamma''(x) = \Gamma(x) \cup \Gamma'(x)$, para todo $x \in \mathbf{Var}$.

Teniendo en cuenta lo anterior, Se cumple la siguiente proposición para el supremo entre tipos y entornos:

Proposición 4. *Para todo $\tau_1, \tau_2, \Gamma_1, \Gamma_2$ y θ, es cierto que:*

1. $\mathcal{T}_\theta[\![\tau_1 \sqcup \tau_2]\!] = \mathcal{T}_\theta[\![\tau_1]\!] \cup \mathcal{T}_\theta[\![\tau_2]\!]$, *y por tanto* $\mathcal{T}[\![\tau_1 \sqcup \tau_2]\!] = \mathcal{T}[\![\tau_1]\!] \cup \mathcal{T}[\![\tau_2]\!]$.
2. $\mathcal{T}_{Env}[\![\Gamma_1 \sqcup \Gamma_2]\!] \supseteq \mathcal{T}_{Env}[\![\Gamma_1]\!] \cup \mathcal{T}_{Env}[\![\Gamma_2]\!]$.

Demostración. Usamos la definición de $\sqcup$ para demostrar (1):

$$\mathcal{T}_\theta[\![\tau_1 \sqcup \tau_2]\!] \quad = \quad \mathcal{T}_\theta[\![\tau_1 \cup \tau_2]\!] \quad = \quad \mathcal{T}_\theta[\![\tau_1]\!] \cup \mathcal{T}_\theta[\![\tau_2]\!]$$

Para demostrar (2), sea $\theta \in \mathcal{T}_{Env}[\![\Gamma_1]\!] \cup \mathcal{T}_{Env}[\![\Gamma_2]\!]$, tenemos dos casos posibles. Si $\theta \in \mathcal{T}_{Env}[\![\Gamma_1]\!]$:

$$\begin{array}{ll} & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_1(x)]\!] \\ \Longrightarrow & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_1(x)]\!] \cup \mathcal{T}_\theta[\![\Gamma_2(x)]\!] \\ \Longrightarrow & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_1(x) \sqcup \Gamma_2(x)]\!] \\ \Longrightarrow & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![(\Gamma_1 \sqcup \Gamma_2)(x)]\!] \\ \Longrightarrow & \theta \in \mathcal{T}_{Env}[\![\Gamma_1 \sqcup \Gamma_2]\!] \end{array}$$

Si $\theta \in \mathcal{T}_{Env}[\![\Gamma_2]\!]$, su demostración es análoga a la anterior.

Observemos que no se cumple en general que $\mathcal{T}_{Env}[\![\Gamma_1 \sqcup \Gamma_2]\!] = \mathcal{T}_{Env}[\![\Gamma_1]\!] \cup \mathcal{T}_{Env}[\![\Gamma_2]\!]$, como muestra el siguiente contraejemplo. Asumamos que $\Gamma_1 = [\mathtt{X} : \mathbb{0}, \mathtt{Y} : 1]$ y $\Gamma_2 = [\mathtt{X} : \mathtt{'a'}, \mathtt{Y} : \mathtt{'b'}]$, por lo que $\Gamma_1 \sqcup \Gamma_2$ daría el entorno $[\mathtt{X} : \mathbb{0} \cup \mathtt{'a'}, \mathtt{Y} : 1 \cup \mathtt{'b'}]$. Pero si escogemos la sustitución $\theta = [\mathtt{X}/\mathtt{'a'}, \mathtt{Y}/1]$, nos encontraremos que $\theta \in \mathcal{T}_{Env}[\![\Gamma_1 \sqcup \Gamma_2]\!]$, pero $\theta \notin \mathcal{T}_{Env}[\![\Gamma_1]\!]$ y $\theta \notin \mathcal{T}_{Env}[\![\Gamma_2]\!]$.

Sin embargo, $\sqcup$ ejerce igualmente de operador de supremo. Para demostrarlo, supongamos un Γ' tal que $\Gamma_1 \subseteq \Gamma'$, $\Gamma_2 \subseteq \Gamma'$ y $\Gamma' \subseteq \Gamma_1 \sqcup \Gamma_2$. Sea $\theta \in \mathcal{T}_{Env}[\![\Gamma_1]\!] \sqcup \mathcal{T}_{Env}[\![\Gamma_2]\!]$, entonces tenemos que:

$$\begin{array}{lll} & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![(\Gamma_1 \sqcup \Gamma_2)(x)]\!] & \\ \Longrightarrow & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_1(x)]\!] \sqcup \mathcal{T}_\theta[\![\Gamma_2(x)]\!] & \\ \Longrightarrow & \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_\theta[\![\Gamma_1(x)]\!] \cup \mathcal{T}_\theta[\![\Gamma_2(x)]\!] & \\ \Longrightarrow & \theta \in \mathcal{T}_{Env}[\![\Gamma_1]\!] \cup \mathcal{T}_{Env}[\![\Gamma_2]\!] & \\ \Longrightarrow & \theta \in \mathcal{T}_{Env}[\![\Gamma']\!] & \text{porque } \Gamma_1 \subseteq \Gamma' \text{ y } \Gamma_2 \subseteq \Gamma' \end{array}$$

demostrando que $\Gamma_1 \sqcup \Gamma_2 \subseteq \Gamma'$ y, por lo tanto $\Gamma' = \Gamma_1 \sqcup \Gamma_2$. □

3.4. Reglas para derivar tipos monomórficos

En la definición 1 hicimos uso de las nociones semánticas introducidas por nosotros para dar una definición precisa de *sucess type* de una expresión, noción que en [55] fue formulada para funciones,

estableciendo que $\tau_1 \to \tau_2$ es un *success type* para una función f si y solo si, para cada $v, v' \in \mathbf{DVal}$ tal que $f(v)$ se evalúe a v', se cumple que v está contenido en τ_1 y v' lo están en τ_2; dicho de otra manera, será un *success type* si el grafo de la función denotada por f está contenida en la semántica de $\tau_1 \to \tau_2$. En [66] se generalizó esta noción para poder aplicarla también a expresiones cerradas. Todas estas nociones se corresponden con la idea intuitiva de que un tipo τ es un *success type* de la expresión e, si contempla todos los valores posibles a los que se puede evaluar e.

Sin embargo, si queremos derivar un *success type* para una expresión con variables libres, además de derivar del tipo para la expresión en sí misma, el sistema de tipos ha de derivar también información sobre las condiciones necesarias sobre dichas variables libres. Por ejemplo, es una condición necesaria (aunque no suficiente) para la evaluación satisfactoria de `X/Y`, que tanto `X` como `Y` sean números. Esta condición la expresamos con el entorno de tipos [`X : number(), Y : number()`].

Con las reglas de tipado mostradas en esta sección podemos obtener, para cada expresión e, un tipo y un entorno, para expresar las condiciones necesarias para la posterior evaluación de e. Además, es recomendable añadir a nuestros juicios de tipado otro entorno, para poder disponer de información supuestamente ya conocida sobre las variables libres de e. Por lo tanto, nuestros juicios de tipado tendrán la forma $\Gamma \vdash e : \tau, \Gamma'$, con el siguiente significado: suponiendo que las variables libres de e tienen valores contenidos en los tipos almacenados en Γ, si e es evaluada al valor v, entonces v es del tipo τ y las variables libres de e tienen valores contenidos en los tipos almacenados en Γ'. Esta intuición la podemos expresar de la siguiente manera:

$$\forall \theta \in \mathbf{Subst}, v \in \mathbf{DVal} : \theta \in \mathscr{T}_{Env}[\![\Gamma]\!] \wedge (\theta, v) \in \mathscr{E}[\![e]\!] \Rightarrow (\theta, v) \in \mathscr{T}[\![\tau]\!] \wedge \theta \in \mathscr{T}_{Env}[\![\Gamma']\!]$$

la cual también podemos escribir de forma más sucinta como $\mathscr{E}[\![e]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \subseteq \mathscr{T}[\![\tau]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$. En adelante utilizaremos el término *entorno de entrada* y *entorno final* para referirnos a Γ y Γ' respectivamente. En la figura 3.2 encontramos las reglas de tipado y las vamos a explicar brevemente a continuación.

La regla [SUB-1] especifica que podemos reemplazar un entorno de entrada Γ_1 por uno más restrictivo, mientras que las reglas [SUB-2] y [SUB-T] permiten relajar el tipo τ y el entorno final Γ_2 respectivamente.

La regla [NONE] permite inferir el fallo—al evaluar una expresión—en el caso de que una variable cualquiera en el entorno de entrada tuviera el tipo `none()`. La regla [TRANS] indica que, cuando sea que tengamos un juicio $\Gamma_1 \vdash e : \tau, \Gamma_2$, podremos analizar de nuevo el tipo para e tomando como entorno de entrada el recibido en Γ_2. Esto permite una mayor posibilidad de inferir tipos más precisos para algunas expresiones, como la que podemos ver explicada en la sección 3.5.1.

Las reglas [CONST] y [VAR] indican que el entorno final no posee restricciones adicionales, más allá de aquellas encontradas en el entorno de entrada, mientras que [TUPLE] y [LIST] toman el ínfimo de todos los entornos finales de cada subexpresión, ya que todos y cada uno de ellos deben ser tenidos en cuenta para evaluar la expresión completa. Con respecto a la regla [ABS], el entorno final es el mismo que el entorno de entrada; ello se debe a que la evaluación de una λ-abstracción siempre

$$\frac{\Gamma_1 \vdash e : \tau, \Gamma_2 \qquad \Gamma_1' \subseteq \Gamma_1}{\Gamma_1' \vdash e : \tau, \Gamma_2} \text{ [SUB-1]} \qquad \frac{\Gamma_1 \vdash e : \tau, \Gamma_2 \qquad \Gamma_2 \subseteq \Gamma_2'}{\Gamma_1 \vdash e : \tau, \Gamma_2'} \text{ [SUB-2]}$$

$$\frac{\Gamma_1 \vdash e : \tau, \Gamma_2 \qquad \tau \subseteq \tau'}{\Gamma_1 \vdash e : \tau', \Gamma_2} \text{ [SUB-T]} \qquad \frac{}{\Gamma_1 \sqcap [x : \texttt{none()}] \vdash e : \texttt{none()}, \bot} \text{ [NONE]}$$

$$\frac{\Gamma_1 \vdash e : \tau, \Gamma_2 \qquad \Gamma_2 \vdash e : \tau', \Gamma_3}{\Gamma_1 \vdash e : \tau', \Gamma_3} \text{ [TRANS]} \qquad \frac{}{\Gamma \vdash c : c, \Gamma} \text{ [CONST]} \qquad \frac{}{\Gamma \vdash x : \Gamma(x), \Gamma} \text{ [VAR]}$$

$$\frac{\Gamma \vdash e_i : \tau_i, \Gamma_i}{\Gamma \vdash \{\overline{e_i}^n\} : \{\overline{\tau_i}^n\}, \bigsqcap_{i=1}^n \Gamma_i} \text{ [TUPLE]} \qquad \frac{\Gamma \vdash e_1 : \tau_1, \Gamma_1 \qquad \Gamma \vdash e_2 : \tau_2, \Gamma_2}{\Gamma \vdash [e_1 \mid e_2] : \texttt{nelist}(\tau_1, \tau_2), \Gamma_1 \sqcap \Gamma_2} \text{ [LIST]}$$

$$\frac{\Gamma \vdash e : \tau, \Gamma'}{\Gamma \vdash \textbf{fun}(x_1, \ldots, x_n) \to e : \left(\left(\Gamma'(x_1), \ldots, \Gamma'(x_n) \right) \xrightarrow{\Gamma'} \tau \right) \setminus \{x_1, \ldots, x_n\}, \Gamma} \text{ [ABS]}$$

$$\frac{\Gamma(f) \sqcap \left(\left(\overline{\texttt{any()}}^n \right) \xrightarrow{[]} \texttt{any()} \right) = (\tau_1, \ldots, \tau_n) \xrightarrow{\Gamma'} \tau}{\Gamma \vdash f(x_1, \ldots, x_n) : \tau, \Gamma \sqcap \left[f : \left(\overline{\texttt{any()}}^n \right) \xrightarrow{[]} \texttt{any()}, x_1 : \tau_1, \ldots, x_n : \tau_n \right] \sqcap \Gamma'} \text{ [APP-1]}$$

$$\frac{\Gamma(f) \sqcap \left(\left(\overline{\texttt{any()}}^n \right) \xrightarrow{[]} \texttt{any()} \right) = \texttt{none()}}{\Gamma \vdash f(x_1, \ldots, x_n) : \texttt{none()}, \bot} \text{ [APP-2]}$$

$$\frac{\Gamma \vdash e_1 : \tau_1, \Gamma_1 \qquad \Gamma_1 \sqcap [x : \tau_1] \vdash e_2 : \tau_2, \Gamma_2}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2 \setminus x, \Gamma_2 \setminus x} \text{ [LET]}$$

$$\frac{\begin{array}{c} cls_i = \left(p_i \textbf{ when } e_i' \to e_i'' \right) \\ \Gamma \Vdash_{\{x\}} cls_i : \tau_i, \Gamma_i \qquad \tau = \tau_i \setminus vars\left(p_i\right) \qquad \Gamma' = \Gamma_i \setminus vars(p_i) \end{array}}{\Gamma \vdash \textbf{case } x \textbf{ of } cls_1 \ldots cls_n \textbf{ end} : \tau, \Gamma'} \text{ [CASE]}$$

$$\frac{\begin{array}{c} cls_i = \left(p_i \textbf{ when } e_i' \to e_i'' \right) \qquad \Gamma \vdash x_t : \tau_t, \Gamma_t \\ \Gamma_t \vdash e : \tau, \Gamma' \qquad \tau_t \sqcap \texttt{number()} \neq \texttt{none()} \qquad \tau_t \sqcap \texttt{'infinity'} \neq \texttt{none()} \\ \Gamma_t \Vdash_{\emptyset} cls_i : \tau_i, \Gamma_i \qquad \tau = \tau_i \setminus vars\left(p_i\right) \qquad \Gamma' = \Gamma_i \setminus vars\left(p_i\right) \end{array}}{\Gamma \vdash \textbf{receive } cls_1 \ldots cls_n \textbf{ after } x_t \to e : \tau, \Gamma'} \text{ [RECEIVE-1]}$$

$$\frac{\begin{array}{c} cls_i = \left(p_i \textbf{ when } e_i' \to e_i'' \right) \qquad \Gamma \vdash x_t : \tau_t, \Gamma_t \\ \tau_t \sqcap \texttt{number()} = \texttt{none()} \qquad \tau_t \sqcap \texttt{'infinity'} \neq \texttt{none()} \\ \Gamma_t \Vdash_{\emptyset} cls_i : \tau_i, \Gamma_i \qquad \tau = \tau_i \setminus vars\left(p_i\right) \qquad \Gamma' = \Gamma_i \setminus vars\left(p_i\right) \end{array}}{\Gamma \vdash \textbf{receive } cls_1 \ldots cls_n \textbf{ after } x_t \to e : \tau, \Gamma'} \text{ [RECEIVE-2]}$$

$$\frac{\Gamma \vdash x_t : \tau_t, \Gamma_t \qquad \tau_t \sqcap (\texttt{number()} \cup \texttt{'infinity'}) = \texttt{none()}}{\Gamma \vdash \textbf{receive } cls_1 \ldots cls_n \textbf{ after } x_t \to e : \texttt{none()}, \bot} \text{ [RECEIVE-3]}$$

$$\frac{\Gamma \sqcap \left[\overline{x_j : \tau_j}^n \right] \vdash e_i : \tau_i', \Gamma_i' \qquad \tau_i' \setminus \{\overline{x_j}^n\} = \tau_i \qquad \bigsqcap_{i=1}^n \left(\Gamma_i' \setminus \{\overline{x_j}^n\} \right) \sqcap \left[\overline{x_j : \tau_j}^n \right] \vdash e : \tau, \Gamma'}{\Gamma \vdash \textbf{letrec } \overline{x_i = e_i}^n \textbf{ in } e : \tau \setminus \{\overline{x_j}^n\}, \Gamma' \setminus \{\overline{x_j}^n\}} \text{ [LETREC]}$$

Figura 3.2: Reglas de tipado para expresiones

$$\frac{\Gamma \vdash p : \tau_p, \Gamma_p \quad \Gamma_p \sqcap [X : \tau_p] \vdash e_g : \tau_g, \Gamma_g \quad \tau_g \sqcap \texttt{'true'} \neq \texttt{none()} \quad \Gamma_g \vdash e : \tau', \Gamma'}{\Gamma \Vdash_X p \textbf{ when } e_g \rightarrow e : \tau', \Gamma'} \text{ [CLS-TRUE]}$$

$$\frac{\Gamma \vdash p : \tau_p, \Gamma_p \quad \Gamma_p \sqcap [X : \tau_p] \vdash e_g : \tau_g, \Gamma_g \quad \tau_g \sqcap \texttt{'true'} = \texttt{none()}}{\Gamma \Vdash_X p \textbf{ when } e_g \rightarrow e : \texttt{none()}, \bot} \text{ [CLS-FALSE]}$$

$$\frac{\Gamma_1 \Vdash_X cls : \tau, \Gamma_2 \qquad \Gamma_2 \Vdash_X cls : \tau', \Gamma_3}{\Gamma_1 \Vdash_X cls : \tau', \Gamma_3} \text{ [CLS-TRANS]}$$

Figura 3.3: Reglas de tipado para cláusulas

tiene éxito. Las restricciones expresadas en el entorno Γ', correspondiente al tipado del cuerpo de la λ-abstracción, que son el resultado de analizar el cuerpo de la función, son solo relevantes a la hora de aplicar la función, por ello estas restricciones en forma de entorno están situadas encima de la flecha del tipo funcional obtenido.

El sistema tiene dos reglas para la aplicación de funciones: [APP-1] tendrá sentido cuando el tipo asumido para f sea compatible con un tipo funcional, en el caso contrario [APP-2] indicará que la evaluación de la expresión ha fallado. En el primer caso, el entorno final exige que los valores pasados como argumentos han de estar contenidos en los correspondientes tipos $\tau_1, \ldots, \tau_n$. También exigiremos—para la ejecución de la función—que las restricciones adicionales, contenidas en Γ', que existan sobre las variables libres sean añadidas al entorno final.

La regla [LET] es bastante estándar. La única diferencia es que eliminaremos la restricción que acompaña a la variable x del tipo y del entorno final, ya que dicha variable no estará libre en la expresión **let**. Para cada juicio $\Gamma \vdash e : \tau, \Gamma'$, las variables ligadas en e deberán tener el mismo tipo en Γ y Γ', pues la ejecución de e tendrá éxito o no independientemente de su valor.

De cara a derivar tipos para una expresión **case** o **receive**, hemos de derivar un tipo para cada una de sus cláusulas. En la figura 3.3 encontramos las reglas de tipado para las cláusulas. En estas reglas obtenemos juicios de la forma $\Gamma \Vdash_X cls : \tau, \Gamma'$, donde X puede ser un conjunto unitario (conteniendo la variable discriminante usada en una expresión **case**) o un conjunto vacío (en el caso de las expresiones **receive**). La notación $[X : \tau]$ denota un entorno donde asignamos el tipo τ a cada variable en X y al resto de variables se les asigna `any()`. La regla [CLS-TRUE] maneja aquellos casos en los que el tipo del discriminante es compatible con el tipo del patrón y el tipo de la guarda es compatible con el átomo `'true'`. Para el resto de casos utilizaremos la regla [CLS-FALSE]. La regla [CLS-TRANS] es el equivalente de la regla [TRANS] pero para cláusulas.

Habiendo derivado los juicios de tipos relativos a cada cláusula en un **case** o en un **receive**, la regla [CASE] toma el tipo y el entorno final de cada cláusula, eliminando las restricciones asociadas a las variables del patrón, ya que estas variables no estarán ya libres. La regla [RECEIVE-2] se encarga de

aquellos casos en los que el tipo para la expresión e_t no contiene el átomo `'infinity'`. En este caso descartaremos el tipo de la expresión que nos encontramos en la cláusula del **after**. La regla [RECEIVE-3] se encarga de aquellos casos en los que no podemos evaluar la expresión e_t a un número dentro de `integer()` o al átomo `'infinity'`. En este caso la evaluación del **receive** siempre falla. Para el resto de casos utilizaremos la regla [RECEIVE-1].

La regla [LETREC] es similar a la regla [LET]. La diferencia principal estriba en que para poder emitir un juicio para cada subexpresión e_i, deberemos restringir en el entorno de entrada de estos juicios las variables $x_1,\ldots,x_n$ con los tipos $\tau_1,\ldots,\tau_n$, para obtener una serie de tipos $\tau'_1,\ldots,\tau'_n$. Cada tipo τ'_i está condicionado a tener que ser igual al tipo τ_i, una vez hayamos eliminado las restricciones sobre todas las variables x_i definidas por el **letrec** en cada tipo τ'_i.

Una vez explicadas las reglas del sistema de tipos, el siguiente resultado indica que siempre podemos encontrar una derivación de tipo para una expresión dada:

Proposición 5. *Dada una expresión e cualquiera y un entorno inicial* Γ*, existe un tipo* τ *y un entorno* Γ' *tal que* $\Gamma \vdash e : \tau, \Gamma'$*. En particular:* $\Gamma \vdash e : \texttt{any()}, []$.

Demostración. Sencilla, inspeccionando las reglas de tipado. Las condiciones secundarias que involucran la relación de inclusión $\subseteq$, entre entornos o tipos, siempre pueden satisfacer eligiendo $[]$ y `any()` respectivamente en el lado derecho de dichas condiciones. Si la condición secundaria de [APP-1] que pide a f ser un tipo funcional no es cierta, entonces la regla [APP-2] puede ser aplicada. De forma análoga tenemos [RECEIVE-1], [RECEIVE-2] y [RECEIVE-3], que se complementan con la condición impuesta sobre τ_t. Luego si la condición secundaria de [CLS-TRUE] que pide a las guardas ser `'true'` no se cumple, se aplicará la regla [CLS-FALSE]. El resto de condiciones secundarias, que involucran tipos o entornos, pueden ser modificadas usando [SUB-T] y [SUB-2] para construir un tipo o un entorno más grande si fuera necesario.

Una vez que demostramos que hemos derivado el juicio $\Gamma \vdash e : \tau, \Gamma'$ para algún τ y Γ', dado que $\tau \subseteq$ `any()` y $\Gamma' \subseteq []$, podemos usar las reglas [SUB-T] y [SUB-2] para obtener $\Gamma \vdash e : \texttt{any()}, []$. □

3.5. Ejemplos de derivación de tipos monomórficos

En los ejemplos abordados en esta sección no mostraremos el uso de las reglas [CONST] y [VAR], ya que su uso es trivial y así evitaremos que el tamaño de los ejemplos sea excesivo.

3.5.1. Usando la regla [TRANS]

El primer ejemplo es una función que toma un número y devuelve una tupla con el número dado y su sucesor. Usamos `'+'`[1] para denotar la función predefinida de la suma, cuyo tipo estará contenido

[1] En *Erlang* los operadores del lenguaje están dentro del módulo `erlang`. La suma en concreto es la función `erlang:'+'/2`.

en el entorno de entrada $\Gamma_0 = [\texttt{'+'} : (\texttt{number()}, \texttt{number()}) \to \texttt{number()}]$. El código en *Erlang* es el siguiente:

```
fun(V) -> {V, V + 1} end
```

Que traducido a *Mini Erlang* sería lo siguiente:

$$\textbf{fun}(\texttt{V0}) \to \textbf{let}\ \texttt{V1} = 1\ \textbf{in}\ \{\texttt{V0}, \texttt{'+'}(\texttt{V0}, \texttt{V1})\}$$

En la siguiente derivación obtenemos el tipo $(\texttt{number()}) \to \{\texttt{number()}, \texttt{number()}\}$ para esta λ-abstracción, aplicando las siguientes reglas:

$$\frac{\begin{array}{c}\Gamma_0 = [\texttt{'+'} : (\texttt{number()}, \texttt{number()}) \xrightarrow{[]} \texttt{number()}] \\ \Gamma_0(\texttt{'+'}) \sqcap ((\texttt{any()}, \texttt{any()}) \xrightarrow{[]} \texttt{any()}) = (\texttt{number()}, \texttt{number()}) \xrightarrow{[]} \texttt{number()}\end{array}}{\Gamma_0 \vdash \texttt{'+'}(\texttt{V0}, \texttt{V1}) : \texttt{number()}, (\Gamma_1 = \Gamma_0 \sqcap [\texttt{V0} : \texttt{number()}; \texttt{V1} : \texttt{number()}])} \quad [\text{APP-1}]$$

$$\frac{\Gamma_0 \vdash \texttt{V0} : \texttt{any()}, \Gamma_0 \qquad \Gamma_0 \vdash \texttt{'+'}(\texttt{V0}, \texttt{V1}) : \texttt{number()}, \Gamma_1}{\Gamma_0 \vdash \{\texttt{V0}, \texttt{'+'}(\texttt{V0}, \texttt{V1})\} : \{\texttt{any()}, \texttt{number()}\}, \Gamma_1} \quad [\text{TUPLE}]$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash \{\texttt{V0}, \texttt{'+'}(\texttt{V0}, \texttt{V1})\} : \{\texttt{any()}, \texttt{number()}\}, \Gamma_1 \\ \Gamma_1 \vdash \{\texttt{V0}, \texttt{'+'}(\texttt{V0}, \texttt{V1})\} : \{\texttt{number()}, \texttt{number()}\}, \Gamma_1\end{array}}{\Gamma_0 \vdash \{\texttt{V0}, \texttt{'+'}(\texttt{V0}, \texttt{V1})\} : \{\texttt{number()}, \texttt{number()}\}, \Gamma_1} \quad [\text{TRANS}]$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash \texttt{1} : 1, \Gamma_0 \qquad \Gamma_0 \sqcap [\texttt{V1} : 1] \vdash \{\texttt{V0}, \texttt{'+'}(\texttt{V0}, \texttt{V1})\} : \\ \{\texttt{number()}, \texttt{number()}\}, \Gamma_0 \sqcap [\texttt{V0} : \texttt{number()}; \texttt{V1} : 1]\end{array}}{\Gamma_0 \vdash \textbf{let}\ \texttt{V1} = 1\ \textbf{in}\ \{\texttt{V0}, \texttt{'+'}(\texttt{V0}, \texttt{V1})\} : \{\texttt{number()}, \texttt{number()}\}, \Gamma_0 \sqcap [\texttt{V0} : \texttt{number()}]} \quad [\text{LET}]$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash \textbf{let}\ \texttt{V1} = 1\ \textbf{in}\ \{\texttt{V0}, \texttt{'+'}(\texttt{V0}, \texttt{V1})\} : \{\texttt{number()}, \texttt{number()}\}, \Gamma_0 \sqcap [\texttt{V0} : \texttt{number()}] \\ (\Gamma_0 \sqcap [\texttt{V0} : \texttt{number()}]) \setminus \{\texttt{V0}\} = \Gamma_0 \\ (\{\texttt{number()}, \texttt{number()}\}) \setminus \{\texttt{V0}\} = (\{\texttt{number()}, \texttt{number()}\})\end{array}}{\Gamma_0 \vdash \textbf{fun}(\texttt{V0}) \to \textbf{let}\ \texttt{V1} = 1\ \textbf{in}\ \cdots : (\texttt{number()}) \xrightarrow{\Gamma_0} \{\texttt{number()}, \texttt{number()}\}, \Gamma_0} \quad [\text{ABS}]$$

En ocasiones un árbol de derivación puede contener varias subderivaciones para la misma expresión de programa, cada una de ellas con un entorno de entrada diferente. Esto se debe a la necesidad de obtener tipos cada vez más refinados usando la regla [TRANS] con dicha expresión. Por ejemplo, la aplicación de las reglas [LIST] y [APP-1] mostradas arriba requieren ser analizadas de nuevo para derivar nuevos juicios con diferentes entornos de entrada.

La regla [TRANS] es usada en esta derivación para actualizar el tipo final de una expresión cuando obtenemos un nuevo entorno. En el ejemplo, cuando analizamos `V0` bajo Γ_0 recibimos `any()` como tipo para la variable, pero después de analizar `'+'(V0,V1)` obtenemos una nueva información—sobre la misma variable—indicando que su tipo es `number()`. Sin la regla [TRANS], el tipo final hubiera sido $(\texttt{number()}) \to \{\texttt{any()}, \texttt{number()}\}$, pero al disponer de esta regla podemos volver a analizar la expresión `{V0,'+'(V0,V1)}` bajo un entorno de entrada en el que tenemos conocimiento de que `V0` es

de tipo `number()`.

3.5.2. Derivando un tipo para la función `map`

En el segundo ejemplo de derivación utilizaremos la función `map` con una función aritmética que toma un número y lo multiplica por dos. Usamos `'*'` para denotar la función predefinida de la multiplicación, cuyo tipo `(number(),number()) → number()` estará contenido en el entorno de entrada Γ_0. El código en *Erlang* es el siguiente:

```
map(_, []) -> [];
map(F, [X|XS]) -> [F(X)|map(F,XS)].

foo(L) -> map(fun(N) -> N * 2 end, L).
```

Que traducimos a *Mini Erlang* de la siguiente manera:

```
letrec Map = fun(V1,V2) →
              case V2 of
                  [] when 'true' → []
                  [X|XS] when 'true' →
                      let V3 = V1(X) in
                      let V4 = Map(V1,XS) in [V3|V4]
              end
    Foo = fun(V0) →
              let V5 = fun(V6) → let V7 = 2 in '*'(V6,V7)
              in Map(V5,V0)
in Foo
```

Para la λ-abstracción `Foo` obtenemos el tipo `([any()]) → [any()]` en la derivación realizada para el ejemplo. La derivación completa del ejemplo la encontramos en el apéndice A, pero en esta sección mostramos un fragmento de la derivación con las reglas usadas más relevantes:

$$\dfrac{\begin{array}{c}\Gamma_0 = [\texttt{'*'} : (\texttt{number()},\texttt{number()}) \xrightarrow{[]} \texttt{number()}] \\ \Gamma_1 = \Gamma_0 \sqcap [\texttt{Map} : (\texttt{any()},\texttt{[any()]}) \xrightarrow{\Gamma_0} \texttt{[any()]}; \texttt{Foo} : (\texttt{[any()]}) \xrightarrow{\Gamma_0} \texttt{[any()]}] \\ (1)\quad \Gamma_1 \vdash \textbf{fun}(\texttt{V1},\texttt{V2}) \rightarrow \cdots : (\texttt{any()},\texttt{[any()]}) \xrightarrow{\Gamma_1} \texttt{[any()]},\Gamma_1 \\ (2)\quad \Gamma_1 \vdash \textbf{fun}(\texttt{V0}) \rightarrow \cdots : (\texttt{[any()]}) \xrightarrow{\Gamma_1} \texttt{[any()]},\Gamma_1 \\ \Gamma_1 \vdash \texttt{Foo} : (\texttt{[any()]}) \xrightarrow{[]} \texttt{[any()]},\Gamma_1\end{array}}{\Gamma_0 \vdash \textbf{letrec}\ \texttt{Map} = \ldots \texttt{Foo} = \ldots \textbf{in} \cdots : (\texttt{[any()]}) \xrightarrow{\Gamma_0} \texttt{[any()]},\Gamma_0}\ [\text{LETREC}]$$

$$\frac{\begin{array}{c}(3)\quad \Gamma_1 \vdash \textbf{case}\ \texttt{V2}\ \textbf{of}\ldots\textbf{end} : \texttt{[any()]}, (\Gamma_2 = \Gamma_1 \sqcap \texttt{[V1 : any(); V2 : [any()]]}) \\ \Gamma_2 \setminus \{\texttt{V1}, \texttt{V2}\} = \Gamma_1 \qquad \texttt{[any()]} \setminus \{\texttt{V1}, \texttt{V2}\} = \texttt{[any()]} \\ (1)\quad \Gamma_1 \vdash \textbf{fun}(\texttt{V1}, \texttt{V2}) \rightarrow \cdots : (\texttt{any()}, \texttt{[any()]}) \xrightarrow{\Gamma_1} \texttt{[any()]}, \Gamma_1\end{array}}{\begin{array}{c}\Gamma_3 = \Gamma_1 \sqcap \texttt{[V1 : any(); V2 : []]} \\ \Gamma_4 = \Gamma_1 \sqcap \texttt{[V1 : (any())} \xrightarrow{[]} \texttt{any(); V2 : [any()]; X : any(); XS : [any()]]} \\ (4)\quad \Gamma_1 \Vdash_{\{\texttt{V2}\}} \texttt{[]}\ \textbf{when}\ \texttt{'true'} \rightarrow \texttt{[]} : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4 \\ (5)\quad \Gamma_1 \Vdash_{\{\texttt{V2}\}} \texttt{[X|XS]}\ \textbf{when}\ \texttt{'true'} \rightarrow \cdots : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4 \\ (3)\quad \Gamma_1 \vdash \textbf{case}\ \texttt{V2}\ \textbf{of}\ldots\textbf{end} : \texttt{[any()]}, \Gamma_2\end{array}}\ [\text{CASE}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \texttt{[]} : \texttt{[]}, \Gamma_1 \qquad \Gamma_1 \sqcap \texttt{[V2 : []]} \vdash \texttt{'true'} : \texttt{'true'}, \Gamma_3 \\ \texttt{'true'} \sqcap true \neq \text{none}() \qquad \Gamma_3 \vdash \texttt{[]} : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4 \\ (4)\quad \Gamma_1 \Vdash_{\{\texttt{V2}\}} \texttt{[]}\ \textbf{when}\ \texttt{'true'} \rightarrow \texttt{[]} : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4\end{array}}\ [\text{CLS-T}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \texttt{[X|XS]} : \texttt{[any()|any()]}, \Gamma_1 \\ (\Gamma_1 \sqcap \texttt{[V2 : [any()|any()]]} = \Gamma_5) \vdash \texttt{'true'} : \texttt{'true'}, \Gamma_5 \\ \texttt{'true'} \sqcap true \neq \text{none}() \qquad \Gamma_5 \vdash \textbf{let}\ \texttt{V3} = \ldots\textbf{in}\cdots : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4 \quad (6) \\ (5)\quad \Gamma_1 \Vdash_{\{\texttt{V2}\}} \texttt{[X|XS]}\ \textbf{when}\ \texttt{'true'} \rightarrow \cdots : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4\end{array}}\ [\text{CLS-T}]$$

$$\vdots$$

$$\frac{\begin{array}{c}(11)\quad \Gamma_1 \vdash \textbf{let}\ \texttt{V5} = \ldots\textbf{in}\cdots : \texttt{[any()]}, (\Gamma_9 = \Gamma_1 \sqcap \texttt{[V0 : [any()]]}) \\ \Gamma_9 \setminus \{\texttt{V0}\} = \Gamma_1 \qquad \texttt{[any()]} \setminus \{\texttt{V0}\} = \texttt{[any()]} \\ (2)\quad \Gamma_1 \vdash \textbf{fun}(\texttt{V0}) \rightarrow \cdots : (\texttt{[any()]}) \xrightarrow{\Gamma_1} \texttt{[any()]}, \Gamma_1\end{array}}\ [\text{ABS}]$$

$$\frac{\begin{array}{c}(12)\quad \Gamma_1 \vdash \textbf{fun}(\texttt{V6}) \rightarrow \cdots : (\texttt{number()}) \xrightarrow{\Gamma_1} \texttt{number()}, \Gamma_1 \\ (13)\quad (\Gamma_1 \sqcap \texttt{[V5 : (number()}) \xrightarrow{\Gamma_1} \texttt{number()]} = \Gamma_{10}) \vdash \texttt{Map(V5, V0)} : \\ \texttt{[any()]}, (\Gamma_{11} = \Gamma_9 \sqcap \texttt{[V5 : (number()}) \xrightarrow{\Gamma_1} \texttt{number()]}) \\ (11)\quad \Gamma_1 \vdash \textbf{let}\ \texttt{V5} = \ldots\textbf{in}\cdots : \texttt{[any()]}, \Gamma_9\end{array}}\ [\text{LET}]$$

$$\vdots$$

$$\frac{\Gamma_{10}(\texttt{Map}) \sqcap ((\texttt{any()}, \texttt{any()}) \xrightarrow{[]} \texttt{any()}) = (\texttt{any()}, \texttt{[any()]}) \xrightarrow{\Gamma_0} \texttt{[any()]}}{(13)\quad \Gamma_{10} \vdash \texttt{Map(V5, V0)} : \texttt{[any()]}, \Gamma_{11}}\ [\text{APP-1}]$$

En el ejemplo mostramos la derivación desde lo más alto de la expresión hacia la más baja. Muchos de los juicios de la derivación están identificados con un número, con esto buscamos hacer más sencillo localizar en el ejemplo cada parte a lo largo de la derivación. En la regla [CASE], del juicio número (3), podemos ver cómo Γ_3 es el entorno obtenido de la primera cláusula y Γ_4 el obtenido de la segunda, pero para poder aplicar la regla debemos unificar los entornos finales y los tipos. Para hacer dicha unificación posible, usamos las reglas [SUB-2] y [SUB-T] sobre el cuerpo de cada cláusula, para lograr obtener el mismo tipo y entorno en ambas cláusulas.

3.5.3. Mejorando los success types obtenidos

En un sistema de tipos monomórfico, los *success types* que podemos obtener de funciones como `Map` distan de ser los que podríamos desear. La causa de estos resultados, poco exactos, es que los tipos de entrada no están conectados con los tipos de salida en el tipo de la función. Para mejorar esta situación hay varias soluciones. Una de ellas, consiste en volver a analizar el tipo de la definición de una función para cada una de sus aplicaciones. En cada uno de esos análisis, inyectaríamos los tipos de los parámetros de entrada—de la llamada en cuestión—en el entorno de entrada.

Cada vez que usamos la regla [APP-1] para analizar la aplicación de una función, podemos intentar volver a derivar el tipo de la definición de la función, siempre que tengamos su código. En esta nueva derivación, el entorno de entrada enlaza los parámetros de entrada a los tipos de los argumentos actuales usados en la aplicación de la función. Así obtenemos una signatura más precisa para esa función. Con el ejemplo de `Map` de la sección anterior, el primer análisis de `Map` devolvería un entorno final $\Gamma_2 = \Gamma_1 \sqcap [\texttt{V1} : \texttt{any()}, \texttt{V2} : \texttt{[any()]}]$. Pero ese análisis, dentro de la definición de `Foo`, tendría que la función `Map` va a ser aplicada con `V5` (de tipo $(\texttt{number()}) \xrightarrow{[]} \texttt{number()}$) y `V0` (de tipo `any()`), así que volveríamos a analizar el cuerpo de la definición de `Map` bajo un entorno de entrada $\Gamma_2 \sqcap [\texttt{V1} : (\texttt{number()}) \xrightarrow{[]} \texttt{number()}, \texttt{V2} : \texttt{any()}]$, lo que llevaría de forma específica para esta aplicación a obtener la signatura $((\texttt{number()}) \rightarrow \texttt{number()}, \texttt{[number()]}) \rightarrow \texttt{[number()]}$ como tipo de `Map`. Entonces podemos actualizar el tipo de la aplicación `Map(V5,V0)` para inferir que `V0` debe ser de tipo `[number()]`.

Este enfoque, que hasta cierto punto sería similar a realizar un *inlining* de una definición de función para cada aplicación, cambia el tipo final de la función `Foo` al tipo $(\texttt{[number()]}) \rightarrow \texttt{[number()]}$, en lugar de $(\texttt{[any()]}) \rightarrow \texttt{[any()]}$ que era el *success type* alcanzado en la derivación de la sección anterior. La ventaja de este refinamiento es que alcanzamos *success types* más precisos a la hora de usar funciones polimórficas, pero su mayor desventaja es que su coste computacional incrementa considerablemente, ya que hemos de volver a derivar un tipo para la función en cada una de sus aplicaciones, lo cual implica a su vez tener que derivar tipos para todas las funciones que sean aplicadas en su interior nuevamente siguiendo un efecto dominó.

Aunque esta técnica podría ofrecer resultados más precisos para un sistema monomórfico, en ningún momento se consideró realizar su implementación. No se hizo porque nunca fue el objetivo de la tesis desarrollar una herramienta de inferencia de tipos monomórficos y, de haberse planteado hacer semejante herramienta, el coste de aplicar esta mejora en la inferencia lo hace desaconsejable. Por lo tanto, resulta una opción mejor considerar el uso de tipos polimórficos, ya que también permite conectar los tipos de entrada en los parámetros a los tipos de la salida del resultado. Como podremos ver en el capítulo 4, usando tipos polimórficos logramos un enfoque más modular sin una sobrecarga computacional como la expuesta en esta sección.

3.6. Resultados de corrección

Antes de abordar la demostración de que el sistema de tipos es correcto—en el sentido de los *success types*, es decir, que los tipos que se derivan para una expresión expresan en efecto sobreaproximaciones a la semántica—daremos algunos resultados y definiciones auxiliares que serán necesitados para ello. Primero, como ya mencionamos en la sección 3.2, una variable está *sin restringir*—en un entorno de tipos Γ—si su tipo es `any()` en Γ y está sin restringir recursivamente en los entornos que existan en los tipos funcionales existentes en Γ. Esto quiere decir que x está sin restringir en Γ si y solo si $\Gamma \setminus x = \Gamma$. De forma similar, diremos que x está sin restringir en τ si y solo si $\tau \setminus x = \tau$.

La primera proposición determina que, aunque la semántica de un tipo τ dependa de una sustitución θ, las variables sin restringir en τ son irrelevantes en esta sustitución cuando determinamos la semántica de τ.

Proposición 6. *Dada una sustitución θ, un tipo τ, un entorno Γ y un conjunto finito de pares (x_1, v_1), ..., (x_n, v_n), tal que las variables $x_1, \ldots, x_n$ están sin restringir en Γ y τ, se tiene que:*

1. $\mathscr{T}_\theta [\![\tau]\!] = \mathscr{T}_{\theta\left[\overline{x_i / v_i}\right]} [\![\tau]\!]$.

2. $\theta \in \mathscr{T}_{Env} [\![\Gamma]\!] \Leftrightarrow \theta \left[\overline{x_i / v_i}\right] \in \mathscr{T}_{Env} [\![\Gamma]\!]$.

Demostración. Por inducción sobre la estructura de τ y Γ. Los casos en los que τ es `none()`, `any()`, un tipo básico o un tipo unitario, son triviales ya que la sustitución θ no juega ningún papel en la semántica de estos tipos. Si τ es un tipo tupla, un tipo lista o una unión de tipos, la proposición se cumple directamente tras aplicar la hipótesis de inducción.

En el caso de que τ sea un tipo funcional, de la forma $\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma} \tau'$, asumiremos que $\theta \notin \mathscr{T}_{Env} [\![\Gamma]\!]$. Entonces, por hipótesis de inducción, obtenemos que $\theta \left[\overline{x_i / v_i}\right] \notin \mathscr{T}_{Env} [\![\Gamma]\!]$, por lo que $\mathscr{T}_\theta \left[\!\left[\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma} \tau' \right]\!\right] = \{\emptyset\} = \mathscr{T}_{\theta\left[\overline{x_i / v_i}\right]} \left[\!\left[\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma} \tau' \right]\!\right]$. Por otro lado, si θ pertenece a $\mathscr{T}_{Env} [\![\Gamma]\!]$, entonces también lo hará $\theta \left[\overline{x_i / v_i}\right]$ y la proposición se cumple aplicando la hipótesis de inducción a cada componente $\tau_1, \ldots, \tau_n, \tau'$ del tipo funcional.

En el caso de los entornos de tipos, tenemos lo siguiente:

$$
\begin{aligned}
& \theta \in \mathcal{T}_{Env}[\![\Gamma]\!] \\
\Leftrightarrow\ & \forall x \in \mathbf{Var} : \theta(x) \in \mathcal{T}_{\theta}[\![\Gamma(x)]\!] \\
\Leftrightarrow\ & \qquad \{ \text{ por hipótesis de inducción } \} \\
& \forall x \in \mathbf{Var} : \theta(x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma(x)]\!] \\
\Leftrightarrow\ & \left(\forall x \in \mathbf{Var} \setminus \{\overline{x_i}\} : \theta(x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma(x)]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}\} : \theta(x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma(x)]\!]\right) \\
\Leftrightarrow\ & \qquad \{ \text{ porque } \overline{x_i} \text{ están sin restringir en } \Gamma \} \\
& \left(\forall x \in \mathbf{Var} \setminus \{\overline{x_i}\} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma(x)]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}\} : \theta(x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\mathsf{any}()]\!]\right) \\
\Leftrightarrow\ & \qquad \{ \text{ porque cada valor pertenece a la semántica de } \mathsf{any}() \} \\
& \left(\forall x \in \mathbf{Var} \setminus \{\overline{x_i}\} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma(x)]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}\} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\mathsf{any}()]\!]\right) \\
\Leftrightarrow\ & \forall x \in \mathbf{Var} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma(x)]\!] \\
\Leftrightarrow\ & \theta\left[\overline{x_i/v_i}\right] \in \mathcal{T}_{Env}[\![\Gamma]\!]
\end{aligned}
$$

□

El siguiente resultado muestra que, siempre que descartemos los tipos de algunas variables en un tipo o un entorno usando la notación $\tau \setminus \{x_1, \ldots, x_n\}$ o $\Gamma \setminus \{x_1, \ldots, x_n\}$, obtendremos tipos o entornos con semánticas iguales o mayores.

Proposición 7. *Dada una sustitución θ y un conjunto finito de pares $(x_1, v_1), \ldots, (x_n, v_n)$:*

1. $\mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\tau]\!] \subseteq \mathcal{T}_{\theta}[\![\tau \setminus \{x_1, \ldots, x_n\}]\!]$ *para todo tipo τ.*

2. $\theta\left[\overline{x_i/v_i}\right] \in \mathcal{T}_{Env}[\![\Gamma]\!] \Rightarrow \theta \in \mathcal{T}_{Env}[\![\Gamma \setminus \{x_1, \ldots, x_n\}]\!]$ *para todo entorno de tipos Γ.*

Demostración. Por inducción sobre la estructura de τ y Γ. De nuevo, todos los casos son sencillos de demostrar, a excepción de los casos del tipo funcional y del entorno de tipos.

Suponiendo que τ es un tipo funcional con la siguiente forma $\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma} \tau'$. Si $\theta\left[\overline{x_i/v_i}\right] \notin \mathcal{T}_{Env}[\![\Gamma]\!]$, entonces el resultado de $\mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}\left[\!\left[\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma} \tau'\right]\!\right]$ tendrá un solo elemento: la función vacía $\emptyset$, cuya pertenencia a $\mathcal{T}_{\theta}\left[\!\left[\left(\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma} \tau'\right) \setminus \{\overline{x_i}^n\}\right]\!\right]$ es trivial. Por otro lado, si $\theta\left[\overline{x_i/v_i}\right]$ pertenece a $\mathcal{T}_{Env}[\![\Gamma]\!]$, entonces

por la hipótesis de inducción obtenemos que $\theta \in \mathcal{T}_{Env}[\![\Gamma \setminus \{\overline{x_i}^n\}]\!]$ y por lo tanto:

$$
\begin{array}{ll}
& \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}\left[\!\left[\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma} \tau'\right]\!\right] \\
= & \quad \{\text{ porque } \theta\left[\overline{x_i/v_i}\right] \in \mathcal{T}_{Env}[\![\Gamma]\!] \} \\
& \left\{F \mid F \subseteq \left\{((v_1,\ldots,v_n),v) \mid \forall i.v_i \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\tau_i]\!], v \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\tau]\!]\right\}\right\} \\
\subseteq & \quad \{\text{ por hipótesis de inducción }\} \\
& \{F \mid F \subseteq \{((v_1,\ldots,v_n),v) \mid \forall i.v_i \in \mathcal{T}_\theta[\![\tau_i \setminus \{\overline{x_i}^n\}]\!], v \in \mathcal{T}_\theta[\![\tau \setminus \{\overline{x_i}^n\}]\!]\}\} \\
= & \quad \{\text{ porque } \theta \in \mathcal{T}_{Env}[\![\Gamma \setminus \{\overline{x_i}^n\}]\!] \} \\
& \mathcal{T}_\theta\left[\!\left[\left(\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma} \tau'\right) \setminus \{\overline{x_i}^n\}\right]\!\right]
\end{array}
$$

En el caso de un entorno de tipos Γ, obtenemos lo siguiente:

$$
\begin{array}{ll}
& \theta\left[\overline{x_i/v_i}\right] \in \mathcal{T}_{Env}[\![\Gamma]\!] \\
\Leftrightarrow & \quad \{\text{ por la definición de } \mathcal{T}_{Env}[\![\Gamma]\!] \} \\
& \forall x \in \mathbf{Var} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma(x)]\!] \\
\Rightarrow & \quad \{\text{ por hipótesis de inducción }\} \\
& \forall x \in \mathbf{Var} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_\theta[\![\Gamma(x) \setminus \{\overline{x_i}^n\}]\!] \\
\Leftrightarrow & \left(\forall x \in \mathbf{Var} \setminus \{\overline{x_i}^n\} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_\theta[\![\Gamma(x) \setminus \{\overline{x_i}^n\}]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}^n\} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_\theta[\![\Gamma(x) \setminus \{\overline{x_i}^n\}]\!]\right) \\
\Leftrightarrow & \quad \{\text{ porque } x_i \text{ está sin restringir en } \Gamma(x) \setminus \{\overline{x_i}^n\} \} \\
& \left(\forall x \in \mathbf{Var} \setminus \{\overline{x_i}^n\} : \theta(x) \in \mathcal{T}_\theta[\![\Gamma(x) \setminus \{\overline{x_i}^n\}]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}^n\} : \theta\left[\overline{x_i/v_i}\right](x) \in \mathcal{T}_\theta[\![\text{any}()]\!]\right) \\
\Rightarrow & \quad \{\text{ porque any}() \text{ contiene todos los valores } \} \\
& \left(\forall x \in \mathbf{Var} \setminus \{\overline{x_i}^n\} : \theta(x) \in \mathcal{T}_\theta[\![\Gamma(x) \setminus \{\overline{x_i}^n\}]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}^n\} : \theta(x) \in \mathcal{T}_\theta[\![\text{any}()]\!]\right) \\
\Rightarrow & \quad \{\text{ porque } x_i \text{ está sin restringir en } \Gamma(x) \setminus \{\overline{x_i}^n\} \} \\
& \forall x \in \mathbf{Var} : \theta(x) \in \mathcal{T}_\theta[\![\Gamma(x) \setminus \{\overline{x_i}^n\}]\!] \\
\Leftrightarrow & \quad \{\text{ por la definición de } \mathcal{T}_{Env}[\![\Gamma \setminus \{\overline{x_i}^n\}]\!] \} \\
& \theta \in \mathcal{T}_{Env}[\![\Gamma \setminus \{\overline{x_i}^n\}]\!]
\end{array}
$$

□

Una propiedad importante de las reglas de tipado es que, aparte de las restricciones ya definidas en el entorno de entrada, el entorno final no posee restricciones adicionales sobre aquellas variables que no estén libres en la expresión que estemos dando tipo, a no ser que el entorno final sea $\bot$.

Proposición 8. *Consideremos un juicio* $\Gamma \vdash e : \tau, \Gamma'$. *Para cada variable x, si x está sin restringir en* Γ *y*

no está libre en e, entonces Γ' *es* $\bot$ *o x está sin restringir en* τ *y* Γ'*. Lo mismo se cumple en los juicios de la forma* $\Gamma \Vdash_X cls : \tau, \Gamma'$ *para aquellas variables distintas a las contenidas por el patrón de cls, que además estén sin restringir en* Γ *y que no sean libres en el cuerpo de la cláusula.*

Demostración. Por inducción sobre la derivación de tipos. En el caso de una regla de subtipado, si x está sin restringir en Γ y en τ, también lo estará en cada entorno $\Gamma'' \supseteq \Gamma$ y cada tipo $\tau'' \supseteq \tau$. En el caso de las reglas [NONE], [APP-2] y [RECEIVE-3], tenemos que $\Gamma' = \bot$. En el resto de casos, el resultado se obtiene directamente de aplicar la hipótesis de inducción en las subderivaciones y del hecho de que al eliminar las variables ligadas por el **let**, el **letrec**, la λ-abstracción y las variables de un patrón, de los tipos y los entornos finales, hace que estas variables queden sin restringir.

Para entenderlo mejor, vamos a desarrollar el caso correspondiente a la regla [LET]. Sea e la expresión **let** $x = e_1$ **in** e_2 y sea z una variable sin restringir en Γ que no esté libre en e. Si $z \neq x$, entonces z no aparece libre ni en e_1 ni en e_2. Por hipótesis de inducción sobre el juicio $\Gamma \vdash e_1 : \tau_1 : \Gamma_1$ tenemos que, o bien $\Gamma_1 = \bot$, o bien z está sin restringir en Γ_1. En el primer caso, se tendría que Γ' sería también $\bot$. En el segundo caso, podemos aplicar hipótesis de inducción sobre el juicio $\Gamma_1 \sqcap [x : \tau_1] \vdash e_2 : \tau_2, \Gamma_2$ para concluir que $\Gamma_2 = \bot$, o bien z está sin restringir en Γ_2. Ambos casos conducen a la conclusión de la proposición para toda la expresión e. Por último, si $z = x$, la conclusión se sigue debido a que $\Gamma' = \Gamma_2 \setminus x$, por lo que x está sin restringir en Γ'. □

Antes de demostrar la corrección del sistema, necesitaremos definir la función $\mathscr{C}[\![\_]\!]$ como semántica para las cláusulas de la siguiente forma:

$$\mathscr{C}[\![p \textbf{ when } e \to e']\!]_V = \{(\theta, v) \mid (\forall v' \in V.(\theta, v') \in \mathscr{E}[\![p]\!]), (\theta, \texttt{'true'}) \in \mathscr{E}[\![e]\!], (\theta, v) \in \mathscr{E}[\![e']\!]\}$$

Con ella podremos conectar la semántica de las expresiones **case** y **receive** en la sección 2.3, con la corrección del sistema de tipos monomórfico presentado en este capítulo, usando la siguiente proposición:

Proposición 9. *Suponiendo una cláusula de la forma* p **when** $e \to e'$*, un conjunto de valores* V*, una sustitución* θ *y el par* $(\theta', v) \in \mathscr{E}[\![e']\!]$*, donde* $v' \in V$ *y* $\theta' \in matches(\theta, v', p \textbf{ when } e \to e')$*. Entonces se tiene que* $v_1, \ldots, v_n \in \textbf{DVal}$ *y* $\left(\theta\left[\overline{x_i / v_i}\right], v\right) \in \mathscr{C}[\![p \textbf{ when } e \to e']\!]_V$*, donde* $\{\overline{x_i}\} = vars(p)$*.*

Demostración. Por la definición de la función *matches* obtenemos:

$$\begin{aligned} & \theta' \in matches(\theta, v', p \textbf{ when } e \to e') \\ \Leftrightarrow\ & \theta' \in \left\{\theta\left[\overline{x_i / v_i}\right] \mid \overline{v_i} \in \textbf{DVal}, \left(\theta\left[\overline{x_i / v_i}\right], v'\right) \in \mathscr{E}[\![p]\!], \left(\theta\left[\overline{x_i / v_i}\right], \texttt{'true'}\right) \in \mathscr{E}[\![e]\!]\right\} \end{aligned}$$

donde $vars(p) = \{\overline{x_i}\}$. De aquí se cumple que $v_1, \ldots, v_n \in \textbf{DVal}$. Entonces tenemos que θ' es $\theta\left[\overline{x_i / v_i}\right]$, obteniendo $\left(\theta\left[\overline{x_i / v_i}\right], v\right) \in \mathscr{E}[\![e']\!]$, que junto a $\left(\theta\left[\overline{x_i / v_i}\right], \texttt{'true'}\right) \in \mathscr{E}[\![e]\!]$, $v' \in V$ y $\left(\theta\left[\overline{x_i / v_i}\right], v'\right) \in \mathscr{E}[\![p]\!]$, por definición, nos da que $\left(\theta\left[\overline{x_i / v_i}\right], v\right) \in \mathscr{C}[\![p \textbf{ when } e \to e']\!]_V$. □

Estamos ya en condiciones de obtener nuestro principal resultado de corrección acerca del sistema de tipos monomórficos de esta sección. Según él, dado un juicio $\Gamma \vdash e : \tau, \Gamma'$, los resultados de corrección nos muestran que, dada una sustitución $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$ tal que $e\theta$ se evalúe al valor v, dicho valor estará en la semántica de τ y θ estará en la semántica del entorno final Γ'. Un resultado similar se puede obtener para los juicios para las cláusulas. Ambos hechos han de ser probados simultáneamente, debido a la mutua dependencia entre los dos conjuntos de reglas.

Teorema 1. *Suponemos unos entornos de tipos Γ y Γ', un tipo τ, una expresión e, una cláusula cls y un conjunto X de variables. Entonces las siguientes dos afirmaciones son ciertas:*

1. *Si $\Gamma \vdash e : \tau, \Gamma'$, entonces $\mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$.*
2. *Si $\Gamma \Vdash_X cls : \tau, \Gamma'$, entonces*

$$\{(\theta, v) \in \mathcal{C}[\![cls]\!]_V \mid \forall x \in X, \theta(x) \in V\} \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$$

*para cualquier $V \subseteq$ **DVal**.*

Demostración. Dados unos entornos de tipos Γ y Γ', una expresión e y un tipo τ, suponemos que $\Gamma \vdash e : \tau, \Gamma'$. Probamos que $\mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$ por inducción sobre el tamaño de la derivación del tipo τ. Distinguimos los casos según la última regla aplicada.

- **Caso [SUB-1]**

  Existe un Γ'' tal que $\Gamma \subseteq \Gamma''$ y $\Gamma'' \vdash e : \tau, \Gamma'$. Por lo tanto:

$$\begin{array}{lll} & \mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma'']\!]} & \{\text{ porque } \Gamma \subseteq \Gamma'' \} \\ \subseteq & \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]} & \{\text{ por hipótesis de inducción } \} \end{array}$$

- **Caso [SUB-2]**

  Existe un Γ'' tal que $\Gamma'' \subseteq \Gamma'$ y $\Gamma \vdash e : \tau, \Gamma''$. Por lo tanto:

$$\begin{array}{lll} & \mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma'']\!]} & \{\text{ por hipótesis de inducción } \} \\ \subseteq & \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]} & \{\text{ porque } \Gamma'' \subseteq \Gamma' \} \end{array}$$

- **Caso [SUB-T]**

  Existe un τ' tal que $\tau' \subseteq \tau$ y $\Gamma \vdash e : \tau', \Gamma'$. Por lo tanto:

$$\begin{array}{lll} & \mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathcal{T}[\![\tau']\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]} & \{\text{ por hipótesis de inducción } \} \\ \subseteq & \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]} & \{\text{ porque } \tau' \subseteq \tau \} \end{array}$$

- **Caso [NONE]**

  Dado un entorno Γ tal que $\Gamma(x) = \texttt{none()}$ para una $x \in \textbf{Var}$, sabemos que $\mathcal{T}_{Env}[\![\Gamma]\!] = \emptyset$. En particular, esto sería lo que pasaría si aplicáramos la regla [NONE], ya que $\Gamma = \Gamma' \sqcap [x : \texttt{none()}]$ para una $x \in \textbf{Var}$ y por lo tanto $\Gamma(x) = \Gamma'(x) \sqcap \texttt{none()} = \texttt{none()}$, lo que hace que $\mathcal{T}_{Env}[\![\Gamma]\!]$ sea el conjunto vacío. Por ello:

  $$\mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} = \emptyset = \mathcal{T}[\![\texttt{none()}]\!] \restriction_{\mathcal{T}_{Env}[\![\bot]\!]}$$

- **Caso [TRANS]**

  Sabemos que existe un entorno intermedio Γ'' y su tipo intermedio τ' tal que $\Gamma \vdash e : \tau', \Gamma''$ y $\Gamma'' \vdash e : \tau, \Gamma'$. Ahora, suponemos una sustitución θ y un valor v tal que $(\theta, v) \in \mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. Esto es $(\theta, v) \in \mathcal{E}[\![e]\!]$ donde $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. Por hipótesis de inducción podemos asegurar que $\theta \in \mathcal{T}_{Env}[\![\Gamma'']\!]$. Por lo tanto, $(\theta, v) \in \mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma'']\!]}$ y podemos aplicar de nuevo la hipótesis de inducción para poder obtener $(\theta, v) \in \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$.

- **Caso [CONST]**

  Para este caso tenemos que:

  $$\begin{array}{ll} & \mathcal{E}[\![c]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \\ = & \{(\theta, c) \mid \theta \in \textbf{Subst}\} \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \\ = & \{(\theta, v) \mid \theta \in \textbf{Subst}, v \in \{c\}\} \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \\ = & \left\{(\theta, v) \mid \theta \in \textbf{Subst}, v \in \mathcal{T}_\theta[\![c]\!]\right\} \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \\ = & \mathcal{T}[\![c]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \end{array}$$

- **Caso [VAR]**

  Para este caso tenemos que:

  $$\begin{array}{ll} & \mathcal{E}[\![x]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \\ = & \{(\theta, v) \mid \theta \in \mathcal{T}_{Env}[\![\Gamma]\!], v = \theta(x)\} \\ \subseteq & \quad \left\{ \text{porque } \theta \in \mathcal{T}_{Env}[\![\Gamma]\!] \text{ implica que } \theta(x) \in \mathcal{T}_\theta[\![\Gamma(x)]\!] \right\} \\ & \left\{(\theta, v) \mid \theta \in \mathcal{T}_{Env}[\![\Gamma]\!], v \in \mathcal{T}_\theta[\![\Gamma(x)]\!]\right\} \\ = & \mathcal{T}[\![\Gamma(x)]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \end{array}$$

- **Caso [TUPLE]**

  Para cada $i \in \{1..n\}$, sabemos que existe una derivación de la forma $\Gamma \vdash e_i : \tau_i; \Gamma_i$ para unos τ_i y

Γ_i.

$$
\begin{array}{ll}
 & \mathscr{E}[\![\{e_1,\ldots,e_n\}]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \\
= & \left\{\left(\theta,\left(\{\overset{n}{\cdots}\},v_1,\ldots,v_n\right)\right) \mid \forall i\in\{1..n\}.\theta\in\mathscr{T}_{Env}[\![\Gamma]\!],(\theta,v_i)\in\mathscr{E}[\![e_i]\!]\right\} \\
\subseteq & \quad \{\text{ por h.i.: } \mathscr{E}[\![e_i]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}\subseteq\mathscr{T}[\![\tau_i]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}\} \\
 & \left\{\left(\theta,\left(\{\overset{n}{\cdots}\},v_1,\ldots,v_n\right)\right) \mid \forall i\in\{1..n\}.\theta\in\mathscr{T}_{Env}[\![\Gamma_i]\!],(\theta,v_i)\in\mathscr{T}[\![\tau_i]\!]\right\} \\
= & \quad \{\text{ porque } \textstyle\bigcap_{i=1}^n\mathscr{T}_{Env}[\![\Gamma_i]\!]=\mathscr{T}_{Env}[\![\bigsqcap_{i=1}^n\Gamma_i]\!] \text{ por la proposición 3}\} \\
 & \left\{\left(\theta,\left(\{\overset{n}{\cdots}\},v_1,\ldots,v_n\right)\right) \mid \theta\in\mathscr{T}_{Env}[\![\textstyle\bigsqcap_{i=1}^n\Gamma_i]\!],\forall i\in\{1..n\}.(\theta,v_i)\in\mathscr{T}[\![\tau_i]\!]\right\} \\
= & \{(\theta,v)\mid\theta\in\mathscr{T}_{Env}[\![\textstyle\bigsqcap_{i=1}^n\Gamma_i]\!],(\theta,v)\in\mathscr{T}[\![\{\tau_1,\ldots,\tau_n\}]\!]\} \\
= & \mathscr{T}[\![\{\tau_1,\ldots,\tau_n\}]\!]\restriction_{\mathscr{T}_{Env}[\![\bigsqcap_{i=1}^n\Gamma_i]\!]}
\end{array}
$$

- **Caso [LIST]**

De forma similar al caso de la regla [TUPLE], supondremos que $e=[e_1\,|\,e_2]$ para unas expresiones cualesquiera e_1 y e_2, y que existe una derivación de tipos para cada subexpresión: $\Gamma\vdash e_1:\tau_1,\Gamma_1$ y $\Gamma\vdash e_2:\tau_2,\Gamma_2$.

$$
\begin{array}{ll}
 & \mathscr{E}[\![[e_1\,|\,e_2]]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \\
= & \{(\theta,([\_\,|\,\_],v_1,v_2))\mid\theta\in\mathscr{T}_{Env}[\![\Gamma]\!],(\theta,v_1)\in\mathscr{E}[\![e_1]\!],(\theta,v_2)\in\mathscr{E}[\![e_2]\!]\} \\
\subseteq & \quad \{\text{ por h.i.: } \mathscr{E}[\![e_i]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}\subseteq\mathscr{T}[\![\tau_i]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}\} \\
 & \{(\theta,([\_\,|\,\_],v_1,v_2))\mid\theta\in\mathscr{T}_{Env}[\![\Gamma_1]\!],\theta\in\mathscr{T}_{Env}[\![\Gamma_2]\!],(\theta,v_1)\in\mathscr{T}[\![\tau_1]\!],(\theta,v_2)\in\mathscr{T}[\![\tau_2]\!]\} \\
= & \quad \{\text{ porque } \mathscr{T}_{Env}[\![\Gamma_1]\!]\cap\mathscr{T}_{Env}[\![\Gamma_2]\!]=\mathscr{T}[\![\Gamma_1\sqcap\Gamma_2]\!] \text{ por la proposición 3}\} \\
 & \{(\theta,([\_\,|\,\_],v_1,v_2))\mid\theta\in\mathscr{T}_{Env}[\![\Gamma_1\sqcap\Gamma_2]\!],(\theta,v_1)\in\mathscr{T}[\![\tau_1]\!],(\theta,v_2)\in\mathscr{T}[\![\tau_2]\!]\} \\
= & \{(\theta,v)\mid\theta\in\mathscr{T}_{Env}[\![\Gamma_1\sqcap\Gamma_2]\!],(\theta,v)\in\mathscr{T}[\![\texttt{nelist}(\tau_1,\tau_2)]\!]\} \\
= & \mathscr{T}[\![\texttt{nelist}(\tau_1,\tau_2)]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma_1\sqcap\Gamma_2]\!]}
\end{array}
$$

- **Caso [ABS]**

Suponemos que la expresión que está siendo tipada tiene la forma $\mathbf{fun}(x_1,\ldots,x_n)\to e_f$ y que tenemos una derivación $\Gamma\vdash e_f:\tau':\Gamma''$ para un tipo τ' y un entorno Γ''. Supondremos que $(\theta,F)\in\mathscr{E}[\![\mathbf{fun}(x_1,\ldots,x_n)\to e_f]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$. Esto implica que $\theta\in\mathscr{T}_{Env}[\![\Gamma]\!]$ y

$$F=\left\{((v_1,\ldots,v_n),v)\mid v_1,\ldots,v_n\in\mathbf{DVal},\left(\theta\left[\overline{x_i/v_i}\right],v\right)\in\mathscr{E}[\![e]\!]\right\} \tag{3.3}$$

Supondremos que $x_1,\ldots,x_n$ están sin restringir en Γ. De lo contrario, podríamos haber aplicado un renombramiento a los parámetros de la λ-abstracción, para obtener distintas variables cerradas a lo largo de la misma. Partiendo de esta presuposición, con la proposición 6 obtenemos que $\theta\left[\overline{x_i/v_i}\right]\in\mathscr{T}_{Env}[\![\Gamma]\!]$, lo cual nos lleva a que:

$$\left(\theta\left[\overline{x_i/v_i}\right],v\right)\in\mathscr{E}[\![e]\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}\subseteq\mathscr{T}[\![\tau']\!]\restriction_{\mathscr{T}_{Env}[\![\Gamma'']\!]}$$

donde la última inclusión se obtiene por la hipótesis de inducción. Entonces podremos reescri-

bir (3.3) como

$$F \subseteq \left\{((v_1,\ldots,v_n),v) \mid v_1,\ldots,v_n \in \mathbf{DVal}, \left(\theta\left[\overline{x_i/v_i}\right],v\right) \in \mathscr{T}[\![\tau']\!]\right\}$$

y es cierto que $\theta\left[\overline{x_i/v_i}\right] \in \mathscr{T}_{Env}[\![\Gamma'']\!]$, lo que implica que $v_j \in \mathscr{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma''(x_j)]\!]$ para cada $j \in \{1..n\}$. Por lo tanto:

$$F \subseteq \left\{((v_1,\ldots,v_n),v) \mid \forall j.v_j \in \mathscr{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\Gamma''(x_j)]\!], v \in \mathscr{T}_{\theta\left[\overline{x_i/v_i}\right]}[\![\tau']\!]\right\}$$

Por la definición de la semántica de un tipo funcional, es cierto que

$$F \in \mathscr{T}_{\theta\left[\overline{x_i/v_i}\right]}\left[\!\left[\left(\Gamma''(x_1),\ldots,\Gamma''(x_n)\right) \xrightarrow{\Gamma''} \tau'\right]\!\right]$$

o, dicho de otro modo,

$$\left(\theta\left[\overline{x_i/v_i}\right],F\right) \in \mathscr{T}\left[\!\left[\left(\Gamma''(x_1),\ldots,\Gamma''(x_n)\right) \xrightarrow{\Gamma''} \tau'\right]\!\right].$$

Entonces aplicando la proposición 7,

$$(\theta,F) \in \mathscr{T}\left[\!\left[\left(\left(\Gamma''(x_1),\ldots,\Gamma''(x_n)\right) \xrightarrow{\Gamma''} \tau'\right) \setminus \{x_1,\ldots,x_n\}\right]\!\right]$$

y, como sabemos que $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$,

$$(\theta,F) \in \mathscr{T}\left[\!\left[\left(\left(\Gamma''(x_1),\ldots,\Gamma''(x_n)\right) \xrightarrow{\Gamma''} \tau'\right) \setminus \{x_1,\ldots,x_n\}\right]\!\right] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$$

- **Caso [APP-1]**

  Suponemos que $e = f(x_1,\ldots,x_n)$, tal que $\Gamma(f) \sqcap \left(\left(\overline{\mathsf{any}()}^n\right) \xrightarrow{[]} \mathsf{any}()\right) = \left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma''} \tau'$ para unos $\tau_1,\ldots,\tau_n,\tau'$ y Γ''. Esta regla especifica que $\Gamma' = \Gamma \sqcap \left[f : \left(\overline{\mathsf{any}()}^n\right) \xrightarrow{[]} \mathsf{any}(), x_1 : \tau_1,\ldots,x_n : \tau_n\right] \sqcap \Gamma''$. Entonces tenemos:

$$\mathscr{E}[\![f(x_1,\ldots,x_n)]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} = \{(\theta,v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], ((\theta(x_1),\ldots,\theta(x_n)),v) \in \theta(f)\}$$

  Supondremos la sustitución θ y el valor v, tal que $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$ y $(\theta,v) \in \mathscr{E}[\![e]\!]$. Dado que $\theta(f)$ es una función de aridad n (de lo contrario no habría ningún v tal que (θ,v) perteneciera al conjunto mencionado arriba) sabemos que $\theta(f) \in \mathscr{T}_\theta\left[\!\left[\left(\overline{\mathsf{any}()}^n\right) \xrightarrow{[]} \mathsf{any}()\right]\!\right]$. Todo ello, junto al hecho cierto de que $\theta(f) \in \mathscr{T}_\theta[\![\Gamma(f)]\!]$, implica que $\theta(f) \in \mathscr{T}_\theta\left[\!\left[\left(\overline{\tau_i}^n\right) \xrightarrow{\Gamma''} \tau\right]\!\right]$. Además, como $((\theta(x_1),\ldots,\theta(x_n)),v) \in \theta(f)$, es cierto que:

  1. $\theta \in \mathscr{T}_{Env}[\![\Gamma'']\!]$. Si no $\theta(f)$ sería vacío.

2. Para cada $i \in \{1..n\}$, $\theta(x_i) \in \mathcal{T}_\theta[\![\tau_i]\!]$, así que $\theta \in \mathcal{T}_{Env}[\![[x_i : \tau_i]]\!]$.

3. $\theta \in \mathcal{T}_{Env}\left[\!\left[f : \left(\overline{\mathsf{any}()}^n\right) \xrightarrow{[]} \mathsf{any}() \right]\!\right]$.

4. $v \in \mathcal{T}_\theta[\![\tau]\!]$.

Por lo tanto, $\theta \in \mathcal{T}_{Env}\left[\!\left[\Gamma \sqcap [f : \left(\overline{\mathsf{any}()}^n\right) \xrightarrow{[]} \mathsf{any}(), x_1 : \tau_1, \ldots, x_n : \tau_n] \sqcap \Gamma''\right]\!\right] = \mathcal{T}_{Env}[\![\Gamma']\!]$ y $(\theta, v) \in \mathcal{T}[\![\tau]\!]$, obteniendo con ello que:

$$(\theta, v) \in \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$$

- **Caso [APP-2]**

De nuevo, tenemos que

$$\mathcal{E}[\![f(x_1, \ldots, x_n)]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} = \{(\theta, v) \mid \theta \in \mathcal{T}_{Env}[\![\Gamma]\!], ((\theta(x_1), \ldots, \theta(x_n)), v) \in \theta(f)\}$$

Probaremos por reducción al absurdo que este conjunto está vacío. Suponemos un par $(\theta, v) \in \mathcal{E}[\![f(x_1, \ldots, x_n)]\!]$ para un $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. La existencia de una tupla en este conjunto implica la existencia de una tupla $((\theta(x_1), \ldots, \theta(x_n)), v)$ en $\theta(f)$, siendo lo último una función de aridad n. Obtenemos entonces que $\theta(f) \in \mathcal{T}_\theta[\![\Gamma(f)]\!]$ y también sabemos que $\theta(f) \in \mathcal{T}_\theta\left[\!\left[\left(\overline{\mathsf{any}()}^n\right) \xrightarrow{[]} \mathsf{any}()\right]\!\right]$. Por lo tanto:

$$\begin{aligned} \theta(f) &\subseteq \Gamma(f) \\ \theta(f) &\subseteq \left(\overline{\mathsf{any}()}^n\right) \xrightarrow{[]} \mathsf{any}() \\ \mathsf{none}() &\subsetneq \theta(f) \end{aligned}$$

contradiciendo el hecho de que $\mathsf{none}()$ es el ínfimo de $\Gamma(f)$ y $\left(\overline{\mathsf{any}()}^n\right) \xrightarrow{[]} \mathsf{any}()$, suposición que exigimos en la regla [APP-2].

Después de probar que $\mathcal{E}[\![f(x_1, \ldots, x_n)]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$ está vacío, la demostración del teorema es trivial:

$$\mathcal{E}[\![f(x_1, \ldots, x_n)]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} = \emptyset \subseteq \mathcal{T}[\![\mathsf{none}()]\!] \restriction_{\mathcal{T}_{Env}[\![\bot]\!]}$$

- **Caso [LET]**

En este caso sabemos que $e =$ **let** $x = e_1$ **in** e_2 para algunos x, e_1 y e_2, y que $\Gamma \vdash e_1 : \tau_1, \Gamma_1$ y

$\Gamma_1 \sqcap [x:\tau_1] \vdash e_2 : \tau_2, \Gamma_2$ para algunas Γ_1, τ_1, Γ_2 y τ_2, así como $\tau = \tau_2 \backslash x$ y $\Gamma' = \Gamma_2 \backslash x$.

$$
\begin{array}{ll}
 & \mathscr{E}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \\
= & \{(\theta, v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], (\theta, v_1) \in \mathscr{E}[\![e_1]\!], (\theta[x/v_1], v) \in \mathscr{E}[\![e_2]\!]\} \\
\subseteq & \quad \{\text{por hipótesis de inducción}\} \\
 & \{(\theta, v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma_1]\!], (\theta, v_1) \in \mathscr{T}[\![\tau_1]\!], (\theta[x/v_1], v) \in \mathscr{E}[\![e_2]\!]\} \\
\subseteq & \quad \{\text{por las proposiciones 8 y 6}\} \\
 & \{(\theta, v) \mid \theta[x/v_1] \in \mathscr{T}_{Env}[\![\Gamma_1]\!], v_1 \in \mathscr{T}_{\theta[x/v_1]}[\![\tau_1]\!], (\theta[x/v_1], v) \in \mathscr{E}[\![e_2]\!]\} \\
= & \quad \{\text{porque } x \text{ tiene el tipo any() en } \Gamma_1\} \\
 & \{(\theta, v) \mid \theta[x/v_1] \in \mathscr{T}_{Env}[\![\Gamma_1 \sqcap [x:\tau_1]]\!], (\theta[x/v_1], v) \in \mathscr{E}[\![e_2]\!]\} \\
\subseteq & \quad \{\text{por hipótesis de inducción}\} \\
 & \{(\theta, v) \mid \theta[x/v_1] \in \mathscr{T}_{Env}[\![\Gamma_2]\!], (\theta[x/v_1], v) \in \mathscr{T}[\![\tau_2]\!]\} \\
\subseteq & \quad \{\text{por la proposición 7}\} \\
 & \{(\theta, v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma_2 \backslash x]\!], (\theta, v) \in \mathscr{T}[\![\tau_2 \backslash x]\!]\} \\
= & \mathscr{T}[\![\tau]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}
\end{array}
$$

- **Caso [CASE]**

Tenemos $e = \textbf{case } x \textbf{ of } cls_1 \ldots cls_n \textbf{ end}$ para alguna variable x y algunas cláusulas $cls_1, \ldots, cls_n$. Además, asumiremos que cada cláusula cls_i ($i = \{1..n\}$) tiene la forma $p_i \textbf{ when } e_i \rightarrow e_i'$, donde p_i es un patrón y e_i y e_i' son expresiones. Con esto tenemos que:

$$
\begin{array}{ll}
 & \mathscr{E}[\![\textbf{case } x \textbf{ of } cls_1 \ldots cls_n \textbf{ end}]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \\
= & \bigcup_{i=1}^{n} \{(\theta, v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], \theta' \in matches(\theta, \theta(x), cls_i), (\theta', v) \in \mathscr{E}[\![e_i']\!], \\
 & \qquad (\forall k < i : matches(\theta, \theta(x), cls_k) = \emptyset)\} \\
\subseteq & \bigcup_{i=1}^{n} \{(\theta, v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], \theta' \in matches(\theta, \theta(x), cls_i), (\theta', v) \in \mathscr{E}[\![e_i']\!]\} \\
\subseteq & \quad \{\text{por la proposición 9}\} \\
 & \bigcup_{i=1}^{n} \left\{(\theta, v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], \overline{v_j} \in \textbf{DVal}, \left(\theta\left[\overline{x_{ij}/v_j}\right], v\right) \in \mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}}\right\}
\end{array}
$$

Suponemos un par (θ, v) que pertenece a $\mathscr{E}[\![\textbf{case } x \textbf{ of } cls_1 \ldots cls_n \textbf{ end}]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$. Entonces $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$ y existe un $i \in \{1..n\}$ tal que $\left(\theta\left[\overline{x_{ij}/v_j}\right], v\right) \in \mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}}$ para unos valores $\overline{v_j}$, donde $\overline{x_{ij}}$ son las variables del patrón i-ésimo. Sabemos que existe una derivación de $\Gamma \Vdash_{\{x\}} cls_i : \tau_i, \Gamma_i$ para un τ_i y un Γ_i. La hipótesis de inducción especifica que:

$$\left\{(\theta, v) \in \mathscr{C}[\![cls_i]\!]_V \mid \forall y \in \{x\}, \theta(y) \in V\right\} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \subseteq \mathscr{T}[\![\tau_i]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}$$

para cualquier $V \subseteq \textbf{DVal}$. En este caso tenemos que $V = \{\theta(x)\}$, por lo que la condición $\forall y \in \{x\}, \theta(y) \in V$ se transforma en $\theta(x) = \theta(x)$, siendo cierta de forma trivial. Con esto último podemos reescribir la hipótesis de inducción como:

$$\mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \subseteq \mathscr{T}[\![\tau_i]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}$$

Sin perder precisión, podemos asumir que las variables del patrón $\{\overline{x_{ij}}\}$ tienen tipo `any()` en Γ, porque de lo contrario podríamos renombrarlas en su correspondiente cláusula. Por lo tanto $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$ implica $\theta\left[\overline{x_{ij}/v_j}\right] \in \mathcal{T}_{Env}[\![\Gamma]\!]$. Por consiguiente $\left(\theta\left[\overline{x_{ij}/v_j}\right], v\right) \in \mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}} \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$ y obtenemos que $\left(\theta\left[\overline{x_{ij}/v_j}\right], v\right) \in \mathcal{T}[\![\tau_i]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_i]\!]}$. Si denotamos como τ'_i el tipo $\tau_i \setminus \{\overline{x_{ij}}\}$, y respectivamente Γ'_i con el entorno $\Gamma' \setminus \{\overline{x_{ij}}\}$, obtenemos usando la proposición 7 que $(\theta, v) \in \mathcal{T}[\![\tau'_i]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma'_i]\!]}$. Finalmente, como es cierto que

$$\tau'_i \subseteq \bigsqcup_{i=1}^{n} \tau'_i = \tau \quad \text{y} \quad \Gamma'_i \subseteq \bigsqcup_{i=1}^{n} \Gamma'_i = \Gamma',$$

obtenemos $(\theta, v) \in \mathcal{T}[\![\tau']\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$, que demuestra el teorema.

- **Caso [RECEIVE-1]**

  Tenemos una expresión de la forma **receive** $cls_1 \ldots cls_n$ **after** $e_t \to e$ para unas expresiones e_t y e y unas cláusulas $cls_1, \ldots, cls_n$. Tenemos que $(\theta, v) \in \mathscr{E}[\![\textbf{receive } cls_1 \ldots cls_n \textbf{ after } e_t \to e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$, por lo que es cierto que $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. Entonces distinguiremos los casos:

  - **Caso 1**. Aplicando sobre la semántica de la expresión **receive** la proposición 9 existe algunas $v_1, \ldots, v_n, v_t \in \textbf{DVal}$ y un $i \in \{1..n\}$ tal que el par $\left(\theta\left[\overline{x_{ij}/v_i}\right], v\right) \in \mathscr{C}[\![cls_i]\!]_{\textbf{DVal}}$, $(\theta, v_t) \in \mathscr{E}[\![e_t]\!]$ y $v_t \in$ `integer()` $\cup$ {`'infinity'`}, donde $\overline{x_{ij}}$ son las variables en el patrón p_i de la cláusula cls_i. Como hemos aplicado la regla [RECEIVE-1], hemos derivado los juicios $\Gamma \vdash e_t : \tau_t, \Gamma_t$ y $\Gamma \Vdash_{\emptyset} cls_i : \tau_i, \Gamma_i$ para un τ_i y un Γ_i. Aplicando la hipótesis de inducción al primer juicio es cierto que $(\theta, v_t) \in \mathcal{T}[\![\tau_t]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_t]\!]}$, por lo que $\theta \in \mathcal{T}_{Env}[\![\Gamma_t]\!]$. La hipótesis de inducción para el segundo juicio corresponde a la siguiente forma:

    $$\left\{(\theta, v) \in \mathscr{C}[\![cls_i]\!]_{\textbf{DVal}} \mid \forall x \in \emptyset, \theta(x) \in \textbf{DVal}\right\} \restriction_{\mathcal{T}_{Env}[\![\Gamma_t]\!]} \subseteq \mathcal{T}[\![\tau_i]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_i]\!]}$$

    que equivale a la siguiente declaración:

    $$\mathscr{C}[\![cls_i]\!]_{\textbf{DVal}} \restriction_{\mathcal{T}_{Env}[\![\Gamma_t]\!]} \subseteq \mathcal{T}[\![\tau_i]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_i]\!]}$$

    Es más, podemos asumir que las variables del patrón $\overline{x_{ij}}$ tienen el tipo `any()` en Γ, porque de lo contrario podríamos renombrar la cláusula. Por lo tanto, $\theta\left[\overline{x_{ij}/v_i}\right] \in \mathcal{T}_{Env}[\![\Gamma_t]\!]$ y también $\left(\theta\left[\overline{x_{ij}/v_i}\right], v\right) \in \mathscr{C}[\![cls_i]\!]_{\textbf{DVal}} \restriction_{\mathcal{T}_{Env}[\![\Gamma_t]\!]}$, de lo que se sigue que $\left(\theta\left[\overline{x_{ij}/v_i}\right], v\right) \in \mathcal{T}[\![\tau_i]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_i]\!]}$ y, usando la proposición 7, (θ, v) pertenece a $\mathcal{T}[\![\tau_i \setminus \{\overline{x_{ij}}\}]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_i \setminus \{\overline{x_{ij}}\}]\!]}$.

  - **Caso 2**. En este caso $(\theta, v) \in \mathscr{E}[\![e]\!]$ y existe un $v_t \in$ `integer()` tal que $(\theta, v_t) \in \mathscr{E}[\![e_t]\!]$. Como en el caso previo, de aplicar la hipótesis de inducción al juicio $\Gamma \vdash e_t : \tau_t : \Gamma_t$ conseguimos que $(\theta, v_t) \in \mathcal{T}[\![\tau_t]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_t]\!]}$ y por lo tanto $\theta \in \mathcal{T}_{Env}[\![\Gamma_t]\!]$. Ahora podemos aplicar de nuevo la hipótesis de inducción sobre el juicio $\Gamma_t \vdash e : \tau, \Gamma'$ para obtener $(\theta, v) \in \mathscr{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_t]\!]} \subseteq \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$.

- **Caso [RECEIVE-2]**

  Como en el caso anterior, tenemos que distinguir casos sobre a qué conjunto pertenece (θ, v). Sin embargo, ahora el **Caso 2** de arriba no lo podemos satisfacer, porque asume la existencia de un par $(\theta, v_t) \in \mathcal{T}[\![\tau_t]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_t]\!]}$ donde $v_t \in$ `integer()`, lo cual contradice los requisitos de la regla, que nos exige [RECEIVE-2] que τ_t y `integer()` sean disjuntos. Como consecuencia, el único caso a considerar en esta regla sería el **Caso 1** de arriba. La demostración de este caso no depende de la existencia de un juicio $\Gamma \vdash e : \tau, \Gamma'$, que es el principal ausente en [RECEIVE-2], por lo tanto el teorema se cumple en este caso.

- **Caso [RECEIVE-3]**

  Una condición necesaria para que (θ, v) pertenezca a $\mathcal{E}[\![\textbf{receive } cls_1 \ldots cls_n \textbf{ after } e_t \to e]\!]$, es que el valor v_t resultante de la evaluación de e_t pertenezca a `integer()` $\cup$ {`'infinity'`}. Sin embargo, los requisitos en [RECEIVE-3] contradicen esta condición, por lo que obtenemos que $\mathcal{E}[\![\textbf{receive } cls_1 \ldots cls_n \textbf{ after } e_t \to e]\!] = \emptyset$ en este caso, con lo que el teorema se cumple de forma trivial.

- **Caso [CLS-TRUE]**

  En este caso conocemos que $\Gamma \Vdash_X cls : \tau, \Gamma'$ y tenemos que probar que:

  $$\{(\theta, v) \in \mathcal{C}[\![cls]\!]_V \mid \forall x \in X, \theta(x) \in V\} \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$$

  para cualquier $V \subseteq$ **Var**. Vamos a suponer un conjunto V de variables y un par (θ, v) tal que $(\theta, v) \in \mathcal{C}[\![cls]\!]_V$ y $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. Si cls_i tiene la forma p **when** $e_g \to e$, sabemos que $(\theta, v') \in \mathcal{E}[\![p]\!]$ para cada $v' \in V$, $(\theta,$ `'true'`$) \in \mathcal{E}[\![e_g]\!]$ y $(\theta, v) \in \mathcal{E}[\![e]\!]$. Aplicando la hipótesis de inducción al juicio $\Gamma \vdash p : \tau_p, \Gamma_p$, obtenemos que $v' \in \mathcal{T}[\![\tau_p]\!]$ para cada $v' \in V$ y $\theta \in \mathcal{T}_{Env}[\![\Gamma_p]\!]$. En otras palabras, $V \subseteq \mathcal{T}[\![\tau_p]\!]$. Pero, como cada variable $x \in X$ nos da que $\theta(x) \in V \subseteq \mathcal{T}[\![\tau_p]\!]$, entonces obtenemos que $\theta \in \mathcal{T}_{Env}[\![[X : \tau_p]]\!]$, por lo que $\theta \in \mathcal{T}_{Env}[\![\Gamma_p \sqcap [X : \tau_p]]\!]$. Ahora podemos aplicar la hipótesis de inducción con el juicio $\Gamma_p \sqcap [X : \tau_p] \vdash e_g : \tau_g, \Gamma_g$ para conseguir $(\theta,$ `'true'`$) \in \mathcal{T}[\![\tau_g]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_g]\!]}$ que implica $\theta \in \mathcal{T}_{Env}[\![\Gamma_g]\!]$. Por último, aplicamos la hipótesis de inducción en el juicio $\Gamma \vdash e : \tau, \Gamma'$, que nos permitirá probar que $(\theta, v) \in \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$.

- **Caso [CLS-FALSE]**

  De forma similar al caso anterior, asumiremos un conjunto $V \subseteq$ **Var** y un par $(\theta, v) \in \mathcal{C}[\![cls]\!]$. Lo último nos dirigirá a una contradicción, ya que implicaría la existencia de una tupla $(\theta,$ `'true'`$) \in \mathcal{E}[\![e_g]\!] \subseteq \mathcal{T}[\![\tau_g]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma_g]\!]}$, pero el requisito de la regla [CLS-FALSE] indica que `'true'` y τ_g han de ser disjuntos. Por lo tanto, el requisito $(\theta, v) \in \mathcal{C}[\![cls]\!]$ no es cierto y como consecuencia $\mathcal{C}[\![cls]\!]$ está vacío. Con ello se cumple el teorema de forma trivial.

- **Caso [CLS-TRANS]**

  Asumiendo un par $(\theta, v) \in \{(\theta, v) \in \mathcal{C}[\![cls]\!]_V \mid \forall x \in X, \theta(x) \in V\} \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. Existen un tipo τ' y un entorno Γ'' tales que $\Gamma \Vdash_X cls : \tau', \Gamma''$ y $\Gamma'' \Vdash_X cls : \tau, \Gamma'$. Después de aplicar la hipótesis de inducción

al primer juicio obtenemos que $(\theta, v) \in \mathcal{T}[\![\tau']\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma'']\!]}$ y por lo tanto $\theta \in \mathcal{T}_{Env}[\![\Gamma'']\!]$. Entonces aplicamos de nuevo la hipótesis de inducción para obtener:

$$(\theta, v) \in \{(\theta, v) \in \mathscr{C}[\![cls]\!]_V \mid \forall x \in X, \theta(x) \in V\} \restriction_{\mathcal{T}_{Env}[\![\Gamma'']\!]} \subseteq \mathcal{T}[\![\tau]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$$

- **Caso [LETREC]**

  En este caso asumimos que $e = \textbf{letrec } \overline{f_i = \textbf{fun}(\overline{x_{ij}}) \to e_i} \textbf{ in } e'$. Sin embargo, para no complicar la demostración asumiremos que tenemos una sola definición recursiva. Esta sería $e = \textbf{letrec } f = \textbf{fun}(\overline{y_i}) \to e_f \textbf{ in } e'$. La demostración se puede extender a varias definiciones mutuamente recursivas de forma sencilla. Dada una sustitución θ, definimos la función $F_\theta : \textbf{DVal} \to \textbf{DVal}$ como lo siguiente:

  $$F_\theta(v) = v'$$
  $$\text{donde } \{v'\} = \{v'' \mid (\theta[f/v], v'') \in \mathscr{E}[\![\textbf{fun}(\overline{y_i}) \to e_f]\!]\}$$

  Porque hemos aplicado la regla [LETREC], tendremos el siguiente juicio:

  $$\Gamma \sqcap [f : \tau_f] \vdash \textbf{fun}(\overline{y_i}) \to e_f : \tau'_f, \Gamma'_f$$

  para algún τ_f, τ'_f y Γ'_f tal que $\tau_f = \tau'_f \backslash f$, obteniendo con ello lo siguiente por hipótesis de inducción:

  $$\mathscr{E}[\![\textbf{fun}(\overline{y_i}) \to e_f]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma \sqcap [f:\tau_f]]\!]} \subseteq \mathcal{T}\left[\!\left[\tau'_f\right]\!\right] \restriction_{\mathcal{T}_{Env}\left[\!\left[\Gamma'_f\right]\!\right]} \tag{3.4}$$

  Además, sabemos que τ'_f debe ser un tipo funcional o un supertipo de un tipo funcional, por lo que asumiremos que existe un tipo funcional $(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r$ tal que $(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r = \tau'_f \sqcap (\overline{\text{any}()}) \to \text{any}()$. Si $\emptyset$ denota la función que no devuelve ningún valor (es decir, la función con un grafo vacío), demostraremos lo siguiente para cada número natural $m \geq 0$:

  $$F_\theta^m(\emptyset) \in \mathcal{T}_\theta\left[\!\left[(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r\right]\!\right] \tag{3.5}$$

  siempre que $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. En otras palabras, si iteramos sobre la aplicación de F_θ con la función vacía $\emptyset$ para un número dado m de veces, obtendremos un valor contenido en las semánticas de $(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r$. Probaremos (3.5) por inducción sobre m:

  - **Caso base ($m = 0$):** La función con el grafo vacío está contenida en las semánticas de cada tipo funcional, por lo que (3.5) se demuestra de forma trivial.
  - **Paso inductivo ($m \geq 1$):** Obtenemos $F_\theta^m(\emptyset) = F_\theta(F_\theta^{m-1}(\emptyset))$, donde $F_\theta^{m-1}(\emptyset)$ pertenece a la semántica de $(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r$ por hipótesis de inducción, así que también pertenece a $\mathcal{T}_\theta\left[\!\left[\tau'_f\right]\!\right] \subseteq \mathcal{T}_\theta\left[\!\left[\tau'_f \backslash f\right]\!\right] = \mathcal{T}_\theta[\![\tau_f]\!]$. Si denotamos con w el grafo representado por $F_\theta^{m-1}(\emptyset)$, es cierto que $\theta[f/w] \in \mathcal{T}_{Env}[\![\Gamma \sqcap [f : \tau_f]]\!]$ y con (3.4) obtenemos $(\theta[f/w], F_\theta(w)) \in \mathcal{T}\left[\!\left[\tau'_f\right]\!\right]$, o su equiva-

lente $F_\theta(w) \in \mathcal{T}_{\theta[f/w]} \left[\!\left[\tau'_f \right]\!\right] \subseteq \mathcal{T}_\theta \left[\!\left[\tau_f \right]\!\right]$, estando este último paso justificado por la proposición 7. Pero, como $F_\theta(w)$ es también un grafo, obtenemos que $F_\theta(w)$ pertenece también a $\mathcal{T}_\theta \left[\!\left[(\overline{\mathsf{any}\,()}) \to \mathsf{any}\,() \right]\!\right]$, por lo que conseguimos $F_\theta(w) \in \mathcal{T}_\theta \left[\!\left[(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r \right]\!\right]$.

Ahora demostraremos que $lfp\ F_\theta \in \mathcal{T}_\theta \left[\!\left[\tau_f \right]\!\right]$. Debido a que la función F_θ es monótona, tenemos la siguiente cadena ascendente de grafos:

$$\emptyset \subseteq F_\theta(\emptyset) \subseteq F_\theta^2(\emptyset) \subseteq F_\theta^3(\emptyset) \subseteq \ldots$$

Además, sabemos que $\bigcup_{i=1}^{\infty} F_\theta^i(\emptyset) = lfp\ F_\theta$ por el teorema del punto fijo de Kleene. Si cada función en la cadena ascendente mostrada arriba es la función con el grafo vacío $\emptyset$, entonces $lfp\ F_\theta = \emptyset$, que pertenece de forma trivial a la semántica de τ_f. Si no, debe haber un valor w en la cadena ascendente con un grafo no vacío tal que $w \in \mathcal{T}_\theta \left[\!\left[(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r \right]\!\right]$. Esto significa que $\theta \in \mathcal{T}_{Env} \left[\!\left[\Gamma_f \right]\!\right]$, ya que de lo contrario obtendríamos que $\mathcal{T}_\theta \left[\!\left[(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r \right]\!\right] = \{\emptyset\}$, contradiciendo el hecho de que el grafo de w no es vacío. Por lo tanto, por definición de la semántica de un tipo funcional, sabemos que:

$$\mathcal{T}_\theta \left[\!\left[(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r \right]\!\right] = \left\{ G \mid G \subseteq \left\{ ((v_1, \ldots, v_n), v) \mid \forall i. v_i \in \mathcal{T}_\theta \left[\!\left[\tau_i \right]\!\right], v \in \mathcal{T}_\theta \left[\!\left[\tau_r \right]\!\right] \right\} \right\}$$

Aplicando (3.5) obtenemos una cota superior a los elementos en la cadena ascendente:

$$\emptyset \subseteq F_\theta(\emptyset) \subseteq F_\theta^2(\emptyset) \subseteq F_\theta^3(\emptyset) \subseteq \ldots \subseteq \left\{ ((v_1, \ldots, v_n), v) \mid \forall i. v_i \in \mathcal{T}_\theta \left[\!\left[\tau_i \right]\!\right], v \in \mathcal{T}_\theta \left[\!\left[\tau_r \right]\!\right] \right\}$$

Por lo tanto:

$$lfp\ F_\theta = \bigcup_{i=1}^{\infty} F_\theta^i(\emptyset) \subseteq \left\{ ((v_1, \ldots, v_n), v) \mid \forall i. v_i \in \mathcal{T}_\theta \left[\!\left[\tau_i \right]\!\right], v \in \mathcal{T}_\theta \left[\!\left[\tau_r \right]\!\right] \right\}$$

y luego:

$$lfp\ F_\theta \in \mathcal{T}_\theta \left[\!\left[(\overline{\tau_i}) \xrightarrow{\Gamma_f} \tau_r \right]\!\right] \subseteq \mathcal{T}_\theta \left[\!\left[\tau'_f \right]\!\right] \subseteq \mathcal{T}_\theta \left[\!\left[\tau'_f \backslash f \right]\!\right] = \mathcal{T}_\theta \left[\!\left[\tau_f \right]\!\right] \tag{3.6}$$

Denotaremos el menor punto fijo de F_θ con w, y asumiremos que el par (θ, v) está contenido en el conjunto $\mathcal{E} \left[\!\left[\textbf{letrec } f = \textbf{fun}(\overline{y_i}) \to e_f \textbf{ in } e' \right]\!\right] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. Entonces es cierto que $(\theta[f/w], v) \in \mathcal{E} \left[\!\left[e' \right]\!\right]$ y $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. Junto al hecho de que $w \in \mathcal{T}_\theta \left[\!\left[\tau_f \right]\!\right]$ (con (3.6)), tenemos que $\theta[f/w] \in \mathcal{T}_{Env} \left[\!\left[\Gamma \sqcap [f : \tau_f] \right]\!\right]$. Además, como w es un punto fijo de F_θ, es cierto que el par $(\theta[f/w], w) \in \mathcal{E} \left[\!\left[\textbf{fun}(\overline{y_i}) \to e_f \right]\!\right]$, así que podemos aplicar la hipótesis de inducción mostrada en (3.4) para obtener que $\theta[f/w] \in \mathcal{T}_{Env} \left[\!\left[\Gamma'_f \right]\!\right]$, o su equivalente $\theta[f/w] \in \mathcal{T}_{Env} \left[\!\left[\left(\Gamma'_f \backslash f \right) \sqcap [f : \tau_f] \right]\!\right]$. Aplicando la hipótesis de indución sobre el juicio $\left(\Gamma'_f \backslash f \right) \sqcap [f : \tau_f] \vdash e' : \tau', \Gamma''$ obtenemos que:

$$(\theta[f/w], v) \in \mathcal{E} \left[\!\left[e' \right]\!\right] \restriction_{\mathcal{T}_{Env}\left[\!\left[\left(\Gamma'_f \backslash f \right) \sqcap [f : \tau_f] \right]\!\right]} \subseteq \mathcal{T} \left[\!\left[\tau' \right]\!\right] \restriction_{\mathcal{T}_{Env}[\![\Gamma'']\!]}$$

Con la proposición 7, obtenemos $(\theta, \nu) \in \mathscr{T}[\![\tau' \setminus f]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma'' \setminus f]\!]} = \mathscr{T}[\![\tau]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$, que demuestra el teorema.

□

Del teorema anterior es fácil deducir el siguiente corolario donde se muestra que, en particular en el caso de las expresiones cerradas, nuestras reglas derivan de hecho *success types* en el sentido de la definición 1.

Colorario 1. *Si e es una expresión cerrada y* $[] \vdash e : \tau, []$, *entonces* τ *es un* success type, *es decir* $\mathscr{E}[\![e]\!]_2 \subseteq \mathscr{T}[\![\tau]\!]_2$.

Capítulo 4

Sistema de success types polimórficos

En el capítulo 3 vimos el sistema para derivar tipos monomórficos, que desarrollamos como base inicial para nuestra investigación. Sobre ese trabajo vamos a realizar otra iteración para incorporar dos elementos: *success types* polimórficos y tipos sobrecargados. Lo primero nos permitirá superar los problemas que vimos a la hora de aplicar funciones polimórficas en la sección 3.5.3 y lo segundo nos permitirá añadir precisión para representar las diferentes ramas de ejecución de una λ-expresión. Por ello, a lo largo del siguiente capítulo vamos a introducir al lector en nuestro sistema para derivar *success types* polimórficos, explicando cómo se fue extendiendo el sistema monomórfico para incorporar tipos polimórficos [65, 67] y tipos sobrecargados [68].

En la sección 4.1 presentaremos la sintaxis de los tipos polimórficos que usa el sistema. En la sección 4.2 detallaremos un poco más sobre el uso de las instancias de variables de tipo, un concepto de especial relevancia en la semántica de los tipos. En la sección 4.3 presentaremos la semántica de los tipos polimórficos. En la sección 4.4 explicaremos cómo los tipos polimórficos se integran en los entornos de variables. En la sección 4.5 presentaremos las reglas de tipado que proponemos. En la sección 4.6 mostraremos ejemplos de derivación de tipos usando dichas reglas. Por último, en la sección 4.7 presentaremos los resultados de corrección del sistema de tipos.

4.1. Sintaxis de los tipos polimórficos

Mostramos la sintaxis de los tipos polimórficos en la figura 4.1. Como ya se explicó con anterioridad, en la sección 3.1, los tipos `none()` y `any()` representan la ausencia y la totalidad de valores, respectivamente. Para cada tipo básico B (como `integer()` o `atom()`), suponemos la definición semántica $\mathscr{B}[\![B]\!] \subseteq \mathbf{DVal}$ que contiene el conjunto de los valores denotados por este tipo. También seguimos teniendo literales, tuplas y listas no vacías, que quedaron explicados en la sección 3.1 del capítulo anterior. La primera novedad que tenemos, es que una variable de tipo α también puede ser un tipo.

$$
\begin{array}{rcl}
\textbf{Type} \ni \tau & ::= & \texttt{none()} \mid \texttt{any()} \mid B \mid c \mid \alpha \mid \{\overline{\tau_i}\} \mid \texttt{nelist}(\tau_1, \tau_2) \mid \tau_1 \cup \tau_2 \mid \bigsqcup_{i=1}^{m} \sigma_i \mid \tau \ \textbf{when} \ C \\
\sigma & ::= & \forall \overline{\alpha_j}.\left(\overline{\tau_i}\right) \to \tau \\
C & ::= & \left\{\overline{\varphi_i}\right\} \\
\varphi & ::= & \alpha \subseteq \tau \mid c \subseteq \tau \mid \tau \Leftarrow \alpha
\end{array}
$$

Figura 4.1: Sintaxis de los tipos, los esquemas de tipos y las restricciones

Suponemos que existe un conjunto **TypeVar** de variables de tipo y usaremos α, β, α_1, etcétera, para denotar los elementos de dicho conjunto.

El conjunto de valores a los que una variable de tipo puede ser instanciada se puede restringir mediante las *restricciones*. En particular, el tipo τ **when** C denota los valores del tipo τ, en los que aquellas variables de tipo existentes en τ serán instanciadas solamente a los conjuntos de valores que satisfagan las restricciones en C. La sintaxis de las restricciones está definida en la última línea de la figura 4.1. Una restricción de la forma $\alpha \subseteq \tau$ especifica que los valores a los que α es instanciada están contenidos en τ. La notación $\alpha_1 = \alpha_2$ la usaremos, a modo de simplificación, para denotar la conjunción de $\alpha_1 \subseteq \alpha_2$ y $\alpha_1 \supseteq \alpha_2$. También tenemos restricciones en las que el lado izquierdo de la relación $\subseteq$ es un literal c. Por último, tenemos restricciones de la forma $\tau \Leftarrow \alpha$, que es una variante de $\alpha \subseteq \tau$ más fuerte (es decir, más restrictiva). El significado preciso y la necesidad de este tipo de restricciones serán explicadas en más detalle en las secciones 4.3 y 4.4.

Un *esquema de tipo polimórfico* σ tiene la forma $\forall \overline{\alpha_j}.\left(\overline{\tau_i}^n\right) \to \tau$ y denota el conjunto de todas las funciones de aridad n que reciben valores de tipo $\overline{\tau_i}^n$ y devuelven valores de tipo τ. Decimos que todas las apariciones de las variables $\overline{\alpha_j}$ dentro de $\left(\overline{\tau_i}^n\right) \to \tau$ están *ligadas*.[1] El número de variables $\overline{\alpha_j}$ ligadas puede ser cero, en cuyo caso se omite el uso de símbolo $\forall$. Un tipo de la forma $\bigsqcup_{i=1}^{m} \sigma_i$ denota un esquema de tipo *sobrecargado*, que representa aquellas funciones que son resultado de unir los valores funcionales obtenidos de cada σ_i. Por ejemplo, el tipo $(0 \to 1) \sqcup (1 \to 0)$ contiene las funciones definidas por los siguientes grafos: $\emptyset$, $\{(0,1)\}$, $\{(1,0)\}$, $\{(0,1),(1,0)\}$. Nótese la diferencia con el tipo unión $(0 \to 1) \cup (1 \to 0)$, que denota los grafos $\emptyset$, $\{(0,1)\}$ y $\{(1,0)\}$, pero no el grafo $\{(0,1),(1,0)\}$.

Diremos que la aparición de una variable α dentro del tipo τ es *libre* si no está ligada. Denotamos con $ftv(\tau)$ el conjunto de variables de tipo que están libres en τ. También consideramos que **when** tiene mayor precedencia que $\{\to, \cup, \sqcup\}$, por lo que el tipo $(\alpha_1) \to \alpha_2$ **when** $\alpha_2 \subseteq \alpha_1$ es equivalente a $(\alpha_1) \to (\alpha_2$ **when** $\alpha_2 \subseteq \alpha_1)$. Hay que señalar que el operador $\sqcup$ tiene menos prioridad que $\to$, por lo que $0 \to 1 \sqcup 1 \to 0$ equivale a $(0 \to 1) \sqcup (1 \to 0)$.

Como se puede ver, la sintaxis de los tipos polimórficos contiene la mayoría de los elementos de los tipos monomórficos, vistos en la sección 3.1. Sin embargo el lector podrá notar que faltan los entornos Γ que se encontraban en los tipos funcionales monomórficos. En este sistema de tipos polimórfico los

[1] También diremos que son variables de tipo *cerradas* cuando se consideran desde fuera del esquema de tipo.

entornos siguen utilizándose, pero han sido movidos a otra estructura de la que hablaremos en la sección 4.4.

4.2. Instancias para variables de tipo

Habiendo introducido la sintaxis, tenemos que dar significado a cada tipo. Para ello necesitamos la función semántica $\mathcal{T}\,[\![\_]\!]$ que asigna a cada tipo el conjunto de valores que lo representan. Por ejemplo, esta función devolvería para el tipo `integer()` el conjunto de los números enteros y devolvería para el tipo `nelist(integer(),[])` el conjunto de todas las listas propias no vacías que se pueden construir con números enteros. Sin embargo, las cosas se vuelven más complicadas cuando tenemos que averiguar el conjunto de valores correspondientes a un tipo como `nelist(`α`,[])`. Al considerar este tipo de forma aislada, resulta que α puede representar cualquier valor y por lo tanto la lista `[3,'true']` pertenece a la semántica de `nelist(`α`,[])`. No obstante, α podría estar restringida en un contexto más amplio, tal como ocurre en el tipo `nelist(`α`,[])` **when** $\alpha \subseteq$ `integer()`, el cual debería excluir la lista `[3,'true']`. Por lo tanto, de cara a determinar cuándo un valor pertenece a la semántica de un tipo dado, tenemos que hacer seguimiento del conjunto de valores a los que α es instanciada para dicho valor. En este ejemplo, α es instanciada al conjunto {`3`, `'true'`} y podríamos determinar, en el contexto exterior, que esta instanciación no cumpliría la restricción $\alpha \subseteq$ `integer()`.

En un sistema de tipos estándar de *Hindley-Milner*, las instancias de variables de tipo son—a su vez—tipos. En este trabajo adoptamos un enfoque algo más genérico y consideramos en su lugar que una variable de tipo es instanciada como un conjunto de valores. Así definimos una *instanciación de tipo* como una función π : **TypeVar** $\to \mathscr{P}$ (**DVal**) y denotamos con **TypeInst** el conjunto de todas las instanciaciones de tipos. Decimos que una variable α es *instanciada* por π si $\pi(\alpha)$ no es vacío. En el ejemplo de arriba, decimos que `[3,'true']` pertenece a la semántica de `nelist(`α`,[])` bajo cualquier instanciación π tal que $\pi(\alpha)$ = {`3`, `'true'`}. Análogamente, dado un tipo unión τ = `{'ok',`α`}` $\cup$ `'error'`, decimos que `{'ok',5}` pertenece a la semántica de τ bajo cada π tal que $\pi(\alpha)$ = {`5`} y que `'error'` pertenece a la semántica de τ bajo cada π tal que $\pi(\alpha) = \emptyset$ (es decir, α no está instanciada en π).

Como consecuencia de lo anterior, nuestra función semántica $\mathcal{T}\,[\![\_]\!]$ devolverá, para cada tipo τ, un conjunto de pares (v,π) en lugar de un conjunto de valores. La instanciación π especifica cómo las variables de tipo en τ son instanciadas para dicho v. Por ejemplo, podemos decir que $(\texttt{[3,'true']},\pi) \in$ `nelist(`α`,[])` para cualquier π tal que $\pi(\alpha)$ = {`3`, `'true'`}. Las apariciones de α en este tipo determinan el valor de $\pi(\alpha)$ en cada par (v,π) que pertenezca a $\mathcal{T}\,[\![$`nelist(`α`,[])`$]\!]$. Sin embargo, puede haber algunas apariciones de una variable de tipo α que determinen α de una manera más débil, a pesar de restringir la $\pi(\alpha)$ correspondiente. Como ejemplo, consideremos las apariciones de α dentro de la restricción $\alpha \subseteq$ `integer()`. Existirán muchas opciones para $\pi(\alpha)$ tal que π satisfaga la restricción. Por lo tanto, será útil distinguir entre las apariciones de α que restrinjan fuertemente a la correspondiente $\pi(\alpha)$, de aquellas que solo estén para verificar los límites de α. Dada una variable $\alpha \in ftv(\tau)$, diremos que una variable de tipo está en una *posición instanciable* de un tipo, si esta aparece fuera

de una restricción o si está en una posición instanciable en el lado izquierdo de una restricción de la forma $\tau \Leftarrow \alpha$. Denotaremos con $itv(\tau)$ el conjunto de variables de tipo libres en τ que aparecen en posiciones instanciables de τ. Por ejemplo, consideremos el siguiente tipo τ:

$$\texttt{nelist(}\alpha\texttt{,[])}\ \textbf{when}\ \alpha \subseteq \beta, \alpha' \Leftarrow \beta$$

Las variables α y α' están en posiciones instanciables, mientras que β no está en una posición instanciable, por lo tanto $itv(\tau) = \{\alpha, \alpha'\}$.

Vamos a introducir ahora la notación para las instancias de tipos. Denotaremos con $[]$ la instanciación que asigna a cada variable de tipo el conjunto vacío. Usaremos $\left[\overline{\alpha_i \mapsto V_i}^n\right]$ para la instanciación que asigna a cada α_i su correspondiente conjunto V_i y el conjunto vacío para cualquier otra variable diferente de $\alpha_1, \ldots, \alpha_n$. Dadas dos instanciaciones π_1 y π_2, junto a un conjunto X de variables de tipo, diremos que $\pi_1 \equiv \pi_2$ (módulo X) si y solo si $\pi_1(\alpha) = \pi_2(\alpha)$ para cada $\alpha \in X$. Dada una instanciación π y un conjunto X de variables de tipo, denotaremos con $\pi \backslash X$ la instanciación π' tal que $\pi' \equiv []$ (módulo X) y $\pi' \equiv \pi$ (módulo $\textbf{TypeVar} \backslash X$). Es decir, π' es la sustitución que se comportará como π pero deja las variables en X sin instanciar. También elevamos el operador de conjuntos $\cup$ para instancias y usaremos la notación $\pi_1 \cup \pi_2$ para denotar aquella π tal que $\pi(\alpha) = \pi_1(\alpha) \cup \pi_2(\alpha)$ para cada $\alpha \in \textbf{TypeVar}$. Esto nos permitirá descomponer una instanciación π en un número de instanciaciones $\pi_1, \ldots, \pi_n$, cada una de ellas dando tipo a un componente de un valor dado (como es el caso de los elementos de una lista dada). Algunas veces la descomposición solo se deberá aplicar a un subconjunto de variables dado. Por lo tanto, vamos a definir una noción de descomposición que es más restrictiva que el operador $\cup$. Para cada conjunto Π de instanciaciones y cada tipo τ, definimos el conjunto $Dcp(\Pi, \tau)$ de la siguiente manera:

$$\pi \in Dcp(\Pi, \tau) \quad \overset{\text{def}}{\iff} \quad \pi = \bigcup \Pi \quad \wedge \quad \forall \pi' \in \Pi.\ \pi \equiv \pi' \text{ (módulo } \textbf{TypeVar} \backslash itv(\tau))$$

Por ejemplo, cuando Π es $\{[\alpha \mapsto \{1\}, \beta \mapsto \{4,5\}], [\alpha \mapsto \{2\}, \beta \mapsto \{4,5\}]\}$ y τ es $\texttt{nelist(}\alpha\texttt{,[])}$, tenemos que $[\alpha \mapsto \{1,2\}, \beta \mapsto \{4,5\}] \in Dcp(\Pi, \tau)$. Pero si $\Pi' = \{[\alpha \mapsto \{1\}, \beta \mapsto \{4\}], [\alpha \mapsto \{2\}, \beta \mapsto \{5\}]\}$ no hay una instanciación π tal que $\pi \in Dcp(\Pi', \tau)$, dado que β no está en una posición instanciable en τ, por ello no permitiremos su descomposición en los subconjuntos $\{4\}$ y $\{5\}$.

Merece la pena señalar que, acorde con la definición semántica dada en la siguiente sección, si hay múltiples apariciones de la misma variable de tipo en diferentes posiciones instanciables, entonces esta variable estará instanciada con el mismo conjunto en todas aquellas apariciones. Por ejemplo, el tipo $\{\alpha, \alpha\}$ denota el conjunto de tuplas cuyos componentes son iguales, y $(\texttt{[}\alpha\texttt{]}) \to \texttt{[}\alpha\texttt{]}$ denota aquellas funciones que, dada una lista, devuelven otra lista que contiene los mismos elementos de la lista de entrada, con una posible reorganización del orden de los elementos, haber añadido duplicados o haber eliminado duplicados de la lista de entrada. Esto supone una diferencia de lo que uno esperaría en un sistema de tipos al estilo de *Hindley-Milner*, en el cual la propiedad de la parametricidad [97, 120] permite que los elementos de la lista de salida sean un subconjunto de los de la lista

$$
\begin{aligned}
\mathscr{T}[\![\texttt{any()}]\!] &= \{(v,\pi) \mid v \in \mathbf{DVal}, \pi \in \mathbf{TypeInst}\} \\
\mathscr{T}[\![\texttt{none()}]\!] &= \emptyset \\
\mathscr{T}[\![B]\!] &= \{(v,\pi) \mid v \in \mathscr{B}[\![B]\!], \pi \in \mathbf{TypeInst}\} \\
\mathscr{T}[\![c]\!] &= \{(c,\pi) \mid \pi \in \mathbf{TypeInst}\} \\
\mathscr{T}[\![\alpha]\!] &= \{(v,\pi) \mid \pi \in \mathbf{TypeInst}, \pi(\alpha) = \{v\}\} \\
\mathscr{T}[\![\{\overline{\tau_i}^n\}]\!] &= \left\{\left(\left(\{\cdot^n\cdot\}, \overline{v_i}^n\right), \pi\right) \;\middle|\; \forall i \in \{1..n\}.\, (v_i,\pi) \in \mathscr{T}[\![\tau_i]\!]\right\} \\
\mathscr{T}[\![\texttt{nelist(}\tau,\tau'\texttt{)}]\!] &= \{(([\_\mid\_], \overline{v_i}^n, v'), \pi) \mid n \geq 1, \forall i \in \{1..n\}.\, (v_i,\pi_i) \in \mathscr{T}[\![\tau]\!], \\
&\qquad \pi \in Dcp(\{\overline{\pi_i}^n\}, \tau), (v',\pi) \in \mathscr{T}[\![\tau']\!]\} \\
\mathscr{T}[\![\tau_1 \cup \tau_2]\!] &= \mathscr{T}[\![\tau_1]\!] \setminus (itv(\tau_2) \setminus ftv(\tau_1)) \cup \mathscr{T}[\![\tau_2]\!] \setminus (itv(\tau_1) \setminus ftv(\tau_2)) \\
&\quad \text{donde } \mathscr{T}[\![\tau]\!] \setminus X = \{(v,\pi\setminus X) \mid (v,\pi) \in \mathscr{T}[\![\tau]\!]\} \\
\mathscr{T}\left[\!\!\left[\bigsqcup_{i=1}^n \sigma_i\right]\!\!\right] &= \left\{\left(\bigcup_{i=1}^n f_i, \pi\right) \;\middle|\; \forall i \in \{1..n\}.\, (f_i,\pi) \in \mathscr{S}[\![\sigma_i]\!]\right\} \\
\mathscr{T}[\![\tau \textbf{ when } C]\!] &= \{(v,\pi) \mid (v,\pi) \in \mathscr{T}[\![\tau]\!], \pi \vDash C\}
\end{aligned}
$$

$$
\mathscr{S}[\![\forall\overline{\alpha_j}^m.(\overline{\tau_i}^n) \to \tau]\!] = \{(f,\pi') \mid (f,\pi) \in \mathscr{F}[\![(\overline{\tau_i}^n) \to \tau]\!], \pi \equiv \pi' \text{ (módulo } \mathbf{TypeVar}\setminus\{\overline{\alpha_j}^m\})\}
$$

$$
\begin{aligned}
\mathscr{F}[\![(\overline{\tau_i}^n) \to \tau]\!] &= \{(\emptyset,\pi) \mid \pi \in \mathbf{TypeInst}, \forall \alpha \in itv((\overline{\tau_i}^n) \to \tau).\pi(\alpha) = \emptyset\} \cup \\
&\quad \{(f|_1,\pi') \mid \emptyset \subset f \subseteq \{(((\overline{v_i}^n), v), \pi) \mid \forall i \in \{1..n\}.\, (v_i,\pi) \in \mathscr{T}[\![\tau_i]\!], \\
&\qquad (v,\pi) \in \mathscr{T}[\![\tau]\!]\}, \pi' \in Dcp(f|_2, (\overline{\tau_i}^n) \to \tau)\} \\
&\quad \text{donde } f|_1 = \{w \mid (w,\pi) \in f\},\ f|_2 = \{\pi \mid (w,\pi) \in f\}
\end{aligned}
$$

Figura 4.2: Semántica de los tipos y los esquemas de tipos

de entrada. Si eso fuera lo que queremos expresar en nuestro sistema de tipos, tendríamos que usar el tipo $(\texttt{[}\alpha\texttt{]}) \to \texttt{[}\alpha'\texttt{]}$ **when** $\alpha' \subseteq \alpha$.

4.3. Semántica de los tipos polimórficos

En esta sección daremos un significado preciso a los tipos definiendo la función semántica $\mathscr{T}[\![\_]\!]$ mencionada antes. La definición de esta función se muestra en la figura 4.2. La semántica de `none()` es vacía, ya que denota la ausencia de valores. Con respecto a la semántica de `any()`, tipos unitarios (literales y la lista vacía) y los tipos básicos, cada valor está emparejado con cada instanciación posible, ya que no hay restricciones impuestas sobre variables de tipo. En el caso de una variable de tipo α, su semántica permite cualquier posible valor v, pero ahora α es instanciada con el conjunto unitario $\{v\}$.

En cuanto a los tipos tupla $\{\overline{\tau_i}^n\}$, estos representan las tuplas de n componentes $\left(\{\cdot^n\cdot\}, v_1, \ldots, v_n\right)$

$$
\begin{array}{rcl}
\pi \vDash \alpha \subseteq \tau & \iff & \pi(\alpha) \subseteq \left\{ v \mid (v, \pi') \in \mathscr{T}\,[\![\tau]\!]\,, \pi' \subseteq \pi \right\} \\
\pi \vDash c \subseteq \tau & \iff & c \in \left\{ v \mid (v, \pi') \in \mathscr{T}\,[\![\tau]\!]\,, \pi' \subseteq \pi \right\} \\
\pi \vDash \tau \Leftarrow \alpha & \iff & \exists (\pi_v)_{v \in \pi(\alpha)} : \pi = \bigcup_{v \in \pi(\alpha)} \pi_v \wedge (\forall v \in \pi(\alpha) : (v, \pi_v) \in \mathscr{T}\,[\![\tau]\!]) \\
 & & \\
\pi \vDash \{\varphi_1, \ldots, \varphi_n\} & \iff & \forall i \in \{1..n\} : \pi \vDash \varphi_i
\end{array}
$$

Figura 4.3: Satisfactibilidad de las restricciones

tales que cada v_i pertenece a la semántica de su correspondiente τ_i. Se requiere que las instanciaciones asociadas a cada componente sean todas iguales. Análogamente, un tipo lista `nelist(`τ`,`τ'`)` denota el conjunto de listas no vacías, cada una asociada con una instanciación π. A diferencia de las tuplas, la instancia π puede ser descompuesta en varias π_i, una para cada elemento de la lista. Para motivar el uso de $\pi \in Dcp\left(\{\overline{\pi_i}^n\}, \tau\right)$ en lugar de $\pi = \bigcup_{i=1}^n \pi_i$, consideremos el tipo `nelist((0` **when** $\alpha \subseteq$ `integer())` $\cup$ `(1` **when** `'true'` $\subseteq \alpha$`))`. Claramente las dos restricciones son incompatibles entre sí, por lo que tiene sentido excluir la lista `[0,1]` de este tipo. Sin embargo, tenemos que $(\texttt{0}, [\alpha \mapsto \{\texttt{3}\}]) \in \mathscr{T}\,[\![\texttt{0}$ **when** $\alpha \subseteq$ `integer()`$]\!]$ y $(\texttt{1}, [\alpha \mapsto \{\texttt{'true'}\}]) \in \mathscr{T}\,[\![\texttt{1}$ **when** `'true'` $\subseteq \alpha]\!]$. Si usáramos la unión conjuntista para unir ambas instancias, podríamos obtener que $(\texttt{[0,1]}, [\alpha \mapsto \{\texttt{3}, \texttt{'true'}\}])$ pertenece a la semántica del tipo lista, algo que no se desea, en concreto debido a que $\{\texttt{3}, \texttt{'true'}\}$ no satisface ninguna de las restricciones en dicho tipo.

Los valores denotados por un tipo $\tau_1 \cup \tau_2$ son la unión de los valores denotados por cada constituyente. Sin embargo, si tomamos el par (v, π) de $\mathscr{T}\,[\![\tau_1]\!]$, tenemos que asegurarnos de que todas las variables en $itv(\tau_2)$—pero no en $ftv(\tau_1)$—están sin instanciar y viceversa. Por ejemplo, el tipo `nelist(`α`,[])` $\cup$ `[]`—que también podemos expresarlo con la notación `[`α`]`—contiene la lista vacía, pero este valor tiene que ser acompañado con una instancia π tal que $\pi(\alpha) = \emptyset$. Para poder comprender los motivos que hay detrás de esta restricción, supongamos la función f de tipo $\forall \alpha_1, \alpha_2.$ `([`α_1`])` $\rightarrow$ `[`α_2`]` **when** $\alpha_2 \subseteq \alpha_1$. Lo que esperamos de f, cuando recibe una lista, es que devuelva otra lista cuyos elementos son un subconjunto de aquellos recibidos en la lista de entrada. Como consecuencia, esperamos de f que si recibe una lista vacía, solamente pueda devolver otra lista vacía. Si no forzáramos a α_1 estar sin instanciar, en el caso de recibir f una lista vacía, entonces α_2 podría llegar a contener cualquier elemento al que α_1 se instancia, permitiendo así que f`([])` se pueda evaluar a cualquier otra lista en lugar de `[]`.

La semántica de un tipo sobrecargado de la forma $\bigsqcup_{i=1}^n \sigma_i$ es más compleja. Primero, tenemos que definir la semántica de un tipo funcional $\left(\overline{\tau_i}^n\right) \rightarrow \tau$ usando la función $\mathscr{F}\,[\![\_]\!]$. Esta semántica denotará grafos de funciones de aridad n. La función que siempre falla (es decir, la función con el grafo vacío) pertenece a todo tipo funcional. En este caso, ninguna de las variables en posiciones instanciables de $\left(\overline{\tau_i}^n\right) \rightarrow \tau$ estarían instanciadas. Si suponemos que existe un par $(f, \pi) \in \mathscr{F}\left[\!\left[\left(\overline{\tau_i}^n\right) \rightarrow \tau\right]\!\right]$ tal que el grafo de f no es vacío, descompondremos π en tantas instanciaciones como tuplas contenga el grafo

de f. Por cada tupla de entrada-salida $w = ((\overline{v_i}^n), v)$ en el grafo acompañada con su correspondiente instanciación π_w, cada argumento v_i tiene que pertenecer al tipo τ_i del correspondiente parámetro y el resultado v tiene que estar contenido por el tipo del resultado τ, todos ellos con la misma π_w. Por ejemplo, supongamos el tipo $([\alpha]) \to$ `integer()`. Tenemos, por un lado, que $([1,5], [\alpha \mapsto \{1,5\}]) \in \mathscr{T}[\![[\alpha]]\!]$ y que $(0, [\alpha \mapsto \{1,5\}]) \in \mathscr{T}[\![\texttt{integer()}]\!]$. Por otro lado, tenemos que $([7,\texttt{'b'},\texttt{'false'}], [\alpha \mapsto \{\texttt{'b'},7,\texttt{'false'}\}]) \in \mathscr{T}[\![[\alpha]]\!]$ y que $(1, [\alpha \mapsto \{\texttt{'b'},7,\texttt{'false'}\}]) \in \mathscr{T}[\![\texttt{integer()}]\!]$. Por lo tanto, tenemos que la función con el grafo $\{(([1,5]),0),(([7,\texttt{'b'},\texttt{'false'}]),1)\}$ pertenece a la semántica de $([\alpha]) \to$ `integer()` con la instancia $[\alpha \mapsto \{1,5,7,\texttt{'b'},\texttt{'false'}\}]$.

Ahora consideremos la semántica para un esquema de tipo funcional $\forall\overline{\alpha_i}.(\overline{\tau_i}) \to \tau$, usando la función $\mathscr{S}[\![\_]\!]$. En cada par $(f,\pi) \in \mathscr{F}[\![(\overline{\tau_i}) \to \tau]\!]$ tenemos que "eliminar" de π la información relativa a las variables de tipo ligadas $\overline{\alpha_i}$, ya que es irrelevante fuera del contexto del esquema de tipo cuantificado. Es por ello que podremos reemplazar la instancia π por cualquier otra π' siempre que se mantengan las variables que no están cerradas en el esquema. Finalmente, la semántica de un esquema de tipo sobrecargado $\bigsqcup_{i=1}^n \sigma_i$ es el conjunto de todas las funciones cuyos grafos pueden ser descompuestos en n subconjuntos, cada uno de ellos contenido por su correspondiente $\mathscr{S}[\![\sigma_i]\!]$.

La semántica de un tipo de la forma τ **when** C contiene aquellos pares (v,τ) que pertenecen a la semántica de τ pero siempre que π satisfaga todas las restricciones en C. Denotamos esta última condición como $\pi \vDash C$. La relación de satisfactibilidad de restricciones está definida en la figura 4.3. Una restricción $\alpha \subseteq \tau$ se satisface con π si los valores de $\pi(\alpha)$ están contenidos en la semántica de τ, cada uno de ellos con una instanciación consistente con los valores de π. En esta definición, suponemos que la relación $\subseteq$ sobre conjuntos es elevada a las instanciaciones de tipos de la manera habitual. Por ejemplo, diremos que $\pi = [\alpha \mapsto \{[1,2]\}, \beta \mapsto \{1,2,3\}]$ satisface $\alpha \subseteq [\beta]$, pero $\pi' = [\alpha \mapsto \{[1,2,5]\}, \beta \mapsto \{1,2,3\}]$ no lo hace, ya que no hay ninguna π'' tal que $([1,2,5], \pi'') \in \mathscr{T}[\![[\beta]]\!]$ y $\pi''(\beta) \subseteq \{1,2,3\}$. La definición para satisfacer una restricción de la forma $c \subseteq \tau$ es similar a la anterior. Las restricciones de encaje de la forma $\tau \Leftarrow \alpha$ son una versión más restrictiva de las restricciones de subconjunto y son útiles para la normalización de los entornos de tipos, operación que será explicada en la siguiente sección 4.4. Como ejemplo, consideremos la diferencia entre $\pi \vDash \alpha \subseteq [\beta]$ y $\pi \vDash [\beta] \Leftarrow \alpha$. Suponemos que $\pi(\alpha)$ contiene un único valor, por ejemplo la lista `[0,4]`. La restricción $\alpha \subseteq \beta$ permite una instancia $\pi(\beta)$ que sea cualquier superconjunto de $\{0,4\}$, mientras que $[\beta] \Leftarrow \alpha$ fuerza a $\pi(\beta)$ a ser *exactamente igual* a $\{0,4\}$. En general, si tenemos $\pi \vDash \tau \Leftarrow \alpha$ y $\pi(\alpha) = \{v\}$, entonces debe cumplirse que $(v,\pi) \in \mathscr{T}[\![\tau]\!]$. En el caso en el que $\pi(\alpha) = \{v_1,\ldots,v_n\}$, entonces las v_i tendrán que estar contenidas por $\mathscr{T}[\![\tau]\!]$, cada una con una posible instanciación diferente, pero la unión de todas estas instanciaciones han de ser exactamente π. Volviendo al ejemplo anterior, si $\pi(\alpha) = \{[0,4], [1,\texttt{'true'}]\}$ y $\pi \vDash [\beta] \Leftarrow \alpha$, entonces ha de cumplirse $\pi(\beta) = \{0,4,1,\texttt{'true'}\}$.

Frente a la semántica de los tipos monomórficos, vemos que la función $\mathscr{T}[\![\_]\!]$ ya no es paramétrica con respecto a ninguna sustitución θ, ya que los entornos de tipos ya no pertenecen a la sintaxis de los tipos. Por ello se pudo prescindir de dichas sustituciones. Para entenderlo mejor, volvamos al ejemplo

de la expresión **fun**(X) → X + Y, donde el resultado del sistema monomórfico era:

$$(\texttt{number()}) \xrightarrow{[\texttt{Y:number()}]} \texttt{number()}$$

Supongamos que la variable Y tiene externamente como tipo a α_Y. Entonces el tipo que usaríamos—para expresar la misma condición necesaria sobre esta variable libre—sería:

$$(\texttt{number()}) \to \texttt{number()} \textbf{ when } \alpha_Y \subseteq \texttt{number()}$$

De esta manera, cuando la función es aplicada, la restricción $\alpha_Y \subseteq$ number() es añadida al contexto donde la función ha sido aplicada, afectando a la información que conocemos de α_Y en dicho contexto. Gracias a este mecanismo, podemos prescindir de los entornos de tipos en los tipos funcionales. Sin embargo, veremos en la siguiente sección que los entornos de tipos siguen existiendo en nuestro sistema de tipos polimórficos.

Hay una propiedad que se cumple entre instanciaciones y tipos que necesitamos. Dado un par (v,π) perteneciente a la semántica de un tipo τ, solo las variables de tipo que aparecen libres en τ son relevantes en la instanciación π. El resto de variables pueden ser reasignadas a otro conjunto de valores cualquiera en π. Esto es lo que formaliza el siguiente lema:

Lema 1. *Sean π y π' dos instanciaciones, τ un tipo y C un conjunto de restricciones. Se tiene:*

1. *Para cualquier valor v, si $(v,\pi) \in \mathcal{T}[\![\tau]\!]$ y $\pi \equiv \pi'$* (módulo $ftv(\tau)$)*, entonces $(v,\pi') \in \mathcal{T}[\![\tau]\!]$.*
2. *Si $\pi \vDash C$ y $\pi \equiv \pi'$* (módulo $ftv(C)$) *entonces $\pi' \vDash C$.*

Demostración. Por inducción sobre la estructura de τ y C. Los casos en los que τ son tipos base simples (unitarios, none(), any(), integer(), etcétera) son triviales. Ahora consideremos el resto de casos.

- **Caso** $\tau = \alpha$.

  Si $(v,\pi) \in \mathcal{T}[\![\alpha]\!]$ entonces $\pi(\alpha) = \{v\}$. Ya que $\pi \equiv \pi'$ (módulo $ftv(\tau)$), sabemos que $\pi(\alpha) = \pi'(\alpha)$ para cada $\alpha \in ftv(\tau)$, por lo tanto $\pi'(\alpha) = \{v\}$ y $(v,\pi') \in \mathcal{T}[\![\alpha]\!]$.

- **Caso** $\tau = \{\overline{\tau_i}^n\}$.

  Suponemos que $(v,\pi) \in \mathcal{T}[\![\{\overline{\tau_i}^n\}]\!]$ y que v tiene la forma $\left(\{\overset{n}{\cdots}\}, \overline{v_i}^n\right)$, entonces tenemos que $(v_i,\pi) \in \mathcal{T}[\![\tau_i]\!]$, para cada $i \in \{1..n\}$. Si suponemos una instanciación $\pi \equiv \pi'$ (módulo $ftv\left(\{\overline{\tau_i}^n\}\right)$), se cumple que $\pi \equiv \pi'$ (módulo $ftv(\tau_i)$), para cada $i \in \{1..n\}$, y por hipótesis de inducción sabemos que $(v_i,\pi') \in \mathcal{T}[\![\tau_i]\!]$, y por lo tanto $(v,\pi') \in \mathcal{T}[\![\{\overline{\tau_i}^n\}]\!]$.

- **Caso** $\tau =$ nelist(τ_1,τ_2).

  Suponemos que $(v,\pi) \in \mathcal{T}[\![\texttt{nelist}(\tau,\tau')]\!]$ y que v tiene la forma $\left([\_|\_], \overline{v_i}^n, v'\right)$. Entonces tenemos que $\pi \in Dcp\left(\{\overline{\pi_i}^n\}, \tau\right)$, de modo que $(v_i,\pi_i) \in \mathcal{T}[\![\tau]\!]$, para cada $i \in \{1..n\}$, y que $\left(v',\pi\right) \in$

$\mathscr{T}[\![\tau']\!]$. Si suponemos $\pi \equiv \pi'$ (módulo $ftv(\texttt{nelist}(\tau,\tau'))$), se cumple también que $\pi \equiv \pi'$ (módulo $ftv(\tau)$) y que $\pi \equiv \pi'$ (módulo $ftv(\tau')$). Por un lado, tenemos por hipótesis de inducción que $(v',\pi') \in \mathscr{T}[\![\tau']\!]$. Por el otro, si definimos π'_i del siguiente modo

$$\forall \alpha \in \mathbf{TypeVar}.\ \pi'_i(\alpha) = \begin{cases} \pi_i(\alpha) & \text{si } \alpha \in itv(\tau) \\ \pi'(\alpha) & \text{e.o.c.} \end{cases}$$

se tiene, para toda $\alpha \in \mathbf{TypeVar} \setminus itv(\tau)$, que se cumple $\pi'(\alpha) = \pi'_i(\alpha)$ y, para toda $\alpha \in \mathbf{TypeVar}$, tenemos además que:

- Si $\alpha \in itv(\tau)$, entonces $\alpha \in ftv(\tau)$ y por ello $\pi'(\alpha) = \pi(\alpha) = \bigcup_{i=1}^{n} \pi_i(\alpha) = \bigcup_{i=1}^{n} \pi'_i(\alpha)$.
- Si $\alpha \notin itv(\tau)$, entonces $\pi'(\alpha) = \bigcup_{i=1}^{n} \pi'(\alpha) = \bigcup_{i=1}^{n} \pi'_i(\alpha)$.

Por lo tanto, se cumple que $\pi' = \bigcup_{i=1}^{n} \pi'_i$ y $\pi' \in Dcp\left(\left\{\overline{\pi'_i}^{\,n}\right\}, \tau\right)$. Entonces, sea $\alpha \in ftv(\tau)$:

- Si $\alpha \in itv(\tau)$, entonces $\pi'_i(\alpha) = \pi_i(\alpha)$, para cada $i \in \{1..n\}$.
- Si $\alpha \notin itv(\tau)$, entonces $\pi'_i(\alpha) = \pi'(\alpha) = \pi(\alpha)$, porque $\pi \equiv \pi'$ (módulo $ftv(\tau)$), y $\pi(\alpha) = \pi_i(\alpha)$, porque $\pi \equiv \pi_i$ (módulo $itv(\tau)$).

Por lo que $\pi_i \equiv \pi'_i$ (módulo $ftv(\tau)$) y con la hipótesis de inducción obtendríamos que $(v_i,\pi'_i) \in \mathscr{T}[\![\tau]\!]$. Por lo tanto, se cumple que $(v,\pi') \in \mathscr{T}[\![\texttt{nelist}(\tau,\tau')]\!]$.

- **Caso** $\tau = \tau_1 \cup \tau_2$.

  Suponemos que $(v,\pi) \in \mathscr{T}[\![\tau_1 \cup \tau_2]\!]$ y $\pi \equiv \pi'$ (módulo $ftv(\tau_1 \cup \tau_2)$). Tenemos dos casos que contemplar:

  - $(v,\pi) \in \mathscr{T}[\![\tau_1]\!] \setminus \left(itv(\tau_2) \setminus ftv(\tau_1)\right)$
    En este caso, para toda $\alpha \in \left(itv(\tau_2) \setminus ftv(\tau_1)\right)$, sabemos que $\pi(\alpha) = \emptyset$ y por lo tanto $\pi'(\alpha)$ también lo será. Entonces, aplicando la hipótesis de inducción, obtendremos que $(v,\pi') \in \mathscr{T}[\![\tau_1]\!] \setminus \left(itv(\tau_2) \setminus ftv(\tau_1)\right)$.
  - $(v,\pi) \in \mathscr{T}[\![\tau_2]\!] \setminus \left(itv(\tau_1) \setminus ftv(\tau_2)\right)$
    Es análogo al caso anterior.

  Juntando los dos resultados tendremos que $(v,\pi') \in \mathscr{T}[\![\tau_1 \cup \tau_2]\!]$.

- **Caso** $\tau = \tau_1$ **when** C.

  Suponemos que $(v,\pi) \in \mathscr{T}[\![\tau \textbf{ when } C]\!]$, teniendo entonces que $(v,\pi) \in \mathscr{T}[\![\tau]\!]$ y $\pi \vDash C$. Si suponemos una instanciación $\pi \equiv \pi'$ (módulo $ftv(\tau \textbf{ when } C)$), por hipótesis de inducción obtendremos que $(v,\pi') \in \mathscr{T}[\![\tau]\!]$ y $\pi' \vDash C$, por lo tanto se cumple que $(v,\pi') \in \mathscr{T}[\![\tau \textbf{ when } C]\!]$.

- **Caso** $\tau = \bigsqcup_{i=1}^{n} \sigma_i$.

  Para simplificar esta demostración, vamos primero a desarrollar dos resultados intermedios que necesitamos para demostrar el caso actual del lema:

1. **Tipos funcionales:**

   Supongamos que $(v,\pi) \in \mathscr{F}\,[\![(\overline{\tau_j}^m) \to \tau]\!]$ y que $\pi \equiv \pi^\circ$ (módulo $ftv((\overline{\tau_j}^m) \to \tau)$). Tendremos que distinguir dos casos para v:

   - **Caso** $v = \emptyset$**:**

     En este caso sabemos que, para toda $\alpha \in itv((\overline{\tau_j}^m) \to \tau)$, se tiene que cumplir que $\pi(\alpha) = \emptyset$, es decir, tenemos que $\pi^\circ(\alpha) = \emptyset$. Por lo que $(\emptyset, \pi^\circ) \in \mathscr{F}\,[\![(\overline{\tau_j}^m) \to \tau]\!]$.

   - **Caso** $v \neq \emptyset$**:**

     En este caso tenemos por la definición que:

     $$\emptyset \subset f \subseteq \{(((\overline{v_j}^m), v'), \pi_w) \mid \forall\, j \in \{1..m\}.\,(v_j, \pi_w) \in \mathscr{T}\,[\![\tau_j]\!], (v', \pi_w) \in \mathscr{T}\,[\![\tau]\!]\}$$

     Sabemos que, para cada $w \in v$, tenemos que $w \in f|_1$ y por lo tanto $w = ((\overline{v_j}^m), v')$, así como que $\pi_w \in f|_2$ tal que $\pi \in Dcp(f|_2, (\overline{\tau_j}^m) \to \tau)$. Entonces, de forma similar a lo expuesto en el caso de las listas no vacías, podemos descomponer π° tal que $\pi \in Dcp(\{\overline{\pi^\circ_w}\}, (\overline{\tau_j}^m) \to \tau)$, donde $\pi_w \equiv \pi^\circ_w$ (módulo $ftv((\overline{\tau_j}^m) \to \tau)$). Por hipótesis de inducción obtendremos que $(v_j, \pi^\circ_w) \in \mathscr{T}\,[\![\tau_j]\!]$, para cada $j \in \{1..m\}$, y que $(v', \pi^\circ_w) \in \mathscr{T}\,[\![\tau]\!]$. Por lo tanto, tendremos que $(v, \pi^\circ) \in \mathscr{F}\,[\![(\overline{\tau_j}^m) \to \tau]\!]$.

   Con la unión de los dos casos para v obtenemos para el tipo funcional lo que queríamos demostrar.

2. **Esquemas de tipo funcional:**

   Supongamos que $(v,\pi) \in \mathscr{S}\,[\![\forall \overline{\alpha_k}.(\overline{\tau_j}^m) \to \tau]\!]$ y que $\pi \equiv \pi^\circ$ (módulo $ftv(\forall \overline{\alpha_k}.(\overline{\tau_j}^m) \to \tau)$). Por la definición de los esquemas de tipos, sabemos que $(v, \pi') \in \mathscr{F}\,[\![(\overline{\tau_j}^m) \to \tau]\!]$ y $\pi \equiv \pi'$ (módulo **TypeVar**$\setminus \{\overline{\alpha_k}\}$). Si suponemos una instanciación $\pi^\circ \equiv \pi^\bullet$ (módulo **TypeVar**$\setminus \{\overline{\alpha_k}\}$), tal que $\pi' \equiv \pi^\bullet$ (módulo $ftv((\overline{\tau_j}^m) \to \tau) \cup \{\overline{\alpha_k}\}$), aplicando el resultado de (1), tendremos que $(v, \pi^\bullet) \in \mathscr{F}\,[\![(\overline{\tau_j}^m) \to \tau]\!]$ y, por lo tanto, $(v, \pi^\circ) \in \mathscr{S}\,[\![\forall \overline{\alpha_k}.(\overline{\tau_j}^m) \to \tau]\!]$.

Ahora, suponemos que $(v,\pi) \in \mathscr{T}\,[\![\bigsqcup_{i=1}^n \sigma_i]\!]$ y $\pi \equiv \pi^\circ$ (módulo $ftv(\bigsqcup_{i=1}^n \sigma_i)$). Sabemos que $v = \bigcup_{i=1}^n f_i$ y que $(f_i, \pi) \in \mathscr{S}\,[\![\sigma_i]\!]$, para cada $i \in \{1..n\}$. Aplicando el resultado de (2), tendremos que $(f_i, \pi^\circ) \in \mathscr{S}\,[\![\sigma_i]\!]$, para cada $i \in \{1..n\}$. Por lo tanto, $(v, \pi^\circ) \in \mathscr{T}\,[\![\bigsqcup_{i=1}^n \sigma_i]\!]$.

- **Caso** $\pi \vDash \alpha \subseteq \tau'$.

  Suponemos que $\pi \vDash \alpha \subseteq \tau$. Por la definición de la restricción tendremos que:

  $$\pi(\alpha) \subseteq \{v \mid (v, \pi') \in \mathscr{T}\,[\![\tau]\!], \pi' \subseteq \pi\}$$

  Si suponemos una instanciación $\pi \equiv \pi^\circ$ (módulo $ftv(\alpha \subseteq \tau)$), sabemos que $\pi^\circ(\alpha) = \pi(\alpha)$. Supongamos ahora que una $\pi^\bullet \subseteq \pi^\circ$, tal que $\pi' \equiv \pi^\bullet$ (módulo $ftv(\alpha \subseteq \tau)$), por hipótesis de inducción tendríamos que $(v, \pi^\bullet) \in \mathscr{T}\,[\![\tau]\!]$, por lo tanto $\pi^\circ \vDash \alpha \subseteq \tau$.

- **Caso** $\pi \vDash c \subseteq \tau$.

  Este caso es análogo al anterior.

- **Caso** $\pi \vDash \alpha \Leftarrow \tau'$.

  Suponemos que $\pi \vDash \tau \Leftarrow \alpha$. Por la definición de la restricción tendremos que:

$$\exists (\pi_\nu)_{\nu \in \pi(\alpha)} : \pi = \bigcup_{\nu \in \pi(\alpha)} \pi_\nu \wedge (\forall \nu \in \pi(\alpha) : (\nu, \pi_\nu) \in \mathcal{T}\,[\![\tau]\!])$$

  Si suponemos una instanciación $\pi \equiv \pi'$ (módulo $ftv(\tau \Leftarrow \alpha)$), sabemos que $\pi'(\alpha) = \pi(\alpha)$ y la podemos descomponer como $\pi' = \bigcup_{\nu \in \pi'(\alpha)} \pi'_\nu$, tal que $\pi_\nu \equiv \pi'_\nu$ (módulo $ftv(\tau \Leftarrow \alpha)$). Por hipótesis de inducción tendríamos que $(\nu, \pi'_\nu) \in \mathcal{T}\,[\![\tau]\!]$, por lo tanto $\pi' \vDash \alpha \Leftarrow \tau'$.

□

4.4. Entornos y tipos anotados

Como ya ocurría en nuestro sistema monomórfico, con el fin de dar tipo a una expresión dada, tenemos que hacer seguimiento de los tipos de las variables que aparecen en la expresión. Un *entorno de tipos* Γ es un par (E, C), donde E es la función que asigna a cada variable de programa su correspondiente tipo—igual que la función que se vio en la sección 3.2 del sistema monomórfico—y C es un conjunto de restricciones que afectan a las variables de tipo existentes en E. Para simplificar la notación, usaremos $\Gamma(x)$ en lugar de $E(x)$ y usaremos $\Gamma|_C$ para denotar el conjunto de restricciones que hay en el lado derecho de Γ. El entorno $[\,]$ denota el par $(E, \emptyset)$ donde E asigna a todas las variables el tipo `any()`, mientras que $\bot$ denota el entorno $(E, \emptyset)$ donde E asigna a todas las variables el tipo `none()`. Usamos la notación $\Gamma\left[\overline{x_i : \tau_i}^n \mid C\right]$ para denotar el entorno Γ' tal que: $\Gamma'(x_i) = \tau_i$ para cada $i \in \{1..n\}$, $\Gamma'(z) = \Gamma(z)$ para cualquier otra $z \notin \{\overline{x_i}^n\}$, y $\Gamma'\big|_C = \Gamma|_C \cup C$. Como caso particular de lo anterior, usaremos $\left[\overline{x_i : \tau_i}^n \mid C\right]$ para abreviar $[\,]\left[\overline{x_i : \tau_i}^n \mid C\right]$.

La semántica de un entorno de tipos es el conjunto de todas las sustituciones que asignan a variables los valores que pertenecen a sus respectivos tipos. Siendo más precisos, dado un entorno Γ y una instanciación π, definimos $\mathcal{T}^{\pi}_{Env}\,[\![\Gamma]\!]$ como:

$$\mathcal{T}^{\pi}_{Env}\,[\![\Gamma]\!] = \left\{\theta \mid \forall x \in \mathbf{Var}.\, (\theta(x), \pi) \in \mathcal{T}\,[\![\Gamma(x)]\!]\,, \pi \vDash \Gamma|_C\right\}$$

Nótese que la instanciación utilizada por cada variable tiene que ser la misma que se usa para validar las restricciones en Γ. En el caso de que la instanciación no sea relevante, usamos $\theta \in \mathcal{T}_{Env}\,[\![\Gamma]\!]$ para denotar que $\theta \in \mathcal{T}^{\pi}_{Env}\,[\![\Gamma]\!]$ para algún π.

Las reglas del sistema de tipos—que describiremos en la sección 4.5—nos permiten derivar no solo un *success type* polimórfico para una expresión, sino también el entorno que describe las condiciones necesarias para poder evaluar dicha expresión. Es más, el *success type* de la expresión podría restringir las variables de tipo que aparecen en dicho entorno y viceversa. Por lo tanto, es conveniente agrupar un tipo y un entorno en un par único con su propia definición semántica. Además, si una

$$\rho ::= \langle \tau ; \Gamma \rangle \mid \rho ; \rho$$

$$\mathcal{T}\,[\![\langle \tau ; \Gamma \rangle]\!] = \left\{ (\theta, \upsilon) \mid \theta \in \mathcal{T}^{\pi}_{Env}\,[\![\Gamma]\!], (\upsilon, \pi) \in \mathcal{T}\,[\![\tau]\!] \right\}$$

$$\mathcal{T}\,[\![\rho_1 ; \rho_2]\!] = \mathcal{T}\,[\![\rho_1]\!] \cup \mathcal{T}\,[\![\rho_2]\!]$$

Figura 4.4: Sintaxis y semántica de los tipos anotados

expresión contiene un **case** o llamadas a función con un esquema de tipo sobrecargado, el sistema de tipos nos permitirá derivar una secuencia de dichos pares. Un *tipo anotado* es una secuencia de pares $\langle \tau_1 ; \Gamma_1 \rangle ; \ldots ; \langle \tau_n ; \Gamma_n \rangle$. La sintaxis y semántica de los tipos anotados se encuentra en la figura 4.4. La semántica de un tipo anotado es un conjunto de pares (θ, υ). Decimos que (θ, υ) pertenece a la semántica de $\langle \tau ; \Gamma \rangle$ si hay una instanciación π tal que (υ, π) pertenezca a la semántica de τ y θ pertenezca a la semántica de Γ para esa misma instanciación.

Definimos la siguiente relación de preorden con los entornos de tipos: $\Gamma_1 \subseteq \Gamma_2$ si y solo si $\mathcal{T}_{Env}\,[\![\Gamma_1]\!] \subseteq \mathcal{T}_{Env}\,[\![\Gamma_2]\!]$. Usamos $\Gamma_1 \approx \Gamma_2$ para denotar que Γ_1 y Γ_2 son equivalentes en su semántica. Podemos definir relaciones análogas para los tipos anotados. En algunas de las reglas de tipado haremos uso del operador de ínfimo ($\sqcap$) entre entornos. Es decir, dados Γ_1 y Γ_2, queremos encontrar Γ tal que $\mathcal{T}_{Env}\,[\![\Gamma]\!] = \mathcal{T}_{Env}\,[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}\,[\![\Gamma_2]\!]$. Este requisito motivó la necesidad de disponer de restricciones de la forma $\tau \Leftarrow \alpha$. Por ejemplo, consideremos el siguiente par de entornos:

$$\Gamma_1 = [\mathtt{X} : \mathtt{nelist}(\alpha_1, []), \mathtt{Y} : \alpha_2 \mid \alpha_2 \subseteq \alpha_1] \qquad \Gamma_2 = [\mathtt{X} : \alpha_3, \mathtt{Z} : \alpha_4 \mid \alpha_3 = \alpha_4]$$

El primero indica que `X` está ligada a una lista y que `Y` es uno de sus elementos. El segundo indica que `X` y `Z` están ligadas al mismo valor. Un candidato para la mayor cota inferior de estos dos entornos podría ser $\Gamma = [X : \alpha_3, \mathtt{Y} : \alpha_2, \mathtt{Z} : \alpha_4 \mid \alpha_3 = \alpha_4, \alpha_3 \subseteq \mathtt{nelist}(\alpha_1, []), \alpha_2 \subseteq \alpha_1]$, pero la semántica de Γ es estrictamente mayor que $\mathcal{T}_{Env}\,[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}\,[\![\Gamma_2]\!]$. De hecho, la sustitución $[\mathtt{X}/[\mathtt{1}], \mathtt{Y}/2, \mathtt{Z}/[\mathtt{1}]]$ pertenece a $\mathcal{T}^{\pi}_{Env}\,[\![\Gamma]\!]$ (si elegimos $\pi = [\alpha_1 \mapsto \{1,2\}, \alpha_2 \mapsto \{2\}, \alpha_3 \mapsto \{[\mathtt{1}]\}, \alpha_4 \mapsto \{[\mathtt{1}]\}]$) pero no pertenece a $\mathcal{T}_{Env}\,[\![\Gamma_1]\!]$. La razón para esta pérdida de precisión es que α_1 aparece en una posición instanciable de Γ_1, pero no así en Γ. Por lo tanto, siempre que encontremos $\theta \in \mathcal{T}^{\pi}_{Env}\,[\![\Gamma_1]\!]$, se cumple que $\pi(\alpha_1)$ contiene exactamente los elementos de la lista $\theta(\mathtt{X})$, pero en Γ el tipo de la variable α_1 puede instanciarse a cualquier superconjunto que contenga los elementos de $\theta(\mathtt{X})$, rompiendo así el requisito de que `Y` es uno de los elementos de `X`. Para evitar esto, la mayor cota inferior debería ser $[\mathtt{X} : \alpha_3, \mathtt{Y} : \alpha_2, \mathtt{Z} : \alpha_4 \mid \alpha_3 = \alpha_4, \mathtt{nelist}(\alpha_1, []) \Leftarrow \alpha_3, \alpha_2 \subseteq \alpha_1]$, que representa $\mathcal{T}_{Env}\,[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}\,[\![\Gamma_2]\!]$ sin pérdidas de precisión.

Para poder obtener una forma explícita de la mayor cota inferior de dos entornos o dos tipos anotados, necesitamos *normalizar* estos de antemano. La idea clave para normalizar un entorno es asignar a cada variable de programa una variable de tipo fresca, añadiendo una nueva restricción para relacionar

esas variables de tipo con los tipos del entorno sin normalizar:

$$norm\big(\big[\overline{x_i : \tau_i} \mid C\big]\big) = \big[\overline{x_i : \alpha_i} \mid C \cup \{\overline{\tau_i \Leftarrow \alpha_i}\}\big] \text{ donde } ftv\big(\big[\overline{x_i : \tau_i} \mid C\big]\big) \cap \{\overline{\alpha_i}\} = \emptyset$$

Es fácil demostrar que $norm(\Gamma) \approx \Gamma$. Entonces definimos el operador de ínfimo entre entornos ($\sqcap$), para combinar la información dada entre varios entornos en uno solo, de la siguiente forma:

$$\begin{aligned}\Gamma_1 \sqcap \Gamma_2 = \big[\overline{x_i : \alpha_i} \mid C_1 \cup C_2\big]\\ \text{donde } norm(\Gamma_1) = \big[\overline{x_i : \alpha_i} \mid C_1\big] \text{ y } norm(\Gamma_2) = \big[\overline{x_i : \alpha_i} \mid C_2\big]\\ \text{y } ftv(\Gamma_1) \cap ftv(\Gamma_2) = \emptyset\end{aligned}$$

Si la condición $ftv(\Gamma_1) \cap ftv(\Gamma_2) = \emptyset$ no se cumpliera, basta con renombrar las variables libres en uno de los dos entornos para evitar la colisión si fuere necesario. Al aseguramos que ninguna variable de tipo es común a excepción de las $\overline{\alpha_i}$, podremos unir los conjuntos de restricciones C_1 y C_2. Es sencillo demostrar que $\mathcal{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!] = \mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!]$ para cada Γ_1 y Γ_2.

Finalmente definimos el operador $\otimes$, que recibe una serie de pares $\langle \tau_1; \Gamma_1 \rangle, \ldots, \langle \tau_n; \Gamma_n \rangle$, como la unión de todos los τ_i en una tupla, junto al ínfimo de todos los entornos Γ_i:

$$\begin{aligned}\langle \tau_1; \Gamma_1 \rangle \otimes \cdots \otimes \langle \tau_n; \Gamma_n \rangle = \big\langle \{\tau_1, \ldots, \tau_n\}; \big[\overline{x_i : \alpha_i} \mid C_1 \cup \cdots \cup C_n\big] \big\rangle\\ \text{donde } norm\big(\Gamma_j\big) = \big[\overline{x_i : \alpha_i} \mid C_j\big] \text{ para todo } j \in \{1..n\}\\ \text{y } ftv\big(\big\langle \tau_j; \Gamma_j \big\rangle\big) \cap ftv(\langle \tau_k; \Gamma_k \rangle) = \emptyset \text{ para todo } j, k \in \{1..n\}, j \neq k\\ \text{y } ftv\big(\big\langle \tau_j; \Gamma_j \big\rangle\big) \cap \{\overline{\alpha_i}\} = \emptyset \text{ para todo } j \in \{1..n\}\end{aligned}$$

Podemos extender este operador a los tipos anotados—sobre las secuencias de pares—aplicando con $\otimes$ la propiedad distributiva a lo largo del operador ; que compone los tipos anotados:

$$\rho_1 \otimes \cdots \otimes (\rho_i; \rho_i') \otimes \cdots \otimes \rho_n = (\rho_1 \otimes \cdots \otimes \rho_i \otimes \cdots \otimes \rho_n); (\rho_1 \otimes \cdots \otimes \rho_i' \otimes \cdots \otimes \rho_n)$$

El operador $\otimes$ satisface la siguiente propiedad, sobre la cual se basan las reglas de tipo para tuplas y listas:

Proposición 10. *Para cada sustitución θ, valores $v_1, \ldots, v_n$ y tipos anotados $\rho_1, \ldots, \rho_n$, tales que $(\theta, v_i) \in \mathcal{T}[\![\rho_i]\!]$ para cada $i \in \{1..n\}$, se tiene que $\Big(\theta, \Big(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\Big)\Big) \in \mathcal{T}[\![\rho_1 \otimes \cdots \otimes \rho_n]\!]$.*

Demostración. Cuando ρ_i tiene la forma $\langle \tau_i; \Gamma_i \rangle$ para cada $i \in \{1..n\}$, por la definición sabemos que:

$$\begin{gathered}\Big(\theta, \Big(\{\overset{n}{\cdots}\}, \overline{v_i}^n\Big)\Big) \in \mathcal{T}[\![\rho_1 \otimes \cdots \otimes \rho_n]\!] \wedge \forall i \in \{1..n\}\,.\,(\theta, v_i) \in \mathcal{T}[\![\rho_i]\!]\\ \Updownarrow\\ \Big(\theta, \Big(\{\overset{n}{\cdots}\}, \overline{v_i}^n\Big)\Big) \in \mathcal{T}[\![\langle \tau_1; \Gamma_1 \rangle \otimes \cdots \otimes \langle \tau_n; \Gamma_n \rangle]\!] \wedge \forall i \in \{1..n\}\,.\,(\theta, v_i) \in \mathcal{T}[\![\langle \tau_i; \Gamma_i \rangle]\!]\end{gathered}$$

De la semántica de$\langle \tau_i;\Gamma_i\rangle$ tenemos:

$$\theta \in \mathcal{T}^{\pi_i}_{Env}[\![\Gamma_i]\!] \wedge (v_i,\pi_i) \in \mathcal{T}[\![\tau_i]\!]$$
$$\Updownarrow$$
$$\forall x \in \mathbf{Var}.\, (\theta(x),\pi_i) \in \mathcal{T}[\![\Gamma_i(x)]\!] \wedge \pi_i \vDash \Gamma_i|_C \wedge (v_i,\pi_i) \in \mathcal{T}[\![\tau_i]\!]$$

Primero normalizamos cada Γ_i en $\Gamma'_i = \left[\overline{x_j : \alpha_j} \mid \Gamma_i|_C \cup \left\{\overline{\Gamma_i(x_j) \Leftarrow \alpha_j}\right\}\right]$, de modo que $ftv(\langle\tau_i;\Gamma_i\rangle) \cap \{\overline{\alpha_j}\} = \emptyset$ para cada $i \in \{1..n\}$. Esto significa que necesitamos una nueva $\pi'_i = \pi_i\left[\overline{\alpha_j \mapsto \{\theta(x_j)\}}\right]$, y ya que $\pi'_i \equiv \pi_i$ (módulo $ftv\left(\Gamma_i|_C\right)$) con el lema 1 obtenemos $\pi'_i \vDash \Gamma_i|_C$. Dado que $\pi'_i(\alpha_j) = \{\theta(x_j)\}$, sabemos que:

$$\begin{aligned}\left(\theta(x_j),\pi'_i\right) \in \mathcal{T}[\![\alpha_j]\!] \quad &\Longleftrightarrow \quad \left(\theta(x_j),\pi'_i\right) \in \mathcal{T}[\![\Gamma'_i(x_j)]\!] \\ &\Longrightarrow \quad \forall x \in \mathbf{Var}.\left(\theta(x),\pi'_i\right) \in \mathcal{T}[\![\Gamma'_i(x)]\!]\end{aligned}$$

También tenemos que $\pi'_i \vDash \left\{\overline{\Gamma_i(x_j) \Leftarrow \alpha_j}\right\}$, ya que $v \in \pi'_i(\alpha_j)$ significa que cuando $v = \theta(x_j)$ entonces $\pi_v = \pi'_i$, además $(v,\pi_i) \in \mathcal{T}[\![\Gamma_i(x_j)]\!]$ puede ser transformado en $(v,\pi'_i) \in \mathcal{T}[\![\Gamma_i(x_j)]\!]$ usando el lema 1, porque $\pi'_i \equiv \pi_i$ (módulo $ftv\left(\Gamma_i(x_j)\right)$). Entonces con $\pi'_i \vDash \Gamma_i|_C$ y $\pi'_i \vDash \left\{\overline{\Gamma_i(x_j) \Leftarrow \alpha_j}\right\}$, tenemos que $\pi'_i \vDash \Gamma'_i\big|_C$ y podemos usar el lema 1 para obtener $(v_i,\pi'_i) \in \mathcal{T}[\![\tau_i]\!]$, por ello:

$$\forall x \in \mathbf{Var}.\left(\theta(x),\pi'_i\right) \in \mathcal{T}[\![\Gamma'_i(x)]\!] \wedge \pi'_i \vDash \Gamma'_i\big|_C \wedge \left(v_i,\pi'_i\right) \in \mathcal{T}[\![\tau_i]\!]$$

para todo $i \in \{1..n\}$. Ahora necesitamos usar un π' único definido como:

$$\pi'(\alpha) = \begin{cases} \{\theta(x_j)\} & \alpha \in \{\overline{\alpha_j}\} \\ \pi'_i(\alpha) & \alpha \in ftv(\langle\tau_i;\Gamma_i\rangle) \text{ para todo } i \in \{1..n\} \\ \emptyset & \text{en otro caso} \end{cases}$$

Dado que $ftv(\langle\tau_i;\Gamma_i\rangle) \cap \{\overline{\alpha_j}\} = \emptyset$ y $ftv(\langle\tau_i;\Gamma_i\rangle) \cap ftv(\langle\tau_k;\Gamma_k\rangle) = \emptyset$ para todo $i,k \in \{1..n\}$ y $i \neq k$, con lo que sabemos que π' está bien definida. Entonces, usando el lema 1:

$$\pi' \equiv \pi'_i(\text{módulo } ftv\left(\left\langle\tau_i;\Gamma'_i\right\rangle\right)) \quad \Longrightarrow \quad \begin{cases} \forall x \in \mathbf{Var}.\left(\theta(x),\pi'\right) \in \mathcal{T}[\![\Gamma'_i(x)]\!] \\ \pi' \vDash \Gamma'_i\big|_C \\ \left(v_i,\pi'\right) \in \mathcal{T}[\![\tau_i]\!] \end{cases}$$

Con estos resultados, obtenemos por la definición que:

$$\forall i \in \{1..n\}.\left(v_i,\pi'\right) \in \mathcal{T}[\![\tau_i]\!] \quad \Longleftrightarrow \quad \left(\left\{\overset{n}{\cdot\cdot}\right\},\overline{v_i}^n\right),\pi'\right) \in \mathcal{T}[\![\{\overline{\tau_i}^n\}]\!] \tag{4.1}$$

Después de obtener la semántica para la tupla, necesitamos unificar los entornos Γ' tal que $\Gamma' =$

$[\overline{x_j : \alpha_j} \mid \Gamma'_1\big|_C \cup \cdots \cup \Gamma'_n\big|_C]$. Puesto que $\Gamma'(x_j) = \Gamma'_i(x_j)$ para cada $i \in \{1..n\}$:

$$\forall x \in \mathbf{Var}.\, (\theta(x), \pi') \in \mathcal{T}\,[\![\Gamma'_i(x)]\!] \iff \forall x \in \mathbf{Var}.\, (\theta(x), \pi') \in \mathcal{T}\,[\![\Gamma'(x)]\!]$$

que nos lleva a:

$$\begin{array}{c} \forall x \in \mathbf{Var}.\, (\theta(x), \pi') \in \mathcal{T}\,[\![\Gamma'(x)]\!] \wedge \pi' \vDash \Gamma'_1\big|_C \wedge \cdots \wedge \pi' \vDash \Gamma'_n\big|_C \\ \Updownarrow \\ \theta \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma']\!] \end{array} \tag{4.2}$$

Finalmente con 4.1 y 4.2 obtenemos:

$$\left(\theta, \left(\{\overset{n}{\cdots}\}, \overline{v_i}^n\right)\right) \in \mathcal{T}\,[\![\langle \{\overline{\tau_i}^n\}; \Gamma'\rangle]\!] = \mathcal{T}\,[\![\langle \tau_1; \Gamma_1\rangle \otimes \cdots \otimes \langle \tau_n; \Gamma_n\rangle]\!]$$

demostrando el caso en el que cada π tiene la forma $\langle \tau_i; \Gamma_i\rangle$.

Cuando ρ_j tiene la forma $\rho'_j; \rho''_j$ para algún $j \in \{1..n\}$, por la definición sabemos:

$$\rho_1 \otimes \cdots \otimes (\rho'_j; \rho''_j) \otimes \cdots \otimes \rho_n = \rho_1 \otimes \cdots \otimes \rho'_j \otimes \cdots \otimes \rho_n; \rho_1 \otimes \cdots \otimes \rho''_j \otimes \cdots \otimes \rho_n$$

y también sabemos por la definición:

$$\begin{array}{rcl} (\theta, v_j) \in \mathcal{T}\,[\![\rho_j]\!] & \iff & (\theta, v_j) \in \mathcal{T}\left[\!\left[\rho'_j; \rho''_j\right]\!\right] \\ & \iff & (\theta, v_j) \in \mathcal{T}\left[\!\left[\rho'_j\right]\!\right] \cup \mathcal{T}\left[\!\left[\rho''_j\right]\!\right] \end{array}$$

Por lo tanto, tenemos dos subcasos:

- Cuando $(\theta, v_j) \in \mathcal{T}\left[\!\left[\rho'_j\right]\!\right]$, por hipótesis de inducción con $(\theta, v_i) \in \mathcal{T}\,[\![\rho_i]\!]$ para cada $i \in \{1..n\}$ cuando $i \neq j$, obtenemos:

$$\begin{array}{l} \left(\theta, \left(\{\overset{n}{\cdots}\}, \overline{v_i}^n\right)\right) \in \mathcal{T}\left[\!\left[\rho_1 \otimes \cdots \otimes \rho'_j \otimes \cdots \otimes \rho_n\right]\!\right] \\ \qquad \subseteq \mathcal{T}\left[\!\left[\rho_1 \otimes \cdots \otimes \rho'_j \otimes \cdots \otimes \rho_n; \rho_1 \otimes \cdots \otimes \rho''_j \otimes \cdots \otimes \rho_n\right]\!\right] \end{array}$$

- Cuando $(\theta, v_j) \in \mathcal{T}\left[\!\left[\rho''_j\right]\!\right]$, por hipótesis de inducción con $(\theta, v_i) \in \mathcal{T}\,[\![\rho_i]\!]$ para cada $i \in \{1..n\}$ cuando $i \neq j$, obtenemos:

$$\begin{array}{l} \left(\theta, \left(\{\overset{n}{\cdots}\}, \overline{v_i}^n\right)\right) \in \mathcal{T}\left[\!\left[\rho_1 \otimes \cdots \otimes \rho''_j \otimes \cdots \otimes \rho_n\right]\!\right] \\ \qquad \subseteq \mathcal{T}\left[\!\left[\rho_1 \otimes \cdots \otimes \rho'_j \otimes \cdots \otimes \rho_n; \rho_1 \otimes \cdots \otimes \rho''_j \otimes \cdots \otimes \rho_n\right]\!\right] \end{array}$$

demostrando el caso actual y la proposición. □

4.5. Reglas de derivación de tipos polimórficos

Con las reglas de tipado mostradas en esta sección podremos obtener tipos anotados para cada expresión e. Sin embargo, al igual que en el sistema de tipos monomórfico expuesto en el capítulo 3, sería conveniente añadir a nuestros juicios un entorno inicial con el que reflejar algunas condiciones—ya conocidas—sobre las variables libres de la expresión e. Por lo tanto, nuestros juicios tendrán la forma $\Gamma \vdash e : \rho$, con el siguiente significado: suponiendo que los valores de las variables libres en e (dados por una sustitución θ) están contenidos en sus correspondientes tipos almacenados en Γ, si e es evaluada al valor v, entonces el par (θ, v) pertenece a la semántica de ρ. Para ser más precisos, si $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$ y $(\theta, v) \in \mathscr{E}[\![e]\!]$ entonces $(\theta, v) \in \mathcal{T}[\![\rho]\!]$, que puede ser expresado de forma sucinta como $\mathscr{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\rho]\!]$. También usaremos el término *entorno de entrada* para referirnos a Γ y *entorno final* para referirnos a cualquier Γ' dentro de un par en la secuencia representada por ρ.

Las reglas de tipado están en la figura 4.5. Las dos primeras son reglas auxiliares para fortalecer o debilitar elementos en nuestros juicios. Mientras [SUB1] especifica que podemos reemplazar el entorno de entrada Γ' por uno más fuerte (es decir, más restrictivo), [SUB2] nos permite debilitar un tipo anotado ρ.

Las reglas [CNS] y [VAR] especifican que el entorno final dentro del par—que da tipo a la expresión—no incluye nuevas restricciones que no estuvieran ya en el entorno de entrada. La regla [TPL] fusiona los tipos anotados de cada subexpresión con el operador $\otimes$, cuya definición dimos en la sección 4.4, para obtener como resultado un tipo tupla. La regla [LST] hace lo mismo para constructores de listas, pero convirtiendo el tipo tupla de cada par—obtenido con el operador $\otimes$—en un tipo `nelist` para formar el tipo anotado del resultado.

En cuanto a la regla [ABS], el entorno final es el mismo que el de entrada, ya que la evaluación de una λ-abstracción siempre tiene éxito. Usamos las variables de tipo $\overline{\alpha_i}$ para denotar los tipos de las variables libres $\overline{y_i}$ en la λ-abstracción. Después de analizar el cuerpo de la λ-abstracción, obtenemos un tipo anotado. Cada par en el tipo anotado contendrá un tipo τ_j como resultado de la función y un tipo $\tau_{j,i}$ por cada parámetro x_i dentro del entorno final. Exigimos a cada entorno final en el tipo anotado que sea igual al entorno de entrada, excepto por los tipos de los parámetros $\overline{x_i}^n$. Con cada par construiremos un tipo funcional, que será generalizado usando el operador $\overline{\forall}$. Este operador liga todas las variables de tipo instanciables dentro del tipo funcional. Finalmente, obtenemos una colección de esquemas de tipos funcionales que usaremos para construir el tipo funcional sobrecargado con $\sqcup$. La condición $\forall j \in \{1..m\} : ftv\left(\overline{\forall}\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \subseteq \left\{\overline{\alpha_i}^l\right\}$ comprueba que las variables de tipo libres de cada esquema existan previamente en Γ y para que sea posible, para cada una de las variables de tipo α_i ha de existir una variable de programa en el entorno que tenga esa α_i como tipo.

Tenemos dos reglas para aplicar funciones: [APP1] solo tendrá sentido cuando el tipo supuesto en el entorno inicial para f es compatible con un tipo funcional, mientras que [APP2] especifica que la evaluación de la expresión fallará en caso contrario. En el primer caso, el resultado consiste en un tipo anotado que contiene tantos pares como sobrecargas haya en el tipo de f. Cada par contiene un tipo

$$\frac{\Gamma' \vdash e : \rho \qquad \Gamma \subseteq \Gamma'}{\Gamma \vdash e : \rho} \text{ [SUB1]} \qquad \frac{\Gamma \vdash e : \rho \qquad \rho \subseteq \rho'}{\Gamma \vdash e : \rho'} \text{ [SUB2]}$$

$$\frac{}{\Gamma \vdash c : \langle c; \Gamma \rangle} \text{ [CNS]} \qquad \frac{}{\Gamma \vdash x : \langle \Gamma(x); \Gamma \rangle} \text{ [VAR]} \qquad \frac{\Gamma \vdash e_i : \rho_i}{\Gamma \vdash \{\overline{e_i}^n\} : \rho_1 \otimes \cdots \otimes \rho_n} \text{ [TPL]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \rho_1 \qquad \Gamma \vdash e_2 : \rho_2 \\ \rho_1 \otimes \rho_2 = \overline{\langle \{\tau_i, \tau_i'\}; \Gamma_i \rangle}^n\end{array}}{\Gamma \vdash [e_1 \mid e_2] : \overline{\langle \texttt{nelist}(\tau_i, \tau_i'); \Gamma_i \rangle}^n} \text{ [LST]} \qquad \frac{\begin{array}{c}\Gamma = \Gamma_0 \left[\overline{x_i : \texttt{any}()}^n, \overline{y_i : \alpha_i}^l\right] \\ \Gamma \vdash e : \overline{\langle \tau_j; \Gamma \left[\overline{x_i : \tau_{j,i}}^n\right] \rangle}^m\end{array}}{\Gamma \vdash \textbf{fun}(\overline{x_i}^n) \to e : \left\langle \bigsqcup_{j=1}^m \overline{\forall}\left(\overline{\tau_{j,i}}^n\right) \to \tau_j; \Gamma \right\rangle} \text{ [ABS]}$$

donde $\overline{\forall}\tau \stackrel{\text{def}}{=} \forall \overline{\alpha_i}.\tau$ y $\{\overline{\alpha_i}\} = itv(\tau)$.

$$\frac{\begin{array}{c}\Gamma \sqcap \left[f : \left(\overline{\texttt{any}()}^n\right) \to \texttt{any}()\right] \approx \Gamma_0 \\ \Gamma_0(f) = \bigsqcup_{j=1}^m \left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \qquad \forall i \in \{1..n\} : \Gamma_0(x_i) = \alpha_{x_i} \\ \forall j \in \{1..m\} : \mu_j = \mathit{freshRenaming}\left(itv\left(\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \cup \{\overline{\alpha_{j,i}}\}\right) \\ \forall j \in \{1..m\} : \Gamma_j = \Gamma_0 \left[\{\overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^n\} \cup \{\beta\mu_j \subseteq \beta \mid \beta \in itv\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)\}\right]\end{array}}{\Gamma \vdash f(\overline{x_i}^n) : \overline{\langle \tau_j \mu_j; \Gamma_j \rangle}^m} \text{ [APP1]}$$

$$\frac{\Gamma \sqcap \left[f : \left(\overline{\texttt{any}()}^n\right) \to \texttt{any}()\right] \approx \bot}{\Gamma \vdash f(\overline{x_i}^n) : \langle \texttt{none}(); \bot \rangle} \text{ [APP2]} \qquad \frac{\forall i \in \{1..n\} : \Gamma[x : \alpha] \Vdash_\alpha cls_i : \rho_i}{\Gamma[x : \alpha] \vdash \textbf{case } x \textbf{ of } \overline{cls_i}^n : \overline{\rho_i}^n} \text{ [CAS]}$$

$$\frac{\begin{array}{c}cls_i = \left(p_i \textbf{ when } e_i \to e_i'\right) \qquad \mathit{fresh}(\alpha) \\ \Gamma \sqcap [x_t : \texttt{integer}() \cup \texttt{'infinity'}] \Vdash_\alpha cls_i : \rho_i \qquad \Gamma \sqcap [x_t : \texttt{integer}()] \vdash e : \rho\end{array}}{\Gamma \vdash \textbf{receive } \overline{cls_i}^n \textbf{ after } x_t \to e : \overline{\rho_i}^n; \rho} \text{ [RCV]}$$

$$\frac{\begin{array}{c}\Gamma\left[\overline{x_i : \alpha_i}\right] \vdash p : \tau \qquad \{\overline{x_i}\} = \mathit{vars}(p) \\ \Gamma\left[\overline{x_i : \alpha_i} \mid \tau \Leftarrow \alpha\right] \vdash e_g : \overline{\langle \tau_i'; \Gamma_i' \rangle}^n \qquad \forall i \in \{1..n\} : \Gamma_i'\left['\texttt{true}' \subseteq \tau_i'\right] \vdash e : \rho_i\end{array}}{\Gamma\left[\overline{x_i : \texttt{any}()}\right] \Vdash_\alpha p \textbf{ when } e_g \to e : \overline{\rho_i \setminus \{\overline{x_i}\}}^n} \text{ [CLS]}$$

$$\frac{\begin{array}{c}\Gamma[x : \texttt{any}()] \vdash e_1 : \overline{\langle \tau_i; \Gamma_i \rangle}^n \\ \forall i \in \{1..n\} : \Gamma_i[x : \tau_i] \vdash e_2 : \rho_i\end{array}}{\Gamma[x : \texttt{any}()] \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \overline{\rho_i \setminus \{x\}}^n} \text{ [LET]} \qquad \frac{\begin{array}{c}\forall i \in \{1..n\} : \Gamma\left[\overline{x_i : \tau_i}^n\right] \vdash f_i : \langle \tau_i; \Gamma\left[\overline{x_i : \tau_i}^n\right] \rangle \\ \Gamma\left[\overline{x_i : \tau_i}^n\right] \vdash e : \rho\end{array}}{\Gamma\left[\overline{x_i : \texttt{any}()}^n\right] \vdash \textbf{letrec } \overline{x_i = f_i}^n \textbf{ in } e : \rho \setminus \{\overline{x_i}^n\}} \text{ [LRC]}$$

$$\frac{\Gamma \vdash p_1 : \tau_1 \qquad \Gamma \vdash p_2 : \tau_2}{\Gamma \vdash [p_1 \mid p_2] : \texttt{nelist}(\tau_1, \tau_2)} \text{ [LST}_P\text{]} \qquad \frac{\forall i \in \{1..n\} : \Gamma \vdash p_i : \tau_i}{\Gamma \vdash \{\overline{p_i}^n\} : \{\overline{\tau_i}^n\}} \text{ [TPL}_P\text{]}$$

$$\frac{}{\Gamma[x : \alpha_x] \vdash x : \alpha_x} \text{ [VAR}_P\text{]} \qquad \frac{}{\Gamma \vdash c : c} \text{ [LIT}_P\text{]}$$

Figura 4.5: Reglas de tipado para expresiones y cláusulas

τ_j como resultado de la función y una modificación del entorno de entrada, donde se han añadido nuevas restricciones para delimitar los tipos de los argumentos $\overline{x_i}^n$. Para poder aplicar esta regla, el entorno inicial Γ_0 que obtenemos desde el entorno de entrada debe satisfacer ciertas condiciones: la primera es que $\Gamma_0(f)$ ha de ser un tipo funcional (posiblemente sobrecargado) y la segunda es que todos los argumentos $\overline{x_i}^n$ deben estar asociados con variables de tipo $\overline{\alpha_{x_i}}^n$ en el entorno inicial. Si en el entorno Γ alguno de los tipos de los argumentos no fuera una variable de tipo, podríamos utilizar la regla [SUB1] para normalizar el entorno antes de aplicar la regla [APP1].

Para poder evitar colisiones entre variables, cuando se aplique la regla [APP1], tenemos que crear un grupo de *renombramientos* $\overline{\mu_j}^m$ para cada sobrecarga. Un renombramiento μ es una función parcial inyectiva, que toma unas variables de tipo y devuelve otras variables de tipo. Suponemos la existencia de la función *freshRenaming* que, dado un conjunto de variables de tipo, devuelve una tabla para renombrar aquellas variables del conjunto por variables frescas. La notación $\tau\mu$ denota un tipo con la misma estructura que τ en el que las variables de tipo han sido renombradas acorde con la información en μ. El entorno final Γ_j mostrado en [APP1] añade las siguientes restricciones a Γ_0: cada variable α_{x_i} correspondiente al i-ésimo argumento deberá encajar con el tipo $\tau_{j,i}$ del correspondiente parámetro y para cada variable de tipo instanciable β—dentro del esquema de tipo funcional—, la variable fresca generada por el renombramiento ha de ser subconjunto de la original.

Ilustremos con el siguiente ejemplo la regla [APP1]: supongamos la expresión `F(Z)` donde `F` está ligada a la función **fun**(`X`) → `{X,X+Y}`. El entorno Γ_0 asigna a `F` el tipo $\forall\alpha.(\alpha) \to$ `{`α`,number()}` **when** $\alpha \subseteq$ `number()`, $\beta \subseteq$ `number()`, `Y` al tipo β y `Z` al tipo $\alpha_{\texttt{Z}}$. Suponemos que el renombramiento μ creado con *freshRenaming* es $\mu = [\alpha/\alpha']$. Entonces, para construir Γ_1 añadimos a Γ_0 la restricción $\alpha' \Leftarrow \alpha_{\texttt{Z}}$. Como no tenemos variables de tipo instanciables libres en el tipo de `F`, no añadimos más restricciones. Como resultado obtenemos el tipo $\langle$`{`α'`,number()}` **when** $\alpha' \subseteq$ `number()`, $\beta \subseteq$ `number()`$;\Gamma_0[\alpha' \Leftarrow \alpha_{\texttt{Z}}]\rangle$ para `F(Z)`, que añade nueva información a las variables de programa `Z` e `Y`. Más concretamente, se indica que $\alpha_{\texttt{Z}}$ ha de coincidir con α', lo que implica que el valor de la variable `Z` ha de coincidir con la primera componente de la tupla resultante de evaluar la expresión `F(Z)`, y además ambos han de ser tipos numéricos. Por otro lado, también se indica que la variable `Y` ha de tener un valor numérico.

Nuestro segundo ejemplo con la regla [APP1] es `F(X)` cuando `F` tiene el tipo $(\alpha) \to$ `bool()`. Esta situación ocurre, por ejemplo en la función `filter`, cuando `F` es una función de predicado recibida como parámetro. Suponemos que la variable `X` tiene el tipo β en Γ_0 y μ es $[\alpha/\alpha']$. Para construir Γ_1 añadimos a Γ_0 la restricción $\alpha' \Leftarrow \beta$ y—como tenemos una variable de tipo instanciable libre dentro del tipo para `F`—añadimos también $\alpha' \subseteq \alpha$ para conectar la variable renombrada con la variable original que está libre en el tipo funcional. Como resultado obtenemos para `F(X)` el tipo $\langle$`bool()`$;\Gamma_0[\alpha' \Leftarrow \beta, \alpha' \subseteq \alpha]\rangle$. Esto implica que β ha de ser subconjunto de α y que, por tanto, tiene que estar ligada a un valor contenido en el dominio de `F`.

Para poder derivar un tipo para una expresión **case** o **receive**, tenemos que derivar un tipo para cada una de sus cláusulas. Con la regla [CLS] obtenemos juicios de la forma $\Gamma \Vdash_\alpha cls : \rho$; donde α es una variable de tipo, que es el tipo del discriminante de la expresión **case** o una variable de tipo fresca en

el caso de la expresión **receive**. La regla [CLS] exige al tipo del discriminante que sea compatible con el tipo del patrón y al tipo de la guarda que contenga el átomo `'true'`. De cara a satisfacer el primer requisito, ligamos las variables que aparecen en el patrón p_i a variables de tipo frescas, entonces derivamos el tipo del patrón—usando las reglas [LIT$_P$], [VAR$_P$], [LST$_P$], y [TPL$_P$]—y finalmente añadimos una restricción de encaje entre el tipo del discriminante y el tipo del patrón. El entorno resultante es usado para analizar la guarda y obtenemos un tipo anotado $\overline{\langle \tau'_i ; \Gamma'_i \rangle}^n$. Para cada par en el tipo anotado de la guarda, usaremos como entorno de entrada Γ'_i con una restricción extra que comprueba si τ'_i contiene el átomo `'true'`. Entonces derivamos el tipo anotado ρ_i para el cuerpo de la cláusula. Finalmente, concatenamos todos los ρ_i obtenidos, sin tener en cuenta la información relativa a las variables de patrón. La notación $\rho \setminus \{\overline{x_i}\}$ devuelve un tipo anotado que es resultado de reemplazar todos los entornos Γ en ρ por $\Gamma \left[\overline{x_i : \mathsf{any}()}\right]$.

La regla [RCV] es similar a la [CAS], pero—de cara a dar tipo a las cláusulas—exige que la variable x_t tenga que ser de un tipo dentro de `integer()` $\cup$ `'infinity'`, y—de cara a dar tipo al cuerpo de la expresión **after**—exige que la variable x_t no contenga `'infinity'`, ya que en ese caso la cláusula **after** no sería evaluada en tiempo de ejecución.

En la regla [LET] obtenemos un tipo anotado $\overline{\langle \tau_i ; \Gamma_i \rangle}^n$ para la expresión auxiliar e_1. Esto nos permite tomar Γ_i, sobrescribir el tipo para x con τ_i, y usar este entorno modificado como entorno de entrada para la expresión principal e_2. Para cada par obtenido a partir de analizar la expresión ligada, obtendremos un tipo anotado ρ_i para la expresión principal. Finalmente, anotaremos toda la expresión **let** con la concatenación de cada ρ_i, eliminando la información relativa a la variable ligada x.

La regla [LRC] funciona como la regla [LET] con una diferencia principal: para analizar la expresión ligada f_i, enlazamos $\overline{x_i}^n$ a los tipos $\overline{\tau_i}^n$ en el entorno de entrada para analizar cada expresión ligada. Ya que cada f_i es una λ-abstracción, sabemos que el resultado será un par con un tipo y el mismo entorno que usamos como entorno de entrada; además exigimos que el tipo inferido para cada f_i sea el tipo τ_i que se asumió inicialmente.

Una vez explicadas las reglas del sistema de tipos, nuestro primer resultado indica que siempre podemos encontrar una derivación de tipo para una expresión dada, a semejanza de lo que ocurría con los tipos monomórficos (proposición 5):

Proposición 11. *Dada una expresión e cualquiera y un entorno inicial* Γ*, existe un tipo anotado* ρ *tal que* $\Gamma \vdash e : \rho$*. En particular:* $\Gamma \vdash e : \langle \mathsf{any}(); [] \rangle$.

Demostración. Sencilla, inspeccionando las reglas de tipado. Las condiciones secundarias que involucran la relación de inclusión $\subseteq$, entre entornos o tipos anotados, siempre pueden satisfacer eligiendo $[]$ y $\langle \mathsf{any}(); [] \rangle$ respectivamente en el lado derecho de dichas condiciones. Si la condición secundaria de [APP1] que pide a f ser un esquema de tipo no es cierta, entonces la regla [APP2] puede ser aplicada. La condición secundaria de [APP1] que involucra a $x_1, \ldots, x_n$ siempre se cumple, porque en cualquier entorno si x está relacionado con un tipo τ, una variable de tipo fresca α puede ser asignada

a x y el tipo puede ser movido a las restricciones del entorno con la restricción $\tau \Leftarrow \alpha$. El resto de condiciones secundarias de [ABS], que involucran variables de tipo libres en el esquema de tipo, depende del tipo anotado para e, que puede ser modificado usando [SUB2] para construir un tipo más grande con menos restricciones si fuera necesario.

Una vez que demostramos que hemos derivado el juicio $\Gamma \vdash e : \rho$ para algún ρ, y dado que $\rho \subseteq \langle \texttt{any()}; [] \rangle$, podemos usar la regla [SUB2] para obtener $\Gamma \vdash e : \langle \texttt{any()}; [] \rangle$. □

Comparando este sistema con las reglas de derivación del sistema monomórfico de la sección 3.4, vemos que gracias a las variables de tipo hemos podido simplificar algunas reglas. En el caso de las reglas para dar tipo a una cláusula o una expresión **receive**, vemos que hemos reducido a una regla para cada caso. Esto se ha logrado porque hemos podido colocar aquellas condiciones que llevarían el resultado `none()` como restricciones en los tipos anotados finales. Es cierto que, como efecto colateral para la regla [CLS], optamos por añadir reglas para dar tipo a los patrones. Dadas las limitaciones sintácticas de los patrones, las reglas para dar tipos a estos son versiones simplificadas de otras reglas, ya que sabemos que vamos a recibir un tipo τ sin restricciones; tampoco hay necesidad de usar tipos anotados porque no hay ramificaciones en la evaluación de un patrón. Pero, sin duda, el cambio más llamativo entre las reglas del sistema monomórfico y las del actual, es la desaparición de la regla [TRANS], que utilizábamos para volver a evaluar una expresión con un nuevo entorno de entrada obtenido de un análisis previo. Al usar variables de tipo y restricciones, basta con asignar a una variable de programa x una variable de tipo α como su tipo general, para que todas las condiciones que van a afectar a x se vayan almacenado como restricciones sobre α, sin perder información.

Para entender mejor por qué ya no necesitamos la regla [TRANS], vamos a rescatar el ejemplo que vimos en la sección 3.5.1, donde teníamos la λ-abstracción **fun**(`V0`) $\rightarrow$ **let** `V1` = `1` **in** `{V0, '+'(V0,V1)}`. El problema inicial era que al analizar la tupla, el resultado de la primera componente era de tipo `any()` y el de la segunda era `number()`, porque no era hasta que se analizaba la aplicación de la función de la suma que obteníamos que `V0` era de tipo `number()` también. Bien, ahora vamos a analizar la tupla con el sistema polimórfico. Supongamos que en el entorno de entrada, `V0` es de tipo α y `V1` es de tipo β. El tipo de la primera componente del par `{V0, '+'(V0,V1)}` será α, sin cambiar el entorno de entrada. Pero con la segunda componente, si bien el tipo vuelve a ser `number()`, el entorno de salida incorpora las restricciones `number()` $\Leftarrow \alpha$ y `number()` $\Leftarrow \beta$. Esto nos lleva a que la tupla tiene el siguiente tipo anotado:

$$\langle \{\alpha, \texttt{number()}\}; \Gamma[\texttt{V0} : \alpha, \texttt{V1} : \beta \mid \texttt{number()} \Leftarrow \alpha, \texttt{number()} \Leftarrow \beta] \rangle$$

Este tipo nos dice que tenemos una tupla donde la primera componente es una variable de tipo que encaja en el tipo `number()`, igual que la segunda componente de la tupla. No hemos tenido que volver a analizar la tupla para poder conectar el tipo de la primera componente con el de la variable `V0` y esta con el tipo `number()`, gracias a la restricción de encaje sobre α.

$$
\begin{array}{rl}
\text{[ABS]:} & [\,] \vdash \mathbf{fun}(\mathtt{D}) \to \cdots : \langle (\mathtt{atom}()) \to \mathtt{'plastic'} \sqcup \forall\alpha.(\alpha) \to \{\mathtt{'love'}, \alpha\}; [\,]\rangle \\
\text{[SUB1]:} & [\,] \vdash \mathbf{case}\ \mathtt{D}\ \mathbf{of}\ \cdots : \langle \mathtt{'plastic'}; [\mathtt{D} : \mathtt{atom}()]\rangle; \langle \{\mathtt{'love'}, \alpha\}; [\mathtt{D} : \alpha]\rangle \\
\text{[CAS]:} & [\mathtt{D} : \alpha] \vdash \mathbf{case}\ \mathtt{D}\ \mathbf{of}\ \cdots : \langle \mathtt{'plastic'}; [\mathtt{D} : \mathtt{atom}()]\rangle; \langle \{\mathtt{'love'}, \alpha\}; [\mathtt{D} : \alpha]\rangle \\
\text{[CLS]:} & [\mathtt{D} : \alpha] \Vdash_{\alpha} \mathtt{M}\ \mathbf{when}\ \cdots : \langle \mathtt{'plastic'}; [\mathtt{D} : \mathtt{atom}()]\rangle \\
\text{[APP1]:} & [\mathtt{D} : \alpha, \mathtt{M} : \beta \mid \beta \Leftarrow \alpha] \vdash \mathtt{is\_atom}(\mathtt{M}) : \big\langle \mathtt{'true'}; [\mathtt{D} : \alpha, \mathtt{M} : \beta \mid \beta \Leftarrow \alpha, \mathtt{atom}() \Leftarrow \beta]\big\rangle; \\
& \quad \big\langle \mathtt{'false'}; [\mathtt{D} : \alpha, \mathtt{M} : \beta \mid \beta \Leftarrow \alpha, \mathtt{any}() \Leftarrow \beta]\big\rangle \\
(*)\ \text{[SUB2]:} & [\mathtt{D} : \alpha, \mathtt{M} : \beta \mid \beta \Leftarrow \alpha, \mathtt{atom}() \Leftarrow \beta, \mathtt{'true'} \subseteq \mathtt{'true'}] \vdash \mathtt{'plastic'} : \\
& \quad \langle \mathtt{'plastic'}; [\mathtt{D} : \mathtt{atom}(), \mathtt{M} : \mathtt{atom}()]\rangle \\
\text{[SUB2]:} & [\mathtt{D} : \alpha, \mathtt{M} : \beta \mid \beta \Leftarrow \alpha, \mathtt{atom}() \Leftarrow \beta, \mathtt{'true'} \subseteq \mathtt{'false'}] \vdash \mathtt{'plastic'} : \langle \mathtt{none}(); \bot\rangle \\
\text{[CLS]:} & [\mathtt{D} : \alpha] \Vdash_{\alpha} \mathtt{T}\ \mathbf{when}\ \cdots : \langle \{\mathtt{'love'}, \alpha\}; [\mathtt{D} : \alpha]\rangle \\
(**)\ \text{[SUB1]:} & [\mathtt{D} : \alpha, \mathtt{T} : \beta \mid \beta \Leftarrow \alpha, \mathtt{'true'} \subseteq \mathtt{'true'}] \vdash \{\mathtt{'love'}, \mathtt{T}\} : \langle \{\mathtt{'love'}, \alpha\}; [\mathtt{D} : \alpha, \mathtt{T} : \alpha]\rangle \\
\text{[TPL]:} & [\mathtt{D} : \alpha, \mathtt{T} : \alpha] \vdash \{\mathtt{'love'}, \mathtt{T}\} : \langle \{\mathtt{'love'}, \alpha\}; [\mathtt{D} : \alpha, \mathtt{T} : \alpha]\rangle
\end{array}
$$

Figura 4.6: Algunos juicios de la derivación del ejemplo en la sección 4.6.1

4.6. Ejemplos de derivación de tipos polimórficos

En esta sección mostraremos al lector algunos ejemplos y los tipos obtenidos en cada uno de ellos con nuestro sistema de tipos. Para mantener relativamente corto el tamaño de los ejemplos, no mostraremos el uso de [CNS], [VAR], [CNS$_P$], [VAR$_P$], [TPL$_P$] y [LST$_P$], porque su uso es trivial.

4.6.1. Usando funciones en las guardas

Para este ejemplo usaremos la función `is_atom` con el siguiente tipo:

$$\tau_{\mathtt{is\_atom}} = (\mathtt{atom}()) \to \mathtt{'true'} \sqcup (\mathtt{any}()) \to \mathtt{'false'}$$

Este tipo lo recibimos como parte del entorno inicial de entrada.[2] Suponemos la siguiente expresión escrita en *Mini Erlang* como candidata a analizar:

```
fun(D) → case D of
    M when is_atom(M) → 'plastic'
    T when 'true' → {'love',T}
end
```

[2] Para mantener la mayor brevedad posible, no mostraremos la entrada de `is_atom` en los entornos de los juicios mostrados en este ejemplo.

Obtenemos $(\texttt{atom()}) \to \texttt{'plastic'} \sqcup \forall\alpha.(\alpha) \to \{\texttt{'love'}, \alpha\}$ como un posible tipo para esta expresión con nuestras reglas de tipado. Una selección de juicios de la derivación es la mostrada en la figura 4.6. Hablemos de la aplicación de la regla [SUB2] marcada con el símbolo (*). Esta regla empieza con el entorno $\Gamma_0 = [\texttt{D} : \alpha, \texttt{M} : \beta \mid \beta \Leftarrow \alpha, \texttt{atom()} \Leftarrow \beta, \texttt{'true'} \subseteq \texttt{'true'}]$, donde `D` es el parámetro de la función y `M` es la variable introducida por la primera cláusula de la expresión **case**. Bajo Γ_0, el átomo `'plastic'` es tipado por la regla [CNS] con el par $\langle \texttt{'plastic'}; \Gamma_0 \rangle$. Este tipo anotado se puede transformar usando [SUB2], donde el primer paso sería quitar la restricción $\texttt{'true'} \subseteq \texttt{'true'}$ ya que se cumple que $\Gamma[\texttt{'true'} \subseteq \texttt{'true'}] \approx \Gamma$ para todo Γ, con lo que la restricción es irrelevante. Después de quitar la restricción sabemos que α es usada en el entorno solo para tipar la variable `D` y además tenemos la restricción $\beta \Leftarrow \alpha$, por lo que podemos sustituir α con β ya que se cumple que $\Gamma[x : \alpha \mid \alpha \Leftarrow \beta] \approx \Gamma[x : \beta]$ para cada Γ y x cuando α no aparece libre en Γ. Tras la sustitución tenemos el par $\langle \texttt{'plastic'}; [\texttt{D} : \beta, \texttt{M} : \beta \mid \texttt{atom()} \Leftarrow \beta] \rangle$. Podemos entonces aplicar que $\Gamma[x : \beta \mid \tau \Leftarrow \beta] \subseteq \Gamma[x : \tau \mid \tau \Leftarrow \beta]$ para cualquier x y Γ, obteniendo con ello $\langle \texttt{'plastic'}; [\texttt{D} : \texttt{atom()}, \texttt{M} : \texttt{atom()} \mid \texttt{atom()} \Leftarrow \beta] \rangle$. En este momento hemos perdido la conexión entre `D` y `M`, pero no importará ya que `M` va a ser retirada del entorno tras salir del ámbito de la cláusula. Finalmente, podemos quitar la restricción $\texttt{atom()} \Leftarrow \beta$ aplicando el hecho de que $\Gamma[\tau \Leftarrow \beta] \approx \Gamma$ cuando β no aparece libre en Γ.

En el juicio (**) transformamos el entorno antes de dar tipo a la tupla, eliminando así la restricción $\texttt{'true'} \subseteq \texttt{'true'}$ obtenida del resultado de la guarda. Sustituimos α por β gracias a la restricción $\beta \Leftarrow \alpha$ y después de la sustitución renombramos β con una nueva variable de tipo α, que deja de estar libre en el entorno. Como resultado obtenemos el par $\langle \{\texttt{'love'}, \alpha\}; [\texttt{D} : \alpha, \texttt{T} : \alpha] \rangle$, donde podemos ver la variable `D` conectada a la segunda componente del resultado de la función mediante la variable de tipo α.

Sin soporte para tipos funcionales sobrecargados, el tipo más preciso que podríamos obtener para `is_atom` sería $(\texttt{any()}) \to \texttt{bool()}$ y no podríamos obtener ninguna información útil de la guarda en este ejemplo. Para resolver este problema, el sistema de tipos requeriría de una regla específica para tratar la aplicación de `is_atom`. Sin embargo, con los tipos funcionales sobrecargados podemos usar la regla [APP1] para obtener la información de tipos que necesitamos. Para cada tipo funcional que se encuentra dentro del tipo de `is_atom`, obtenemos el tipo anotado formado por los pares $\langle \texttt{'true'}; [\texttt{D} : \alpha, \texttt{M} : \beta \mid \beta \Leftarrow \alpha, \texttt{atom()} \Leftarrow \beta] \rangle$ y $\langle \texttt{'false'}; [\texttt{D} : \alpha, \texttt{M} : \beta \mid \beta \Leftarrow \alpha, \texttt{any()} \Leftarrow \beta] \rangle$. Ambos pares se usarán para encontrar una derivación para el cuerpo de la cláusula (es decir, el átomo `'plastic'` en este ejemplo). En el caso del segundo par, obtenemos un entorno con la restricción $\texttt{'false'} \subseteq \texttt{'true'}$. Se cumple que $\Gamma[\texttt{'false'} \subseteq \texttt{'true'}] \approx \bot$ para cualquier Γ y que $\rho; \bot \approx \rho$ para cualquier ρ, por lo que la segunda rama del tipo anotado será cancelada al aplicar la regla [SUB2].

4.6.2. Función `map`

En el siguiente ejemplo vamos a derivar un tipo para la función `map`, que recibe una función `F` y una lista `L`, y devuelve una nueva lista donde cada elemento es el resultado de aplicar `F` al elemento en la

misma posición en L. El código de map en *Erlang* es el siguiente:

```
map(F, []) -> [];
map(F, [X|XS]) -> [F(X)|map(F,XS)].
```

cuya traducción a *Mini Erlang* es la siguiente:

letrec Map = **fun**(F,L) → **case** L **of**
  [] **when** 'true' → []
  [X|XS] **when** 'true' → [F(X)|Map(F,XS)]
end in Map

El tipo $(\texttt{any()},\texttt{[]}) \to \texttt{[]} \sqcup \forall\alpha,\alpha',\beta,\beta'.((\alpha)\to\beta,\texttt{nelist(}\alpha'\texttt{,[])})\to\texttt{nelist(}\beta'\texttt{,[])}$ **when** $\alpha' \subseteq \alpha, \beta' \subseteq \beta$ es uno de los que podemos obtener para esta función Map con nuestras reglas de tipado. Este tipo representa aquellas funciones que:

- dado cualquier valor y una lista vacía, devuelve una lista vacía, y
- dada una función con un parámetro y una lista no vacía cuyos elementos están dentro del dominio de dicha función, devuelve una lista no vacía cuyos elementos han de estar dentro del rango de la función pasada como parámetro.

Una selección de los juicios obtenidos en la derivación se muestran en la figura 4.7, donde $\tau_{\texttt{Map}}$ es el tipo obtenido para Map (el mismo que hemos descrito arriba), y el entorno inicial de entrada está vacío cuando la regla para la expresión **letrec** es aplicada. Empezamos el ejemplo con la aplicación de la regla [ABS], ya que la aplicación de la regla [LRC] es trivial para este ejemplo.

Usando la herramienta *TypEr* [61] incluida en las distribuciones de *Erlang*, el tipo que obtenemos es $((\texttt{any()})\to\texttt{any()},\texttt{[any()]})\to\texttt{[any()]}$, que consiste en una sobreaproximación del tipo obtenido por nuestra derivación, pero sin tener en cuenta el polimorfismo.

Transformación de las restricciones

El paso más significativo de la derivación, una vez se ha obtenido el tipo anotado con el juicio (**) tras aplicar la regla [LST], es la transformación de dicho tipo aplicando la regla [SUB2] en el juicio (*) de la figura 4.7. En este apartado vamos a discutir sobre la transformación realizada para obtener el tipo anotado para dicho juicio.

Uno de los pasos en la transformación es el tratamiento de la conjunción de restricciones que define a la variable de tipo $\alpha_{\texttt{F}}$. En el primer par de tipos anotados obtenemos $(\alpha)\to\beta \Leftarrow \alpha_{\texttt{F}}$ y $\texttt{any()} \Leftarrow \alpha_{\texttt{F}}$. Toda instanciación de la primera restricción satisface la segunda, por lo que se puede descartar esta última del entorno para obtener uno equivalente. En el segundo par, la conjunción está compuesta por $(\alpha)\to\beta \Leftarrow \alpha_{\texttt{F}}$ y $(\alpha'')\to\beta'' \Leftarrow \alpha_{\texttt{F}}$. Para este caso, ya que tenemos dos tipos funcionales conformados

[ABS]: $[\texttt{Map}:\tau_{\texttt{Map}}] \vdash \textbf{fun}(\texttt{F},\texttt{L}) \to \cdots : \langle \tau_{\texttt{Map}}; [\texttt{Map}:\tau_{\texttt{Map}}] \rangle$

[SUB1]: $[\texttt{Map}:\tau_{\texttt{Map}}] \vdash \textbf{case}\ \texttt{L}\ \textbf{of}\ \cdots : \langle [\,]; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\texttt{any}(), \texttt{L}:[\,]] \rangle;$
$\langle \texttt{nelist}(\beta', [\,])\ \textbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:(\alpha) \to \beta, \texttt{L}:\texttt{nelist}(\alpha', [\,])] \rangle$

[CAS]: $[\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}] \vdash \textbf{case}\ \texttt{L}\ \textbf{of}\ \cdots : \langle [\,]; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\texttt{any}(), \texttt{L}:[\,]] \rangle;$
$\langle \texttt{nelist}(\beta', [\,])\ \textbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:(\alpha) \to \beta, \texttt{L}:\texttt{nelist}(\alpha', [\,])] \rangle$

[CLS]: $[\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}] \Vdash_{\alpha_{\texttt{L}}} [\,]\ \textbf{when}\ \cdots : \langle [\,]; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\texttt{any}(), \texttt{L}:[\,]] \rangle$

[SUB2]: $[\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}} \mid [\,] \Leftarrow \alpha_{\texttt{L}}, \texttt{'true'} \subseteq \texttt{'true'}] \vdash [\,] :$
$\langle [\,]; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\texttt{any}(), \texttt{L}:[\,]] \rangle$

[CLS]: $[\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}] \Vdash_{\alpha_{\texttt{L}}} [\texttt{X}\,|\,\texttt{XS}]\ \textbf{when}\ \cdots :$
$\langle \texttt{nelist}(\beta', [\,])\ \textbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:(\alpha) \to \beta, \texttt{L}:\texttt{nelist}(\alpha', [\,])] \rangle$

(∗) [SUB2]: $[\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha_{\texttt{X}}, \texttt{XS}:\alpha_{\texttt{XS}} \mid \texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}] \vdash [\texttt{F}(\texttt{X})\,|\,\texttt{Map}(\texttt{F},\texttt{XS})] :$
$\langle \texttt{nelist}(\beta', [\,])\ \textbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; \Gamma_4 \rangle; \langle \texttt{nelist}(\beta', [\,])\ \textbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; \Gamma_5 \rangle$
donde $\Gamma_4 = [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:(\alpha) \to \beta, \texttt{L}:\texttt{nelist}(\alpha', [\,]), \texttt{X}:\alpha', \texttt{XS}:[\,]]$
donde $\Gamma_5 = [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:(\alpha) \to \beta, \texttt{L}:\texttt{nelist}(\alpha', [\,]), \texttt{X}:\alpha', \texttt{XS}:\texttt{nelist}(\alpha', [\,])]$

(∗∗) [LST]: $[\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha_{\texttt{X}}, \texttt{XS}:\alpha_{\texttt{XS}} \mid \texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}]$
$\vdash [\texttt{F}(\texttt{X})\,|\,\texttt{Map}(\texttt{F},\texttt{XS})] : \langle \texttt{nelist}(\beta', [\,]); \Gamma_4 \rangle; \langle \texttt{nelist}(\beta', \texttt{nelist}(\beta''', [\,])); \Gamma_5 \rangle$
donde $\Gamma_4 = [\texttt{Map}:\alpha_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha_{\texttt{X}}, \texttt{XS}:\alpha_{\texttt{XS}} \mid \tau_{\texttt{Map}} \Leftarrow \alpha_{\texttt{Map}}, (\alpha) \to \beta \Leftarrow \alpha_{\texttt{F}}, \texttt{any}() \Leftarrow \alpha_{\texttt{F}},$
$\texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}, \alpha' \Leftarrow \alpha_{\texttt{X}}, [\,] \Leftarrow \alpha_{\texttt{XS}}, \alpha' \subseteq \alpha, \beta' \subseteq \beta]$
donde $\Gamma_5 = [\texttt{Map}:\alpha_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha_{\texttt{X}}, \texttt{XS}:\alpha_{\texttt{XS}} \mid \tau_{\texttt{Map}} \Leftarrow \alpha_{\texttt{Map}}, (\alpha) \to \beta \Leftarrow \alpha_{\texttt{F}}, (\alpha'') \to \beta'' \Leftarrow \alpha_{\texttt{F}},$
$\texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}, \alpha' \Leftarrow \alpha_{\texttt{X}}, \texttt{nelist}(\alpha''', [\,]) \Leftarrow \alpha_{\texttt{XS}}, \alpha' \subseteq \alpha, \alpha''' \subseteq \alpha'', \beta' \subseteq \beta, \beta''' \subseteq \beta'']$

[APP1]: $[\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha_{\texttt{X}}, \texttt{XS}:\alpha_{\texttt{XS}} \mid \texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}] \vdash \texttt{F}(\texttt{X}) :$
$\langle \beta'; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:(\alpha) \to \beta, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha', \texttt{XS}:\alpha_{\texttt{XS}} \mid \texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}, \alpha' \Leftarrow \alpha_{\texttt{X}}, \alpha' \subseteq \alpha, \beta' \subseteq \beta] \rangle$

[APP1]: $[\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha_{\texttt{X}}, \texttt{XS}:\alpha_{\texttt{XS}} \mid \texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}] \vdash \texttt{Map}(\texttt{F},\texttt{XS}) :$
$\langle [\,]; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha_{\texttt{X}}, \texttt{XS}:\alpha_{\texttt{XS}} \mid C_2] \rangle;$
$\langle \texttt{nelist}(\beta''', [\,])\ \textbf{when}\ \alpha''' \subseteq \alpha'', \beta''' \subseteq \beta''; [\texttt{Map}:\tau_{\texttt{Map}}, \texttt{F}:\alpha_{\texttt{F}}, \texttt{L}:\alpha_{\texttt{L}}, \texttt{X}:\alpha_{\texttt{X}}, \texttt{XS}:\alpha_{\texttt{XS}} \mid C_3] \rangle$
donde $C_2 = \{\texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}, \texttt{any}() \Leftarrow \alpha_{\texttt{F}}, [\,] \Leftarrow \alpha_{\texttt{XS}}\}$
donde $C_3 = \{\texttt{nelist}(\alpha_{\texttt{X}}, \alpha_{\texttt{XS}}) \Leftarrow \alpha_{\texttt{L}}, (\alpha'') \to \beta'' \Leftarrow \alpha_{\texttt{F}}, \texttt{nelist}(\alpha''', [\,]) \Leftarrow \alpha_{\texttt{XS}}\}$

Figura 4.7: Algunos juicios involucrados en la derivación de `Map`

por variables de tipo, podremos eliminar el tipo $(\alpha'') \to \beta''$ añadiendo las restricciones $\alpha \Leftarrow \alpha''$ y $\beta \Leftarrow \beta''$. Justificamos esto porque siempre que se cumple $(\theta, \nu) \in \mathcal{T}^{\pi}_{Env}[\![\Gamma[(\alpha) \to \beta \Leftarrow \alpha_{\mathtt{F}}, (\alpha') \to \beta' \Leftarrow \alpha_{\mathtt{F}}]]\!]$, entonces $\pi(\alpha) = \pi(\alpha')$ y $\pi(\beta) = \pi(\beta')$. Esta transformación, con algunos matices, se puede también aplicar a otras estructuras, tales como listas o tuplas. En el capítulo 5 presentaremos este tipo de transformaciones con más detalle.

Otro de los pasos consiste en unir dos variables de tipo en un supertipo bajo ciertas condiciones. Por ejemplo, en la segunda rama del tipo anotado obtenido tras aplicar [LST], tenemos como resultado el tipo `nelist(`β'`,nelist(`β'''`,[]))` y sabemos que $\beta' \subseteq \beta$ y $\beta''' \subseteq \beta''$. Recordemos que la transformación anterior introdujo la restricción $\beta \Leftarrow \beta''$, por lo que $\beta''' \subseteq \beta''$ se transforma en $\beta''' \subseteq \beta$, y por lo tanto β' y β''' son subconjunto de β. Entonces introduzcamos la variable $\beta^\star$ definida como $\beta' \cup \beta'''$. Podemos reemplazar $\beta' \subseteq \beta$ y $\beta''' \subseteq \beta$ por $\beta^\star \subseteq \beta$, y además podemos cambiar `nelist(`β'`,nelist(`β'''`,[]))` por `nelist(`$\beta^\star$`,nelist(`$\beta^\star$`,[]))`, que podemos sobreaproximar usando `nelist(`$\beta^\star$`,[])`. Después de finalizar estos cambios y eliminar las restricciones que ya no se usan, podemos renombrar $\beta^\star$ con β', ya que la anterior β' fue eliminada previamente del par. Este paso también se aplica para unir α' y α'''.

Hemos mencionado, en la sección 4.6.1, la sustitución de variables de tipo involucradas con restricciones de encaje y el renombramiento de variables de tipo dentro de los pares de un tipo anotado. Otra transformación trivial consiste en mover restricciones desde un entorno al lado derecho de un tipo **when** y viceversa. Es decir, $\langle \tau; \Gamma[C] \rangle \approx \langle \tau \textbf{ when } C; \Gamma \rangle$, lo cual se demuestra fácilmente a partir de las definiciones semánticas de tipos anotados y entornos.

Usando la función `map`

En este apartado mostraremos algunos casos de uso donde `Map` es aplicada con argumentos válidos e inválidos. El primer ejemplo será `Map(F,L)` bajo un entorno Γ que asigna a `F` el tipo `(number())` $\to$ `number()`, a `L` el tipo `[integer()]` y a `Map` el tipo obtenido antes en esta sección. Mediante la regla [APP1] se obtiene el siguiente juicio:

$$\begin{aligned}
\Gamma \vdash \mathtt{Map(F,L)} : \langle \mathtt{[]}; \Gamma[\mathtt{F}: \alpha_{\mathtt{F}}, \mathtt{L}: \alpha_{\mathtt{L}} \mid C_1] \rangle ; \\
&\langle \mathtt{nelist(}\beta'\mathtt{,[])} \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta; \Gamma[\mathtt{F}: \alpha_{\mathtt{F}}, \mathtt{L}: \alpha_{\mathtt{L}} \mid C_2] \rangle \\
\text{donde } C_1 = \{ &(\mathtt{number()}) \to \mathtt{number()} \Leftarrow \alpha_{\mathtt{F}}, \mathtt{[integer()]} \Leftarrow \alpha_{\mathtt{L}}, \\
&\mathtt{any()} \Leftarrow \alpha_{\mathtt{F}}, \mathtt{[]} \Leftarrow \alpha_{\mathtt{L}} \} \\
\text{donde } C_2 = \{ &(\mathtt{number()}) \to \mathtt{number()} \Leftarrow \alpha_{\mathtt{F}}, \mathtt{[integer()]} \Leftarrow \alpha_{\mathtt{L}}, \\
&(\alpha) \to \beta \Leftarrow \alpha_{\mathtt{F}}, \mathtt{nelist(}\alpha'\mathtt{,[])} \Leftarrow \alpha_{\mathtt{L}} \}
\end{aligned}$$

Se obtienen dos pares. El primer par fuerza a `L` a contener la lista vacía, no impone ninguna restricción a `F`, y fuerza al resultado a ser la lista vacía. El segundo par se puede transformar para obtener que

$\texttt{number()} \Leftarrow \alpha$, $\texttt{number()} \Leftarrow \beta$ y $\texttt{integer()} \Leftarrow \alpha'$. El resultado obtenido en el segundo par es una lista no vacía de tipo `number()`. Aplicando la regla [SUB2], el tipo anotado que obtendríamos es:

$$\Gamma \vdash \texttt{Map(F,L)} : \langle \texttt{[]}; \Gamma[\texttt{F} : (\texttt{number()}) \to \texttt{number()}, \texttt{L} : \texttt{[]}] \rangle ;$$
$$\langle \texttt{nelist(number(),[])}; \Gamma[\texttt{F} : (\texttt{number()}) \to \texttt{number()}, \texttt{L} : \texttt{nelist(integer(),[])}] \rangle$$

El segundo caso de uso que vamos a estudiar es la misma expresión `Map(F,L)`, bajo un Γ definido como el anterior, pero en el que ahora `F` tiene como tipo $(\texttt{bool()}) \to \texttt{bool()}$ y `L` es de tipo `[integer()]`:

$$\begin{aligned}
\Gamma \vdash \texttt{Map(F,L)} : \langle \texttt{[]}; \Gamma[\texttt{F} : \alpha_\texttt{F}, \texttt{L} : \alpha_\texttt{L} \mid C_1] \rangle ; & \\
\left\langle \texttt{nelist}(\beta', \texttt{[]}) \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta; \Gamma[\texttt{F} : \alpha_\texttt{F}, \texttt{L} : \alpha_\texttt{L} \mid C_2] \right\rangle & \\
\text{where } C_1 = \{(\texttt{bool()}) \to \texttt{bool()} \Leftarrow \alpha_\texttt{F}, \texttt{[integer()]} \Leftarrow \alpha_\texttt{L}, & \\
\texttt{any()} \Leftarrow \alpha_\texttt{F}, \texttt{[]} \Leftarrow \alpha_\texttt{L}\} & \\
\text{where } C_2 = \{(\texttt{bool()}) \to \texttt{bool()} \Leftarrow \alpha_\texttt{F}, \texttt{[integer()]} \Leftarrow \alpha_\texttt{L}, & \\
(\alpha) \to \beta \Leftarrow \alpha_\texttt{F}, \texttt{nelist}(\alpha', \texttt{[]}) \Leftarrow \alpha_\texttt{L}\} &
\end{aligned}$$

En esta ocasión, la aplicación de la regla [APP1] también da lugar a dos pares, pero el segundo de ellos es equivalente a $\langle \texttt{none()}; \bot \rangle$. Esto ocurre porque el entorno $\Gamma[\texttt{F} : \alpha_\texttt{F}, \texttt{L} : \alpha_\texttt{L} \mid C_2]$ es equivalente a $[\texttt{F} : \alpha_\texttt{F}, \texttt{L} : \alpha_\texttt{L} \mid C_2 \cup \{\texttt{bool()} \Leftarrow \alpha, \texttt{bool()} \Leftarrow \beta, \texttt{integer()} \Leftarrow \alpha'\}]$, así que toda sustitución θ que pertenece a la semántica de este entorno tiene que ser obtenida con una instanciación π en la que $\pi \vDash \texttt{bool()} \Leftarrow \alpha$ y $\pi \vDash \texttt{integer()} \Leftarrow \alpha'$. Pero no hay ningún valor v tal que (v, π) pertenece a la semántica del tipo $\texttt{nelist}(\beta', \texttt{[]})$ **when** $\alpha' \subseteq \alpha, \beta' \subseteq \beta$, ya que π no puede satisfacer la restricción $\alpha' \subseteq \alpha$. Después de aplicar la regla [SUB2], el tipo anotado que obtenemos es:

$$\Gamma \vdash \texttt{Map(F,L)} : \langle \texttt{[]}; \Gamma[\texttt{F} : (\texttt{bool()}) \to \texttt{bool()}, \texttt{L} : \texttt{[]}] \rangle ;$$

Esto significa que la única posibilidad de éxito para evaluar esta aplicación es que la variable de programa `L` contenga una lista vacía.

Este ejemplo nos muestra una vez más la mayor flexibilidad de los *success types* frente a los tipos *Hindley-Milner*, en donde la función `map(F,L)` sería rechazada como mal tipada si el argumento `F` tiene tipo $(\texttt{bool()}) \to \texttt{bool()}$ y `L` tiene tipo `[integer()]`.

4.6.3. Funciones sobre listas y funciones de orden superior

En esta sección mostraremos tipos obtenidos para algunas funciones que manejan listas, tales como `foldl`, `reverse`, `filter`, y `nth`. Algunas de estas funciones son de orden superior, como sucedía con

`map`. Con la herramienta *TypEr* [61], los tipos que se obtuvieron fueron:

$$\begin{aligned}
\texttt{foldl} &: ((\texttt{any()},\texttt{any()}) \to \texttt{any()},\texttt{any()},[\texttt{any()}]) \to \texttt{any()} \\
\texttt{filter} &: ((\texttt{any()}) \to \texttt{any()},[\texttt{any()}]) \to [\texttt{any()}] \\
\texttt{reverse} &: ([\texttt{any()}]) \to [\texttt{any()}] \\
\texttt{nth} &: (\texttt{pos_integer()},\texttt{nonempty_maybe_improper_list()}) \to \texttt{any()}
\end{aligned}$$

Mientras que con nuestro sistema de tipos pudimos derivar los siguientes tipos en su lugar:

$$\begin{aligned}
\texttt{foldl} &: \forall\beta.\big(\texttt{any()},\beta,[]\big) \to \beta \sqcup \\
&\quad \forall\alpha,\alpha',\beta,\beta',\gamma,\gamma'.\big((\alpha,\beta) \to \gamma,\beta',\texttt{nelist}(\alpha',[])\big) \to \gamma' \textbf{ when } \alpha' \subseteq \alpha,\beta' \subseteq \beta,\gamma' \subseteq \gamma \\
\texttt{filter} &: (\texttt{any()},[]) \to [] \sqcup \\
&\quad \forall\alpha,\alpha',\beta,\beta'.\big((\alpha) \to \alpha',\texttt{nelist}(\beta,[])\big) \to \texttt{nelist}(\beta',[]) \textbf{ when } \beta \subseteq \alpha,\beta' \subseteq \beta,\texttt{bool()} \subseteq \alpha' \\
\texttt{reverse} &: \forall\alpha.([\alpha]) \to [\alpha] \\
\texttt{nth} &: \forall\alpha,\beta.(\texttt{number()},\texttt{nelist}(\alpha,\texttt{any()})) \to \beta \textbf{ when } \beta \subseteq \alpha
\end{aligned}$$

Los tipos dados por *TypEr* para `foldl`, `filter` y `reverse`, son supertipos de los que nosotros hemos derivado con nuestro sistema de tipos. Pero en el caso del tipo para `nth`, podemos observar que *TypEr* ha encontrado un tipo más preciso para el primer argumento de la función. Para mejorar nuestras derivaciones con operadores aritméticos, necesitaríamos tipos sobrecargados para estos operadores, pues con ello podríamos distinguir entre casos según los tipos de sus argumentos.

En los siguientes apartados vamos a explicar cómo se obtuvieron los tipos mostrados anteriormente.

Función `foldl`

La función `foldl` toma como parámetros una función, un valor acumulador y una lista. Esta función se utiliza para reducir una lista a un único valor acumulado. Para lograrlo se aplica la función recibida a cada elemento de la lista—de izquierda a derecha—para obtener el siguiente valor acumulado que será utilizado en la próxima iteración de la función. El código de la función en *Erlang* es el siguiente:

```
foldl(F, A, []) -> A;
foldl(F, A, [X|XS]) -> foldl(F, F(X,A), XS).
```

El tipo que hemos obtenido para `foldl` con una derivación usando nuestro sistema de tipos es:

$$\begin{aligned}
&\forall\beta.\big(\texttt{any()},\beta,[]\big) \to \beta \sqcup \\
&\forall\alpha,\alpha',\beta,\beta',\gamma,\gamma'.\big((\alpha,\beta) \to \gamma,\beta',\texttt{nelist}(\alpha',[])\big) \to \gamma' \textbf{ when } \alpha' \subseteq \alpha,\beta' \subseteq \beta,\gamma' \subseteq \gamma
\end{aligned}$$

En la primera rama del tipo dado, el primer parámetro `F` puede ser cualquier valor, el segundo parámetro `A` es también el tipo del resultado, y el tercer parámetro—al que llamaremos `L`—es una lista vacía. En la segunda rama, el primer parámetro `F` es la función que mezcla el valor acumulado recibido—en el segundo parámetro `A`—con la cabeza de la lista en el tercer parámetro `L`. Por esta razón el tipo de `A` está contenido en el tipo del segundo parámetro de la función de `F` y el tipo de `L` está contenido en el tipo del primer parámetro de la función de `F`. Por último, el resultado de tipo funcional en esta rama es el tipo del resultado de la función de `F`.

Al contrario de lo que podría parecer a simple vista, el tipo $\forall\alpha,\alpha',\beta,\beta',\gamma,\gamma'.((\alpha,\beta)\rightarrow\gamma,\beta',[\alpha'])\rightarrow\gamma'$ **when** $\alpha'\subseteq\alpha,\beta'\subseteq\beta,\gamma'\subseteq\gamma$ no es un *success type* para esta función, porque cuando `L` es una lista vacía, el parámetro `A` no necesita estar relacionado con el resultado del tipo de `F`, ya que la función acumuladora no va a ser aplicada. Por esta razón, cuando `L` es una lista vacía, el resultado obtenido con la regla [CLS] es el par $\langle\beta;[\texttt{F}:\texttt{any()},\texttt{A}:\beta,\texttt{L}:[]]\rangle$, mientras que en la cláusula que maneja las listas no vacías, obtenemos:

$$\langle\gamma'\ \textbf{when}\ \alpha'\subseteq\alpha,\beta'\subseteq\beta,\gamma'\subseteq\gamma;[\texttt{F}:(\alpha,\beta)\rightarrow\gamma,\texttt{A}:\beta',\texttt{L}:\texttt{nelist(}\alpha'\texttt{,[])}]\rangle$$

Función `filter`

La función `filter` recibe una función predicado y una lista, para devolver una nueva lista sin aquellos elementos para los que el predicado devuelve `'false'`. El código de la función en *Erlang* es el siguiente:

```
filter(P, []) -> [];
filter(P, [X|XS]) ->
    case P(X) of
        'true' -> [X|filter(P, XS)];
        'false' -> filter(P, XS)
    end.
```

El tipo que hemos derivado para `filter` es:

$$\begin{array}{l}(\texttt{any()},[])\rightarrow[]\sqcup\\ \forall\alpha,\alpha',\beta,\beta'.((\alpha)\rightarrow\alpha',\texttt{nelist(}\beta\texttt{,[])})\rightarrow\texttt{nelist(}\beta'\texttt{,[])}\ \textbf{when}\ \beta\subseteq\alpha,\beta'\subseteq\beta,\texttt{bool()}\subseteq\alpha'\end{array}$$

En la primera rama del tipo, el primer parámetro `P` puede ser cualquier valor (no necesariamente funcional), el segundo parámetro `L` es una lista vacía y el tipo del resultado es la lista vacía. En el tipo funcional de la segunda rama, el primer parámetro `P` es una función predicado, el segundo parámetro `L` es una lista no vacía cuyos elementos están contenidos por el tipo del único parámetro de la función en `P`, y el tipo del resultado es un subtipo de la lista no vacía `L`.

De modo similar al ejemplo previo, el tipo $\forall\alpha.\forall\beta.((\alpha)\rightarrow\texttt{bool()},[\beta])\rightarrow[\beta']$ **when** $\beta\subseteq\alpha,\beta'\subseteq\beta$ no es un *success type* para esta función, porque cuando `L` es una lista vacía, `P` no se utiliza, por lo que

no es necesario que sea una función. Cuando L no es una lista vacía, sabemos que el tipo de P es $(\alpha) \to \alpha'$ y el resultado de `P(X)` toma como tipo la variable de tipo α'' que termina con las siguientes restricciones: $\alpha'' \subseteq \alpha'$ y `'true'` $\cup$ `'false'` $\Leftarrow \alpha''$. Como la definición de `bool()` es `'true'` $\cup$ `'false'`, obtenemos transformando las restricciones que `bool()` $\subseteq \alpha'$. El hecho de que la función tenga que contener valores booleanos, no quiere decir que no pueda ofrecer más valores, siempre que contenga al menos el conjunto de los booleanos. Por ello limitar P al tipo $(\alpha) \to$ `bool()` daría una semántica menor que la que tiene la expresión.

Función `reverse`

La función `reverse` recibe una lista y devuelve una nueva lista con los mismos elementos en orden inverso. El código de la función en *Erlang* es el siguiente:

```
reverse(L) -> reverse(L, []).

revtail([], A) -> A;
revtail([X|XS], A) -> reverse(XS, [X|A]).
```

El tipo $\forall\alpha.([\alpha]) \to [\alpha]$ es el que hemos obtenido para `reverse` de la derivación con nuestro sistema de tipos, donde L—el único parámetro—es una lista. La inversión de la lista es realizada con la función auxiliar `revtail`, que tiene dos parámetros. El primer parámetro, al que llamaremos L, es una lista y el segundo parámetro A es un acumulador. Mediante las reglas de derivación puede obtenerse el siguiente tipo para `revtail`:

$$\forall\beta.([],\beta) \to \beta \sqcup$$
$$\forall\alpha,\beta.(\texttt{nelist}(\alpha,[]),\beta) \to \texttt{nelist}(\alpha,\beta)$$

A partir de esto, la derivación de la expresión `revtail`(L, `[]`) genera el siguiente tipo anotado como resultado:

$$\langle\beta;\Gamma[\mathtt{L}:\alpha_\mathtt{L},\mathtt{K}:\alpha_\mathtt{K} \mid [] \Leftarrow \alpha_\mathtt{K}, [] \Leftarrow \alpha_\mathtt{L}, \beta \Leftarrow \alpha_\mathtt{K}]\rangle;$$
$$\langle\texttt{nelist}(\alpha,\beta);\Gamma[\mathtt{L}:\alpha_\mathtt{L},\mathtt{K}:\alpha_\mathtt{K} \mid [] \Leftarrow \alpha_\mathtt{K}, \texttt{nelist}(\alpha,[]) \Leftarrow \alpha_\mathtt{L}, \beta \Leftarrow \alpha_\mathtt{K}]\rangle$$

donde Γ contiene los tipos de `reverse` y `revtail`. La variable K es una variable auxiliar que almacena la lista vacía que pasamos como segundo argumento en la llamada a `revtail`, ya que en *Mini Erlang* solo se aceptan variables como argumentos en la aplicación. Entonces podemos aplicar la regla [SUB2] por la siguiente razón:

$$\begin{array}{ll} & \langle[];\Gamma[\mathtt{L}:[],\mathtt{K}:[]]\rangle;\langle\texttt{nelist}(\alpha,[]);\Gamma[\mathtt{L}:\texttt{nelist}(\alpha,[]),\mathtt{K}:[]]\rangle \\ \subseteq & \langle[\alpha];\Gamma[\mathtt{L}:[\alpha],\mathtt{K}:[]]\rangle; \end{array}$$

Por lo tanto, después de simplificar el tipo anotado en un solo par, el parámetro `L` y el resultado de `reverse` es `[`α`]`. La aparición de la misma α—tanto en la entrada como en la salida—implica que el conjunto de elementos de la lista de salida es el mismo que el conjunto de elementos de la lista de entrada.

Función `nth`

La función `nth` recibe un número y una lista, para devolver el elemento situado en la posición especificada por el número dentro de dicha lista. Su código en *Erlang* es el siguiente:

```
nth(1, [X|_]) -> X;
nth(N, [_|XS]) when N > 1 -> nth(N - 1, XS).
```

El tipo $\forall\alpha,\beta.$(`number()`,`nelist(`α`,any())`) $\rightarrow \beta$ **when** $\beta \subseteq \alpha$ es el que hemos obtenido para la función `nth`. Como no es necesario recorrer la lista entera para devolver un resultado, no podemos asegurar que la continuación de la lista sea `[]`. Por ello obtenemos el tipo `nelist(`α`,any())` para la lista de entrada.

4.7. Resultados de corrección

En esta sección demostraremos que los tipos derivados con el conjunto de reglas mostrados en la sección 4.5 son *success types* para sus correspondientes expresiones. Para ello, antes de entrar en el teorema de corrección, necesitaremos algunos resultados auxiliares.

4.7.1. Lemas para los renombramientos

Antes de mostrar los lemas, necesitamos definir cómo se puede renombrar una instanciación. Dado un renombramiento μ y una instanciación π, la notación $\pi\mu$ denota una instanciación tal que $(\pi\mu)(\alpha) = \pi(\mu^{-1}(\alpha))$ para cada $\alpha \in rng\,\mu$ y $(\pi\mu)(\alpha) = \pi(\alpha)$ para cada $\alpha \notin rng\,\mu$. Por ejemplo, si partimos de la instanciación $\pi = [\alpha \mapsto \{3\}, \beta \mapsto \{1\}]$ y el renombramiento $\mu = [\alpha/\gamma]$, entonces obtenemos que $\pi\mu = [\alpha \mapsto \{3\}, \beta \mapsto \{1\}, \gamma \mapsto \{3\}]$.

Para poder aplicar un renombramiento a las variables de tipo que aparecen libres en un tipo, este renombramiento ha de aplicarse también a la operación de descomposición de las instanciaciones en la semántica de los tipos. En el siguiente resultado veremos cómo se pueden renombrar las variables dentro de una descomposición de instancias:

Lema 2. *Suponemos una instanciación π, un tipo τ y un conjunto de instanciaciones Π tales que $\pi \in Dcp(\Pi,\tau)$. Suponemos también un renombramiento μ tal que dom $\mu \backslash rng\,\mu = \{\alpha_1,\ldots,\alpha_n\}$. Entonces:*

$$\pi\mu \in Dcp\left(\left\{(\pi'\mu)\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right] \mid \pi' \in \Pi\right\}, \tau\mu\right)$$

Demostración. Vamos a denotar con Π° el conjunto $\left\{(\pi'\mu)\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right] \mid \pi' \in \Pi\right\}$. Tenemos que demostrar que:

$$\pi\mu = \bigcup \Pi^\circ \tag{4.3}$$

$$\pi\mu \equiv \pi^\circ \text{ (módulo } \mathbf{TypeVar} \backslash itv(\tau\mu)) \text{ for each } \pi^\circ \in \Pi^\circ \tag{4.4}$$

Para poder demostrar (4.3) vamos a suponer una variable α. Si $\alpha \in rng\ \mu$ entonces:

$$\begin{array}{ll} & (\pi\mu)(\alpha) \\ = & \pi(\mu^{-1}(\alpha)) \\ = & \bigcup\{\pi'(\mu^{-1}(\alpha)) \mid \pi' \in \Pi\} \\ = & \bigcup\{(\pi'\mu)(\alpha) \mid \pi' \in \Pi\} \\ = & \bigcup\left\{(\pi'\mu)\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right](\alpha) \mid \pi' \in \Pi\right\} \\ = & \bigcup\{\pi^\circ(\alpha) \mid \pi^\circ \in \Pi^\circ\} \end{array}$$

Si $\alpha \notin rng\ \mu$ pero $\alpha \in dom\ \mu$ entonces:

$$\begin{array}{ll} & (\pi\mu)(\alpha) \\ = & \pi(\alpha) \\ = & \bigcup\{\pi^\circ(\alpha) \mid \pi^\circ \in \Pi^\circ\} \end{array}$$

El último paso se justifica con que $\alpha \in dom\ \mu \backslash rng\ \mu$ y por ello $\pi^\circ(\alpha) = \pi(\alpha)$ para cada $\pi^\circ \in \Pi^\circ$. Finalmente, si $\alpha \notin rng\ \mu$ y $\alpha \notin dom\ \mu$ obtenemos que:

$$\begin{array}{ll} & (\pi\mu)(\alpha) \\ = & \pi(\alpha) \\ = & \bigcup\{\pi'(\alpha) \mid \pi' \in \Pi\} \\ = & \bigcup\{(\pi'\mu)(\alpha) \mid \pi' \in \Pi\} \\ = & \bigcup\left\{(\pi'\mu)\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right](\alpha) \mid \pi' \in \Pi\right\} \\ = & \bigcup\{\pi^\circ(\alpha) \mid \pi^\circ \in \Pi^\circ\} \end{array}$$

Ahora vamos a demostrar (4.4). Suponemos un $\pi^\circ \in \Pi^\circ$. Sabemos que $\pi^\circ = (\pi'\mu)[\overline{\alpha_i \mapsto \pi(\alpha_i)}]$ para un $\pi' \in \Pi$. Suponiendo una variable $\alpha \notin itv(\tau\mu)$ tenemos que demostrar que $(\pi\mu)(\alpha) = \pi^\circ(\alpha)$. Si $\alpha \in rng\ \mu$ obtenemos que $\mu^{-1}(\alpha) \notin itv(\tau)$ y por ello $\pi(\mu^{-1}(\alpha)) = \pi'(\mu^{-1}(\alpha))$. Por lo tanto:

$$(\pi\mu)(\alpha) = \pi(\mu^{-1}(\alpha)) = \pi'(\mu^{-1}(\alpha)) = (\pi'\mu)(\alpha) = \pi^\circ(\alpha)$$

Al contrario, si $\alpha \notin rng\ \mu$ pero $\alpha \in dom\ \mu$ tenemos que

$$(\pi\mu)(\alpha) = \pi(\alpha) = \pi^\circ(\alpha)$$

ya que $\alpha \in dom\ \mu \backslash rng\ \mu$. Finalmente, en el caso en el que α no pertenece a $dom\ \mu$ ni a $rng\ \mu$ tenemos

que $\alpha \notin itv(\tau)$ y por lo tanto:

$$(\pi\mu)(\alpha) = \pi(\alpha) = \pi'(\alpha) = (\pi'\mu)(\alpha) = \pi^\circ(\alpha)$$

□

A continuación, dado un conjunto X de variables de tipo y un renombramiento μ, denotamos con $X\mu$ el resultado de aplicar el renombramiento a todas las variables que aparecen dentro de X. Siendo más concretos:

$$X\mu = \{\mu(\alpha) \mid \alpha \in X, \alpha \in dom\ \mu\} \cup \{\alpha \mid \alpha \in X, \alpha \notin dom\ \mu\}$$

El siguiente lema determina algunas propiedades de esta definición:

Lema 3. *Suponemos el renombramiento μ.*

1. *Para cada tipo τ, $itv(\tau)\,\mu = itv(\tau\mu)$ y $ftv(\tau)\,\mu = ftv(\tau\mu)$.*
2. *Para cada $X, Y \subseteq$ **TypeVar** tales que $rng\ \mu \cap (X \cup Y) = \emptyset$ se cumple que $(X \setminus Y)\mu = X\mu \setminus Y\mu$.*
3. *Para cada $X, Y \subseteq$ **TypeVar** se cumple que $(X \cup Y)\mu = X\mu \cup Y\mu$.*

Demostración. Vamos a demostrar la primera propiedad. Supongamos una variable de tipo $\alpha \in itv(\tau)\,\mu$. Con la definición de este conjunto podemos distinguir entre dos casos:

- Si $\alpha = \mu(\beta)$ tal que $\beta \in itv(\tau)$, entonces $\alpha \in itv(\tau\mu)$.
- Si $\alpha \in itv(\tau)$ con $\alpha \notin dom\ \mu$, entonces $\alpha \in itv(\tau\mu)$.

Ahora supongamos que $\alpha \in itv(\tau\mu)$. Tenemos dos posibilidades:

- α existe en $itv(\tau\mu)$ como consecuencia de alguna β existente en $itv(\tau)$ que ha sido renombrada como α por μ. En este caso tenemos que $\alpha = \mu(\beta) \in itv(\tau)\,\mu$.
- α existe en $itv(\tau\mu)$ porque ya aparecía en τ y no ha sido renombrada por μ. Esto significa que $\alpha \notin dom\ \mu$ y por lo tanto $\alpha \in itv(\tau)\,\mu$.

Podríamos repetir el mismo razonamiento de arriba con $ftv(\tau)$ en lugar de $itv(\tau)$. Ahora vamos a demostrar la segunda propiedad. Supongamos que $\alpha \in (X \setminus Y)\mu$. Acorde con la última definición, hay dos posibilidades:

- $\alpha = \mu(\beta)$ para una $\beta \in X \setminus Y$. Como $\beta \in X$ tenemos que $\alpha \in X\mu$. Así que vamos a demostrar que $\alpha \notin Y\mu$ por reducción al absurdo. Supongamos que $\alpha \in Y\mu$.
  - Si $\alpha = \mu(\beta')$ para una $\beta' \in Y$, entonces tendríamos que $\beta' = \beta$ porque μ es inyectiva, y por lo tanto $\beta \in Y$ que contradice que $\beta \in X \setminus Y$.

- Si $\alpha \in Y$ y $\alpha \notin dom\ \mu$, entonces estamos contradiciendo que $Y \cap rng\ \mu = \emptyset$.

Por lo tanto, $\alpha \notin Y\mu$, así que $\alpha \in X\mu \setminus Y\mu$.

- $\alpha \in X \setminus Y$ y $\alpha \notin dom\ \mu$. Como $\alpha \in X$ tenemos en este caso que $\alpha \in X\mu$. Ahora vamos a demostrar, de nuevo por reducción al absurdo, que $\alpha \notin Y\mu$. Supongamos que $\alpha \in Y\mu$.

  - Si $\alpha = \mu(\beta)$ para una $\beta \in Y$ entonces estaríamos contradiciendo que $X \cap rng\ \mu = \emptyset$.
  - Si $\alpha \in Y$ y $\alpha \notin dom\ \mu$, lo primero contradice que $\alpha \in X \setminus Y$.

  Por lo tanto, $\alpha \notin Y\mu$ y como consecuencia $\alpha \in X\mu \setminus Y\mu$.

Hemos demostrado así que $(X \setminus Y)\mu \subseteq X\mu \setminus Y\mu$. Ahora demostraremos la inclusión opuesta. Supongamos $\alpha \in X\mu \setminus Y\mu$. El hecho de que $\alpha \notin Y\mu$ implica lo siguiente:

$$\alpha \notin \{\mu(\beta) \mid \beta \in Y, \beta \in dom\ \mu\} \tag{4.5}$$

$$\alpha \notin \{\alpha \mid \alpha \in Y, \alpha \notin dom\ \mu\} \tag{4.6}$$

Con el hecho de que $\alpha \in X\mu$ podemos distinguir dos casos:

- Si $\alpha = \mu(\beta)$ para una $\beta \in X$ y $\beta \in dom\ \mu$, esto implica que $\beta \notin Y$, ya que de lo contrario estaríamos contradiciendo a (4.5). Por lo tanto, $\beta \in X \setminus Y$ y $\alpha \in (X \setminus Y)\mu$.
- Si $\alpha \in X$ y $\alpha \notin dom\ \mu$ entonces necesitamos que $\alpha \notin Y$ para no contradecir a (4.6). Por lo tanto, $\alpha \in X \setminus Y$ y $\alpha \in (X \setminus Y)\mu$.

Por consiguiente, hemos demostrado $(X \setminus Y)\mu \supseteq X\mu \setminus Y\mu$ y con ello la igualdad entre ambos conjuntos.

Finalmente, vamos a demostrar la tercera propiedad. Primero demostraremos que $(X \cup Y)\mu \subseteq X\mu \cup Y\mu$. Suponemos que $\alpha \in (X \cup Y)\mu$. Distinguiremos los siguientes casos:

- Suponemos que $\alpha = \mu(\beta)$ para una $\beta \in X \cup Y$. Si $\beta \in X$ obtenemos que $\alpha \in X\mu \subseteq X\mu \cup Y\mu$. De manera similar que el caso $\beta \in Y$.
- Suponemos que $\alpha \in X \cup Y$ y $\alpha \notin dom\ \mu$. Si $\alpha \in X$ entonces $\alpha \in X\mu$. Si $\alpha \in Y$ entonces $\alpha \in Y\mu$. Por lo tanto, $\alpha \in X\mu \cup Y\mu$.

Ahora vamos a demostrar que $X\mu \cup Y\mu \subseteq (X \cup Y)\mu$. Suponemos que $\alpha \in X\mu \cup Y\mu$. Solo demostraremos el caso en el que $\alpha \in X\mu$, ya que el caso $\alpha \in Y\mu$ es similar. Tenemos dos posibilidades:

- Si $\alpha = \mu(\beta)$ para una $\beta \in X$, entonces $\beta \in X \cup Y$ y por ello $\alpha \in (X \cup Y)\mu$.
- Si $\alpha \in X$ y $\alpha \notin dom\ \mu$, entonces $\alpha \in X \cup Y$ y por ello $\alpha \in (X \cup Y)\mu$.

Por lo tanto, $X\mu \cup Y\mu \subseteq (X \cup Y)\mu$ y con ello hemos demostrado la igualdad. □

A continuación tenemos los resultados, con dos lemas, que nos indican que es posible aplicar un renombramiento a instanciaciones bajo ciertas condiciones:

Lema 4. *Supongamos dos instanciaciones π y π', un renombramiento μ, y un conjunto X de variables de tipo, tales que $\pi \equiv \pi'$ (módulo* **TypeVar**$\setminus X$*). Denotemos con $\{\overline{\alpha_i}\}$ el conjunto de variables de tipo en dom $\mu \setminus$ rng μ. Entonces:*

$$\pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \equiv \pi'\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \text{ (módulo } \mathbf{TypeVar} \setminus X\mu)$$

Demostración. Por motivos de concisión, las instanciaciones $\pi\mu\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right]$ y $\pi'\mu\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right]$ las vamos a denotar con π_0 y π_1 respectivamente. Supongamos una variable $\alpha \notin X\mu$. Demostraremos que $\pi_0(\alpha) = \pi_1(\alpha)$. Si $\alpha \in rng\ \mu$ obtenemos que $\mu^{-1}(\alpha) \notin X$ y por consiguiente:

$$\pi_0(\alpha) = \pi(\mu^{-1}(\alpha)) = \pi'(\mu^{-1}(\alpha)) = \pi_1(\alpha)$$

Si $\alpha \notin rng\ \mu$ pero $\alpha \in dom\ \mu$ obtenemos que $\alpha \in dom\ \mu \setminus rng\ \mu$. Por lo tanto:

$$\pi_0(\alpha) = \emptyset = \pi_1(\alpha)$$

Finalmente, si $\alpha \notin rng\ \mu$ y $\alpha \notin dom\ \mu$ obtenemos que $\alpha \notin X$ y con ello:

$$\pi_0(\alpha) = \pi(\alpha) = \pi'(\alpha) = \pi_1(\alpha)$$

□

Lema 5. *Supongamos un renombramiento μ y un conjunto de variables X tales que $X \cap rng\ \mu = \emptyset$. Entonces, para cada instanciación π, si $\pi \equiv [\,]$ (módulo X), entonces $\pi\mu \equiv [\,]$ (módulo $X\mu$).*

Demostración. Suponemos la variable $\alpha \in X\mu$. Tenemos que demostrar que $(\pi\mu)(\alpha) = \emptyset$. Teniendo en cuenta la definición de $X\mu$ tenemos dos posibilidades:

- $\alpha = \mu(\beta)$ para una $\beta \in X$. Entonces, de la suposición, $\pi(\beta) = \emptyset$. Así obtenemos:

$$(\pi\mu)(\alpha) = \pi(\mu^{-1}(\alpha)) = \pi(\beta) = \emptyset$$

- $\alpha \in X$ con $\alpha \notin dom\ \mu$. Ya que X y $rng\ \mu$ son disjuntos obtenemos que $\alpha \notin rng\ \mu$, lo que implica que $(\pi\mu)(\alpha) = \pi(\alpha)$. De la suposición conseguimos que $\pi(\alpha) = \emptyset$.

□

El siguiente resultado establece que si cambiamos el nombre de las variables de tipo que aparecen libres en un tipo, el tipo resultante denota los mismos conjuntos de valores, pero con sus instancias modificadas en consecuencia.

Lema 6. *Sea μ una sustitución de variables de tipo a variables de tipo. Para cada τ, C, v y π tales que $rng\ \mu \cap ftv(\tau) = \emptyset$ se cumple que:*

1. $(v,\pi) \in \mathcal{T}\ [\![\tau]\!] \Longrightarrow (v,\pi\mu) \in \mathcal{T}\ [\![\tau\mu]\!]$
2. $\pi \vDash C \Longrightarrow \pi\mu \vDash C\mu$

Demostración. Por inducción sobre la estructura de τ y C. Los casos en los que τ son tipos base simples (unitarios, `none()`, `any()`, `integer()`, etcétera) son triviales. Sin pérdida de generalidad, supongamos que $dom\ \mu \subseteq ftv(\tau)$. De lo contrario restringiríamos el dominio de μ a $ftv(\tau)$ para obtener μ' y demostrar la propiedad sobre μ' en vez de μ. De esta forma, obtendríamos $(v,\pi\mu) \in \mathcal{T}\ [\![\tau\mu]\!] \Leftrightarrow (v,\pi\mu') \in \mathcal{T}\ [\![\tau\mu']\!]$ aplicando el lema 1 y tomando en cuenta que $\mathcal{T}\ [\![\tau\mu']\!] = \mathcal{T}\ [\![\tau\mu]\!]$. Ahora consideremos el resto de casos.

- **Caso** $\tau = \alpha$. Si $(v,\pi) \in \mathcal{T}\ [\![\alpha]\!]$ entonces $\pi(\alpha) = \{v\}$. Ya que $\alpha \notin rng\ \mu$ sabemos que $(\pi\mu)(\alpha) = \{v\}$, así que si $\alpha \notin dom\ \mu$ tenemos que $(v,\pi\mu) \in \mathcal{T}\ [\![\alpha]\!] = \mathcal{T}\ [\![\alpha\mu]\!]$. Si $\alpha \in dom\ \mu$ supongamos que $\mu(\alpha) = \beta$. Por ello $(\pi\mu)(\beta) = \pi(\alpha) = \{v\}$ y por lo tanto $(v,\pi\mu) \in \mathcal{T}\ [\![\beta]\!] = \mathcal{T}\ [\![\alpha\mu]\!]$.
- **Caso** $\tau = \{\overline{\tau_i}^n\}$. Suponemos que $(v,\pi) \in \mathcal{T}\ [\![\{\overline{\tau_i}^n\}]\!]$, entonces $v = \{\overline{v_i}^n\}$ para algún $\overline{v_i}^n$ tal que $(v_i,\pi) \in \mathcal{T}\ [\![\tau_i]\!]$ para cada $i \in \{1..n\}$. Por hipótesis de inducción obtenemos que $(v_i,\pi\mu) \in \mathcal{T}\ [\![\tau_i\mu]\!]$ para cada $i \in \{1..n\}$ y por lo tanto $(v,\pi\mu) \in \mathcal{T}\ [\![\tau\mu]\!]$.
- **Caso** $\tau = \texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}$. En este caso tenemos que $v = [v_1,\ldots,v_n \mid v']$ tal que $(v_i,\pi_i) \in \mathcal{T}\ [\![\tau_1]\!]$ para cada $i \in \{1..n\}$ tal que $\pi \in Dcp\left(\{\overline{\pi_i}^n\},\tau_1\right)$ y $(v',\pi) \in \mathcal{T}\ [\![\tau_2]\!]$. Por h.i. tenemos que $(v_i,\pi_i\mu) \in \mathcal{T}\ [\![\tau_1\mu]\!]$ para cada $i \in \{1..n\}$ y $(v',\pi\mu) \in \mathcal{T}\ [\![\tau_2\mu]\!]$. Ahora supongamos que $dom\ \mu \backslash rng\ \mu = \{\overline{\alpha_i}\}$. Entonces ninguna de las α_i aparecerían libres en $\tau_1\mu'$. Por ello podemos usar el lema 1 para asegurar que $\left(v_i,(\pi_i\mu)\left[\overline{\alpha_i \mapsto \pi_i(\alpha_i)}\right]\right) \in \mathcal{T}\ [\![\tau_1\mu]\!]$ para cada $i \in \{1..n\}$. Ya que sabemos que $\pi \in Dcp\left(\{\overline{\pi_i}^n\},\tau_1\right)$. Podemos aplicar el lema 2 para obtener:

  $$\pi\mu \in Dcp\left(\left\{(\pi_i\mu)\left[\overline{\alpha_i \mapsto \pi_i(\alpha_i)}\right] \mid i \in \{1..n\}\right\},\tau_1\mu\right)$$

  Y por lo tanto $(v,\pi\mu) \in \mathcal{T}\ [\![\texttt{nelist(}\tau_1\mu\texttt{,}\tau_2\mu\texttt{)}]\!] = \mathcal{T}\ [\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}\mu]\!]$.
- **Caso** $\tau = \tau_1 \cup \tau_2$. Suponemos que $(v,\pi) \in \mathcal{T}\ [\![\tau_1]\!] \setminus (itv(\tau_2) \setminus ftv(\tau_1))$, ya que demostrar el caso $(v,\pi) \in \mathcal{T}\ [\![\tau_2]\!] \setminus (itv(\tau_1) \setminus ftv(\tau_2))$ es similar. Entonces existe un π' tal que $(v,\pi') \in \mathcal{T}\ [\![\tau_1]\!]$ y

  $$\pi \equiv \pi' \text{ (módulo } \textbf{TypeVar} \setminus (itv(\tau_2) \setminus ftv(\tau_1)))$$
  $$\pi \equiv [\,] \text{ (módulo } (itv(\tau_2) \setminus ftv(\tau_1)))$$

Suponemos que $dom\,\mu \backslash rng\,\mu = \{\overline{\alpha_i}\}$ para unas $\overline{\alpha_i}$. Podemos aplicar el lema 4 a la primera congruencia y, ya que $rng\,\mu$ y $itv(\tau_2)$ son disjuntos, podemos aplicar el lema 5 a la segunda congruencia. Así tenemos:

$$\pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \equiv \pi'\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \text{ (módulo } \mathbf{TypeVar} \backslash (itv(\tau_2) \backslash ftv(\tau_1))\mu)$$
$$\pi\mu \equiv [\,] \text{ (módulo } (itv(\tau_2) \backslash ftv(\tau_1))\mu)$$

Así podemos transformar estas congruencias usando el lema 3 y el hecho de que si $\pi \equiv [\,]$ módulo un conjunto, tenemos que $\pi\left[\overline{\alpha_i \mapsto \emptyset}\right] \equiv [\,]$ módulo el mismo conjunto:

$$\pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \equiv \pi'\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \text{ (módulo } \mathbf{TypeVar} \backslash (itv(\tau_2\mu) \backslash ftv(\tau_1\mu)))$$
$$\pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \equiv [\,] \text{ (módulo } (itv(\tau_2\mu) \backslash ftv(\tau_1\mu)))$$

Ahora sabemos que $(v,\pi') \in \mathscr{T}[\![\tau_1]\!]$. Por h.i. tenemos $(v,\pi'\mu) \in \mathscr{T}[\![\tau_1\mu]\!]$. Como las variables en $dom\,\mu \backslash rng\,\mu$ no están libres en $\tau_1\mu$, podemos usar el lema 1 para obtener $\left(v,\pi'\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]\right) \in \mathscr{T}[\![\tau_1\mu]\!]$. Esto último, con las ecuaciones mostradas arriba, nos conduce a $\left(v,\pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]\right) \in \mathscr{T}[\![\tau_1\mu]\!] \backslash (itv(\tau_2\mu) \backslash ftv(\tau_1\mu))$, así que al final $\left(v,\pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]\right) \in \mathscr{T}[\![\tau_1\mu \cup \tau_2\mu]\!] = \mathscr{T}[\![(\tau_1 \cup \tau_2)\mu]\!]$. Como las variables $\overline{\alpha_i}$ no están libres en $(\tau_1 \cup \tau_2)\mu$, podemos usar el lema 1 para obtener $(v,\pi\mu) \in \mathscr{T}[\![(\tau_1 \cup \tau_2)\mu]\!]$.

- **Caso** $\tau = \tau_1$ **when** C. Se demuestra directamente al aplicar la hipótesis de inducción sobre sus constituyentes.

- **Caso** $\tau = \bigsqcup_{i=1}^{l} \sigma_i$. Primero vamos a demostrar el siguiente hecho para cada grafo v de aridad n, toda instanciación π, todo renombramiento μ y todos los $\overline{\tau_i}^n, \tau'$ tales que $rng\,\mu \cap ftv\left(\left(\overline{\tau_i}^n\right) \to \tau\right) = \emptyset$:

$$(v,\pi) \in \mathscr{F}[\![\left(\overline{\tau_i}^n\right) \to \tau']\!] \Longrightarrow \left(v,\pi\mu\right) \in \mathscr{F}[\![\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu]\!] \tag{4.7}$$

Suponiendo un $(v,\pi) \in \mathscr{F}[\![\left(\overline{\tau_i}^n\right) \to \tau']\!]$. Teniendo en cuenta la definición de $\mathscr{F}[\![\_]\!]$ hay dos casos. En el primero, tenemos $v = \emptyset$ y $\pi \equiv [\,]$ (módulo $itv\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)$). Podemos aplicar el lema 5 para obtener $\pi\mu \equiv [\,]$ (módulo $itv\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu$) y entonces el lema 3 para obtener $\pi\mu \equiv [\,]$ (módulo $itv\left(\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right)$). Por lo tanto, $(v,\pi\mu) = (\emptyset,\pi\mu) \in \mathscr{F}[\![\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu]\!]$. En el segundo caso, tenemos que $v = f\big|_1$ y $\pi \in Dcp\left(f\big|_2, \left(\overline{\tau_i}^n\right) \to \tau'\right)$ para un f tal que

$$\emptyset \subset f \subseteq \left\{\left(\left(\overline{v_i}^n, v\right), \pi'\right) \mid \forall i \in \{1..n\}\,.\,\left(v_i,\pi'\right) \in \mathscr{T}[\![\tau_i]\!], \left(v,\pi'\right) \in \mathscr{T}[\![\tau']\!]\right\}$$

Vamos a definir $f\mu$ de la siguiente manera:

$$f\mu = \left\{\left(\left(\overline{v_i}^n, v\right), \pi'\mu\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right]\right) \mid \left(\left(\overline{v_i}^n, v\right), \pi'\right) \in f\right\}$$

donde $\{\overline{\alpha_i}\}$ es el conjunto de variables de tipo en $dom\,\mu \backslash rng\,\mu$. Para cada par $((\overline{v_i}^n, v), \pi') \in f$ se cumple que $(v_i,\pi') \in \mathscr{T}[\![\tau_i]\!]$ para cada $i \in \{1..n\}$ y $(v,\pi') \in \mathscr{T}[\![\tau']\!]$. Por hipótesis de inducción tenemos que $(v_i,\pi'\mu) \in \mathscr{T}[\![\tau_i\mu]\!]$ para cada $i \in \{1..n\}$ y $(v,\pi'\mu) \in \mathscr{T}[\![\tau'\mu]\!]$. Además, como

las variables $\{\overline{\alpha_i}\}$ no están libres en $\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu$, podemos aplicar el lema 1 para conseguir que $\left(v_i, \pi'\mu\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right]\right) \in \mathscr{T}\left[\!\left[\tau_i\mu\right]\!\right]$ para cada $i \in \{1..n\}$ y $\left(v, \pi'\mu\left[\overline{\alpha_i \mapsto \pi(\alpha_i)}\right]\right) \in \mathscr{T}\left[\!\left[\tau'\mu\right]\!\right]$. Por lo tanto, hemos demostrado que:

$$\emptyset \subset f\mu \subseteq \left\{\left(\left(\overline{v_i}^n, v\right), \pi''\right) \mid \forall i \in \{1..n\}.\left(v_i, \pi''\right) \in \mathscr{T}\left[\!\left[\tau_i\mu\right]\!\right], \left(v, \pi''\right) \in \mathscr{T}\left[\!\left[\tau'\mu\right]\!\right]\right\}$$

Es más, podemos aplicar el lema 2 sobre $\pi \in Dcp\left(f\big|_2, \left(\overline{\tau_i}^n\right) \to \tau'\right)$ para poder obtener que $\pi\mu \in Dcp\left(f\mu\big|_2, \left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right)$. Por lo tanto, $(v, \pi\mu) = (f\big|_1, \pi\mu) = (f\mu\big|_1, \pi\mu) \in \mathscr{F}\left[\!\left[\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right]\!\right]$ y así logramos demostrar (4.7).

Ahora para cada v, π, μ, α, $\overline{\tau_i}^n$ y τ' tales que $rng\,\mu \cap ftv\left(\left(\overline{\tau_i}^n\right) \to \tau'\right) = \emptyset$ podemos demostrar lo siguiente:

$$(v, \pi) \in \mathscr{S}\left[\!\left[\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right]\!\right] \Longrightarrow \left(v, \pi\mu\right) \in \mathscr{S}\left[\!\left[\left(\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right]\!\right] \tag{4.8}$$

Primero vamos a suponer que α no pertenece ni a $dom\,\mu$, ni a $rng\,\mu$. En este caso tendríamos que $\left(\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu = \forall\alpha.\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu$. Suponemos que $(v, \pi) \in \mathscr{S}\left[\!\left[\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right]\!\right]$. Entonces sabemos que $(v, \pi') \in \mathscr{F}\left[\!\left[\left(\overline{\tau_i}^n\right) \to \tau'\right]\!\right]$ para un π' tal que $\pi' \equiv \pi$ (módulo **TypeVar**$\setminus\{\alpha\}$). Dado que $ftv\left(\left(\overline{\tau_i}^n\right) \to \tau'\right) \subseteq ftv\left(\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right) \cup \{\alpha\}$ y α no pertenece a $rng\,\mu$, podemos asegurar que $ftv\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)$ es disjunto de $rng\,\mu$ y así aplicar (4.7) para obtener $(v, \pi'\mu) \in \mathscr{F}\left[\!\left[\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right]\!\right]$. Ahora denotemos con $\{\overline{\alpha_i}\}$ las variables en el conjunto $dom\,\mu \setminus rng\,\mu$. Con aquellas variables que no están libres en $\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu$, podemos aplicar el lema 1 obteniendo así $\left(v, \pi'\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]\right) \in \mathscr{F}\left[\!\left[\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right]\!\right]$. Podemos aplicar el lema 4 con $\pi' \equiv \pi$ (módulo **TypeVar**$\setminus\{\alpha\}$) para obtener $\pi'\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \equiv \pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]$ (módulo **TypeVar**$\setminus\{\alpha\}\mu$) o, ya que α no está en el dominio de μ, $\pi'\mu\left[\overline{\alpha_i \mapsto \emptyset}\right] \equiv \pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]$ (módulo **TypeVar**$\setminus\{\alpha\}$). Por la definición de la función $\mathscr{S}\left[\!\left[\_\right]\!\right]$ sabemos que $\left(v, \pi\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]\right) \in \mathscr{S}\left[\!\left[\forall\alpha.\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right]\!\right] = \mathscr{S}\left[\!\left[\left(\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right]\!\right]$. De nuevo, ya que las variables de tipo $\overline{\alpha_i}$ no están libres en el esquema de tipo (ya que α no aparece en ellas) podemos aplicar el lema 1 para obtener $(v, \pi\mu) \in \mathscr{S}\left[\!\left[\left(\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu\right]\!\right]$, tal como queríamos demostrar.

Ahora supongamos que α pertenece a $dom\,\mu \cup rng\,\mu$. En este caso tenemos $\left(\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right)\mu = \forall\beta.\left(\left(\overline{\tau_i[\alpha/\beta]}^n\right) \to \tau'[\alpha/\beta]\right)\mu$ donde β es una variable fresca que no aparece en $dom\,\mu \cup rng\,\mu \cup ftv\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)$. Suponemos que $(v, \pi) \in \mathscr{S}\left[\!\left[\forall\alpha.\left(\overline{\tau_i}^n\right) \to \tau'\right]\!\right]$. Entonces acabamos sabiendo que $(v, \pi') \in \mathscr{F}\left[\!\left[\left(\overline{\tau_i}^n\right) \to \tau'\right]\!\right]$ para un π' tal que $\pi' \equiv \pi$ (módulo **TypeVar**$\setminus\{\alpha\}$). Como β es fresca, podremos aplicar la propiedad (4.7) para obtener $(v, \pi'[\alpha/\beta]) \in \mathscr{F}\left[\!\left[\left(\overline{\tau_i[\alpha/\beta]}^n\right) \to \tau'[\alpha/\beta]\right]\!\right]$. En este momento sabemos que α no está libre en este tipo funcional, después de aplicar el renombramiento $[\alpha/\beta]$, pudiendo—por un lado—aplicar el lema 1 para obtener $(v, \pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]) \in \mathscr{F}\left[\!\left[\left(\overline{\tau_i[\alpha/\beta]}^n\right) \to \tau'[\alpha/\beta]\right]\!\right]$ y—por otro lado—asegurar que $rng\,\mu$ es disjunto de las variables libres en el tipo. Por ello aplicaremos (4.7) de nuevo para obtener $(v, \pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu) \in \mathscr{F}\left[\!\left[\left(\left(\overline{\tau_i[\alpha/\beta]}^n\right) \to \tau'[\alpha/\beta]\right)\mu\right]\!\right]$. A continuación usando el lema 1 tenemos como resultado que $\left(v, \pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]\right) \in \mathscr{F}\left[\!\left[\left(\left(\overline{\tau_i[\alpha/\beta]}^n\right) \to \tau'[\alpha/\beta]\right)\mu\right]\!\right]$ donde $\overline{\alpha_i}$ son aquellas variables que aparecen en $dom\,\mu \setminus rng\,\mu$. Entonces aplicamos a $\pi' \equiv \pi$ (módulo **TypeVar**$\setminus\{\alpha\}$) el lema 4 y tenemos que $\pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)] \equiv \pi[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]$ (módulo **TypeVar**$\setminus\{\beta\}$). Lo volvemos a aplicar de nuevo a la equivalencia obtenida, pero ahora con μ, para obtener

$\pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu[\overline{\alpha_i \mapsto \emptyset}] \equiv \pi[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu[\overline{\alpha_i \mapsto \emptyset}]$ (módulo **TypeVar**$\setminus \{\beta\}$). Dado lo anterior, podemos finalmente establecer que:

$$\left(v, \pi\left[\alpha/\beta\right]\left[\alpha \mapsto (\pi\mu)(\alpha)\right]\mu\left[\overline{\alpha_i \mapsto \emptyset}\right]\right) \in \mathscr{S}\left[\!\left[\forall\beta.\left(\left(\overline{\tau_i\left[\alpha/\beta\right]}^{n}\right) \to \tau'\left[\alpha/\beta\right]\right)\mu\right]\!\right].$$

Como las variables $\overline{\alpha_i}$ no están libres en el esquema de tipo vamos a aplicar el lema 1 (en el lado derecho) para obtener:

$$\left(v, \pi\left[\alpha/\beta\right]\left[\alpha \mapsto (\pi\mu)(\alpha)\right]\mu\right) \in \mathscr{S}\left[\!\left[\forall\beta.\left(\left(\overline{\tau_i\left[\alpha/\beta\right]}^{n}\right) \to \tau'\left[\alpha/\beta\right]\right)\mu\right]\!\right] \tag{4.9}$$

Ahora vamos a demostrar que, para toda variable de tipo $\gamma \neq \beta$,

$$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = (\pi\mu)(\gamma).$$

Suponemos que $\gamma \in rng\ \mu$. Entonces obtenemos:

$$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = (\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)])(\mu^{-1}(\gamma))$$

Si $\mu^{-1}(\gamma) = \alpha$, lo anterior sería equivalente a $\pi(\alpha) = \pi(\mu^{-1}(\gamma)) = (\pi\mu)(\gamma)$. De lo contrario obtendríamos

$$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = \pi[\alpha/\beta](\mu^{-1}(\gamma))$$

ya que al no estar β en el dominio de μ, esto es igual a $\pi(\mu^{-1}(\gamma)) = (\pi\mu)(\gamma)$, con lo que suponemos que $\gamma \in rng\ \mu$. De ello obtendríamos:

$$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = (\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)])(\gamma)$$

Si $\gamma = \alpha$ el lado derecho es equivalente a $\pi(\alpha) = \pi(\gamma) = (\pi\mu)(\gamma)$. De lo contrario obtendríamos:

$$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = (\pi[\alpha/\beta])(\gamma)$$

Ya que estamos suponiendo que $\gamma \neq \beta$, lo anterior es equivalente a $\pi(\gamma) = (\pi\mu)(\gamma)$. Por lo tanto, hemos demostrado que $\pi[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu \equiv \pi\mu$ (módulo **TypeVar**$\setminus \{\beta\}$). Así podemos reescribir (4.9) usando el lema 1 para obtener

$$\left(v, \pi\mu\right) \in \mathscr{S}\left[\!\left[\forall\beta.\left(\left(\overline{\tau_i\left[\alpha/\beta\right]}^{n}\right) \to \tau'\left[\alpha/\beta\right]\right)\mu\right]\!\right],$$

o, su equivalente,

$$\left(v, \pi\mu\right) \in \mathscr{S}\left[\!\left[\left(\forall\alpha.\left(\overline{\tau_i}^{n}\right) \to \tau'\right)\mu\right]\!\right],$$

que era el que queríamos para demostrar (4.8).

Finalmente, vamos a suponer que $(v, \pi) \in \mathscr{T}\left[\!\left[\bigsqcup_{i=1}^{n} \sigma_i\right]\!\right]$. Esto significa que v es la unión de un

f_i, para cada $i \in \{1..n\}$, tal que $(f_i, \pi) \in \mathscr{S}[\![\sigma_i]\!]$. Para cada $i \in \{1..n\}$ podemos aplicar repetidas veces la propiedad (4.8) (tantas veces como variables ligadas haya en σ_i) para obtener $(f_i, \pi\mu) \in \mathscr{S}[\![\sigma_i\mu]\!]$ y por ello $(v, \pi\mu) \in \mathscr{T}[\![(\bigsqcup_{i=1}^{n} \sigma_i)\mu]\!]$.

- **Caso** $\pi \vDash \alpha \subseteq \tau'$. Sabemos que $\alpha\mu$ es una variable de tipo. Para demostrar que $\pi\mu \vDash \alpha\mu \subseteq \tau'\mu$, vamos a suponer un $v \in (\pi\mu)(\alpha\mu)$. Primero demostraremos que $v \in \pi(\alpha)$ con una distinción de casos:

  - Si $\alpha \in dom\ \mu$ tenemos que $(\pi\mu)(\alpha\mu) = (\pi\mu)(\mu(\alpha)) = \pi(\mu^{-1}(\mu(\alpha))) = \pi(\alpha)$. Por lo tanto, $v \in \pi(\alpha)$.
  - Si $\alpha \notin dom\ \mu$ tenemos que $(\pi\mu)(\alpha\mu) = (\pi\mu)(\alpha)$. Puesto que α está libre en la restricción, esta no pertenecerá a $rng\ \alpha$, así que $(\pi\mu)(\alpha) = \pi(\alpha)$. Por lo tanto, $v \in \pi(\alpha)$.

  Dado que $v \in \pi(\alpha)$, por la definición de $\pi \vDash \alpha \subseteq \tau'$ tenemos que $(v, \pi') \in \mathscr{T}[\![\tau']\!]$ para un $\pi' \subseteq \pi$. Por hipótesis de inducción obtenemos que $(v, \pi'\mu) \subseteq \mathscr{T}[\![\tau'\mu]\!]$. Dado que $\pi' \subseteq \pi$ implica que $\pi'\mu \subseteq \pi\mu$. Por ello v pertenece al conjunto $\{v \mid (v, \pi'') \in \mathscr{T}[\![\tau'\mu]\!], \pi'' \subseteq \pi\mu\}$. Así que $\pi\mu \vDash \alpha\mu \subseteq \tau'\mu$ se cumple.

- **Caso** $\pi \vDash c \subseteq \tau$. Es similar al caso anterior.

- **Caso** $\pi \vDash \alpha \Leftarrow \tau'$. Sabemos que existe una familia de instanciaciones $\Pi = \{\pi_v \mid v \in \pi(\alpha)\}$, tales que $\pi = \bigcup\Pi$ y $(v, \pi_v) \in \mathscr{T}[\![\tau']\!]$ para cada $v \in \pi(\alpha)$. Ahora vamos a definir el conjunto $\Pi' = \{\pi'_v \mid v \in (\pi\mu)(\alpha\mu)\}$, donde cada instancia π'_v es definida como $\pi_v\mu$. Este está bien definido, ya que $(\pi\mu)(\alpha\mu) = \pi(\alpha)$ y hay un π_v por cada elemento $v \in \pi(\alpha)$. Con $(v, \pi_v) \in \mathscr{T}[\![\tau']\!]$ y la hipótesis de inducción tenemos que $(v, \pi'_v) \in \mathscr{T}[\![\tau'\mu]\!]$ para cada $v \in (\pi\mu)(\alpha\mu)$. Ahora vamos a demostrar que $\pi\mu = \bigcup\Pi'$. Suponemos una variable de tipo β. Si $\beta \in rng\ \mu$ obtenemos que:

$$\begin{aligned}(\pi\mu)(\beta) &= \pi(\mu^{-1}(\beta)) \\ &= \bigcup_{v \in \pi(\alpha)} \pi_v(\mu^{-1}(\beta)) \\ &= \bigcup_{v \in (\pi\mu)(\alpha\mu)} (\pi_v\mu)(\beta) \\ &= \bigcup_{v \in (\pi\mu)(\alpha\mu)} \pi'_v(\beta)\end{aligned}$$

  mientras que si $\beta \notin rng\ \mu$ obtendremos que:

$$\begin{aligned}(\pi\mu)(\beta) &= \pi(\beta) \\ &= \bigcup_{v \in \pi(\alpha)} \pi_v(\beta) \\ &= \bigcup_{v \in (\pi\mu)(\alpha\mu)} (\pi_v\mu)(\beta) \\ &= \bigcup_{v \in (\pi\mu)(\alpha\mu)} \pi'_v(\beta)\end{aligned}$$

  Por lo tanto hemos demostrado que $\pi\mu \vDash \alpha\mu \Leftarrow \tau'\mu$.

□

4.7.2. Lemas auxiliares para la corrección

En nuestro sistema de tipos hay algunas reglas que nos permiten derivar tipos para los patrones al obtener juicios de la forma $\Gamma \vdash p : \tau$. El significado que estas reglas pretenden tener viene dado por el siguiente resultado:

Lema 7. *Suponemos un entorno* Γ*, un patrón* p *y un tipo* τ *tal que* $\Gamma \vdash p : \tau$*. Entonces para toda instanciación* π *y valor* v *tales que* $\theta \in \mathcal{T}^{\pi}_{Env} [\![\Gamma]\!]$ *y* $(\theta, v) \in \mathcal{E} [\![p]\!]$ *se cumple que* $(v, \pi) \in \mathcal{T} [\![\tau]\!]$.

Demostración. Por inducción sobre la derivación de $\Gamma \vdash p : \tau$. Distinguimos casos en función de la última regla aplicada.

- **Caso [LIT$_P$]**

  En este caso tenemos $p = c$ para una constante c, así que se cumple que $(c, \pi) \in \mathcal{T} [\![c]\!]$.

- **Caso [VAR$_P$]**

  En este caso tenemos $p = x$ para una variable x tal que $\Gamma(x) = \alpha_x$ para una variable de tipo α_x y $v = \theta(x)$. Con $\theta \in \mathcal{T}^{\pi}_{Env} [\![\Gamma]\!]$ conseguimos que $(\theta(x), \pi) \in \mathcal{T} [\![\Gamma(x)]\!]$ o, equivalentemente, $(v, \pi) \in \mathcal{T} [\![\alpha_x]\!]$.

- **Caso [LST$_P$]**

  Ahora el patrón p tiene la forma $[p_1 \mid p_2]$ para unos patrones p_1 y p_2. Además v tiene la forma $([\_ \mid \_], v_1, v_2)$ para un v_1 y un v_2 tales que $(\theta, v_1) \in \mathcal{E} [\![p_1]\!]$ y $(\theta, v_2) \in \mathcal{E} [\![p_2]\!]$. Aplicando la hipótesis de inducción a la derivación de ambos patrones, obtenemos $(v_1, \pi) \in \mathcal{T} [\![\tau_1]\!]$ y $(v_2, \pi) \in \mathcal{T} [\![\tau_2]\!]$ que nos conduce a $(v, \pi) \in \mathcal{T} [\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}]\!]$.

- **Caso [TPL$_P$]**

  En este caso p tiene la forma $\{\overline{p_i}^n\}$ para unos patrones $p_1, \ldots, p_n$. Además v tiene la forma $(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n)$ para unos $v_1, \ldots, v_n$. Aplicamos la hipótesis de inducción a cada subderivación $\Gamma \vdash p_i : \tau_i$ para obtener que $(v_i, \pi) \in \mathcal{T} [\![\tau_i]\!]$ para cada $i \in \{1..n\}$, por ello $(v, \pi) \in \mathcal{T} [\![\{\overline{\tau_i}^n\}]\!]$.

□

El siguiente resultado nos muestra que, siempre que descartemos los tipos de algunas variables en los entornos de un tipo anotado con la notación $\rho \setminus \{x_1, \ldots, x_n\}$, obtendremos tipos anotados con una semántica igual o mayor.

Lema 8. *Suponemos que* $(\theta, v) \in \mathcal{T} [\![\rho]\!]$*. Para cada variable* x *y cada valor* v' *se cumple que* $(\theta[x/v'], v) \in \mathcal{T} [\![\rho \setminus \{x\}]\!]$.

Demostración. Cuando $(\theta, v) \in \mathcal{T}\,[\![\rho_1;\rho_2]\!]$, sabemos que $(\theta, v) \in \mathcal{T}\,[\![\rho_1]\!]$ o $(\theta, v) \in \mathcal{T}\,[\![\rho_2]\!]$. Entonces por hipótesis de inducción obtenemos que $(\theta[x/v'], v) \in \mathcal{T}\,[\![\rho_1 \setminus \{x\}]\!]$ o $(\theta[x/v'], v) \in \mathcal{T}\,[\![\rho_2 \setminus \{x\}]\!]$, que se convierte en $(\theta[x/v'], v) \in \mathcal{T}\,[\![(\rho_1;\rho_2)\setminus\{x\}]\!]$.

Cuando $(\theta, v) \in \mathcal{T}\,[\![\langle\tau;\Gamma\rangle]\!]$, sabemos que $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$ y $(v,\pi) \in \mathcal{T}\,[\![\tau]\!]$. Con $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$, sabemos que $\forall z \in \mathbf{Var}.(\theta(z),\pi) \in \mathcal{T}\,[\![\Gamma(z)]\!]$ y $\pi \vDash \Gamma|_C$, entonces obtenemos:

$$
\begin{array}{lll}
 & \forall z \in \mathbf{Var}.(\theta(z),\pi) \in \mathcal{T}\,[\![\Gamma(z)]\!] & \\
\Leftrightarrow & \forall z \in \mathbf{Var}\setminus\{x\}.(\theta(z),\pi) \in \mathcal{T}\,[\![\Gamma(z)]\!] \vee \forall z \in \{x\}.(\theta(z),\pi) \in \mathcal{T}\,[\![\Gamma(z)]\!] & \\
 & \quad \text{ya que } \Gamma(x) = \mathsf{any}() & \\
\Rightarrow & \forall z \in \mathbf{Var}\setminus\{x\}.(\theta(z),\pi) \in \mathcal{T}\,[\![\Gamma(z)]\!] \vee \forall z \in \{x\}.(v',\pi) \in \mathcal{T}\,[\![\mathsf{any}()]\!] & \\
\Leftrightarrow & \forall z \in \mathbf{Var}.(\theta[x/v'](z),\pi) \in \mathcal{T}\,[\![(\Gamma\setminus\{x\})(z)]\!] &
\end{array}
$$

Dado que $\Gamma|_C = \Gamma\setminus\{x\}|_C$, con $\forall z \in \mathbf{Var}.(\theta[x/v'](z),\pi) \in \mathcal{T}\,[\![(\Gamma\setminus\{x\})(z)]\!]$ y $\pi \vDash \Gamma\setminus\{x\}|_C$, obtenemos que $\theta[x/v'] \in \mathcal{T}^{\pi}_{Env}[\![\Gamma\setminus\{x\}]\!]$; y con $(v,\pi) \in \mathcal{T}\,[\![\tau]\!]$, obtenemos que $(\theta[x/v'], v) \in \mathcal{T}\,[\![\langle\tau;\Gamma\rangle\setminus\{x\}]\!]$. □

Con el siguiente resultado podremos comprobar cuándo una instanciación satisface algunas restricciones de subconjunto particulares.

Lema 9. *Para cada π, α_1 y α_2:*

1. *$\pi \vDash \alpha_1 \subseteq \alpha_2$ si y solo si $\pi(\alpha_1) \subseteq \pi(\alpha_2)$.*
2. *$\pi \vDash \alpha_1 = \alpha_2$ si y solo si $\pi(\alpha_1) = \pi(\alpha_2)$.*

Demostración. Vamos a demostrar (1): ($\Rightarrow$) Suponiendo que $\pi \vDash \alpha_1 \subseteq \alpha_2$. Dado un $v \in \pi(\alpha_1)$ se cumple que $(v,\pi') \in \mathcal{T}\,[\![\alpha_2]\!]$ para un $\pi' \subseteq \pi$. Esto significa que $\pi'(\alpha_2) = \{v\}$ y por consiguiente $v \in \pi(\alpha_2)$. Por lo tanto $\pi(\alpha_1) \subseteq \pi(\alpha_2)$. ($\Leftarrow$) Vamos a demostrar que $\pi \vDash \alpha_1 \subseteq \alpha_2$. Para cualquier $v \in \pi(\alpha_1)$ se cumple que $v \in \pi(\alpha_2)$. Por lo tanto $\pi[\alpha_2 \mapsto \{v\}] \subseteq \pi$ y $(v, \pi[\alpha_2 \mapsto \{v\}]) \in \mathcal{T}\,[\![\alpha_2]\!]$.

La propiedad (2) se puede demostrar aplicando (1) dos veces. □

4.7.3. Teorema de corrección

El resultado más importante de este capítulo determina que, siempre que obtengamos un juicio de la forma $\Gamma \vdash e : \rho$, el tipo anotado ρ es un *success type* para e.

Teorema 2. *Suponemos un entorno Γ, una expresión e y un tipo anotado ρ. Si $\Gamma \vdash e : \rho$ entonces*

$$\mathcal{E}\,[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}\,[\![\rho]\!]$$

Demostración. Por inducción sobre el tamaño de la derivación del tipo. Distinguimos los casos según la última regla aplicada.

- **Caso [SUB1]**

  Suponemos una derivación $\Gamma' \vdash e : \rho$ para un $\Gamma \subseteq \Gamma'$. Entonces tenemos que:

  $$\begin{array}{lll} & \mathscr{E}[\![e]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathscr{E}[\![e]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]} & \text{ya que } \mathscr{T}_{Env}[\![\Gamma]\!] \subseteq \mathscr{T}_{Env}[\![\Gamma']\!] \\ \subseteq & \mathscr{T}[\![\rho]\!] & \text{por h.i.} \end{array}$$

- **Caso [SUB2]**

  En este caso suponemos una derivación $\Gamma \vdash e : \rho'$ para un $\rho \subseteq \rho'$. Entonces tenemos que:

  $$\begin{array}{lll} & \mathscr{E}[\![e]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathscr{T}[\![\rho]\!] & \text{por h.i.} \\ \subseteq & \mathscr{T}[\![\rho']\!] & \text{ya que } \mathscr{T}[\![\rho]\!] \subseteq \mathscr{T}[\![\rho']\!] \end{array}$$

- **Caso [CNS]**

  En este caso la expresión e para tipar es un literal c. Suponemos que $(\theta, v) \in \mathscr{E}[\![c]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$. Esto significa que $v = c$ y que $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$. Esto último nos lleva a la existencia de una instanciación π tal que $\pi \vDash \Gamma|_C$ y $(\theta(x), \pi) \in \mathscr{T}[\![\Gamma(x)]\!]$ para todo $x \in \mathbf{Var}$. Además, obtenemos que $(c, \pi) \in \mathscr{T}[\![c]\!]$, y así finalmente tenemos $(\theta, c) \in \mathscr{T}[\![\langle c; \Gamma \rangle]\!]$.

- **Caso [VAR]**

  Sabemos que la expresión e es una variable $x \in \mathbf{Var}$. Suponemos el par $(\theta, v) \in \mathscr{E}[\![x]\!]$ tal que $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$. En este caso $v = \theta(x)$. Sabemos que existe una instanciación π tal que $\theta \in \mathscr{T}^{\pi}_{Env}[\![\Gamma]\!]$. En particular, $(\theta(x), \pi) \in \mathscr{T}_{Env}[\![\Gamma(x)]\!]$. Por lo tanto, $(\theta, v) = (\theta, \theta(x)) \in \mathscr{T}[\![\langle \Gamma(x); \Gamma \rangle]\!]$, que demuestra el teorema.

- **Caso [TPL]**

  Suponemos que el par $(\theta, v) \in \mathscr{E}[\![\{\overline{e_i}^n\}]\!]$ con $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$. En este caso v es un valor de la forma $(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n)$, donde $v_i \in \mathscr{E}[\![e_i]\!]$ para cada $i \in \{1..n\}$. Por hipótesis de inducción, ya que $(\theta, v_i) \in \mathscr{E}[\![e_i]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$, se cumple que $(\theta, v_i) \in \mathscr{T}[\![\rho_i]\!]$ para cada i. Con el lema 10 obtenemos que $(\theta, v) \in \mathscr{T}[\![\rho_1 \otimes \cdots \otimes \rho_n]\!]$.

- **Caso [LST]**

  Suponemos que $(\theta, v) \in \mathscr{E}[\![[e_1 \mid e_2]]\!]$, tal que la $v = ([\_ \mid \_], v_1, v_2)$ para algunos valores v_1, v_2. De forma similar al caso anterior, podemos aplicar la hipótesis de inducción y el lema 10 para obtener que $(\theta, (\{\overset{2}{\cdots}\}, v_1, v_2)) \in \mathscr{T}[\![\rho_1 \otimes \rho_2]\!] = \mathscr{T}\left[\!\!\left[\overline{\langle \{\tau_i, \tau'_i\}; \Gamma_i \rangle}^n\right]\!\!\right]$, así que $(\theta, (\{\overset{2}{\cdots}\}, v_1, v_2))$ pertenece a $\mathscr{T}[\![\langle \{\tau_k, \tau'_k\}; \Gamma_k \rangle]\!]$ para un $k \in \{1..n\}$. Existe una instanciación π tal que $(\theta(x), \pi) \in \mathscr{T}[\![\Gamma_k(x)]\!]$, $\pi \vDash \Gamma_k|_C$ y $((\{\overset{2}{\cdots}\}, v_1, v_2), \pi) \in \mathscr{T}[\![\langle \{\tau_k, \tau'_k\}; \Gamma_k \rangle]\!]$. Esto último implica que $(v_1, \pi) \in \mathscr{T}[\![\tau_k]\!]$ y

$(v_2, \pi) \in \mathcal{T}\,[\![\tau'_k]\!]$. Ahora podemos aplicar la definición semántica de $\texttt{nelist}(\tau_k, \tau'_k)$ para conseguir $((\texttt{[\_|\_]}, v_1, v_2), \pi) \in \mathcal{T}\,[\![\texttt{nelist}(\tau_k, \tau'_k)]\!]$. Por lo tanto, tenemos que:

$$(\theta, v) \in \mathcal{T}\,[\![\langle \texttt{nelist}(\tau_k, \tau'_k); \Gamma_k \rangle]\!]$$

para algún $k \in \{1..n\}$ y por consiguiente

$$(\theta, v) \in \mathcal{T}\,\left[\!\left[\overline{\langle \texttt{nelist}(\tau_k, \tau'_k); \Gamma_k \rangle}^{\,n} \right]\!\right].$$

- **Caso [ABS]**

  En este caso $e = \textbf{fun}(\overline{x_i}^{\,n}) \to e'$ para algunas variables $\overline{x_i}^{\,n}$ y una expresión e'. Suponemos un par $(\theta, v) \in \mathcal{E}\,[\![e]\!]\,\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. Con la definición semántica de $\mathcal{E}\,[\![\textbf{fun}(\overline{x_i}^{\,n}) \to e']\!]$ obtenemos que v es el grafo de una función de aridad n. Es más, sabemos que $\theta \in \mathcal{T}\,[\![\Gamma]\!]$ implica la existencia de una instanciación π tal que:

  $$\forall z \in \textbf{Var}.(\theta(z), \pi) \in \mathcal{T}\,[\![\Gamma(z)]\!] \qquad \pi \vDash \Gamma|_C \tag{4.10}$$

  Supongamos una tupla $w \in v$. Entonces v tiene la forma $((\overline{v_i}^{\,n}), v')$ para algunos valores $\overline{v_i}^{\,n}, v'$ tales que $(\theta[\overline{x_i / v_i}^{\,n}], v') \in \mathcal{E}\,[\![e']\!]$. Dado que $\Gamma(x_i) = \texttt{any()}$ para cada $i \in \{1..n\}$, obtendremos que $(\theta[\overline{x_i / v_i}^{\,n}], v') \in \mathcal{T}\,[\![e']\!]\,\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. Por lo tanto, la tupla $(\theta[\overline{x_i / v_i}^{\,n}], v')$ también pertenece a la semántica de $\mathcal{T}\,\left[\!\left[\overline{\langle \tau_j; \Gamma[\overline{x_i : \tau_{j,i}}^{\,n}] \rangle}^{\,m}\right]\!\right]$. En particular, pertenece a $\mathcal{T}\,[\![\langle \tau_k; \Gamma[\overline{x_i : \tau_{k,i}}^{\,n}] \rangle]\!]$ para algún $k \in \{1..m\}$. Como resultado, hemos demostrado lo siguiente

  $$\begin{array}{c} \text{Para cada } w = ((\overline{v_i}^{\,n}), v') \in v \text{ existe un } k_w \in \{1..m\} \\ \text{tal que } (\theta[\overline{x_i / v_i}^{\,n}], v') \in \mathcal{T}\,[\![\langle \tau_k; \Gamma[\overline{x_i : \tau_{k,i}}^{\,n}] \rangle]\!] \end{array} \tag{4.11}$$

  Ponemos el subíndice w en k_w para destacar que el índice k es posiblemente distinto para cada tupla w. Para cada $j \in \{1..m\}$, vamos a definir el conjunto W_j de la siguiente manera:

  $$W_j = \{w \in v \mid k_w = j\}$$

  Con (4.11) tenemos que $v = W_1 \cup \ldots \cup W_m$. Ahora vamos a suponer que, dado un $j \in \{1..m\}$, si W_j es vacío, vamos a definir una instanciación π' de la siguiente manera:

  $$\forall \alpha \in \textbf{TypeVar} : \pi'(\alpha) = \begin{cases} \emptyset & \text{si } \alpha \in itv\left((\overline{\tau_{j,i}}^{\,n}) \to \tau_j\right) \\ \pi(\alpha) & \text{en otro caso} \end{cases}$$

  Entonces, teniendo en cuenta la definición semántica de los tipos funcionales, sabemos que $(\emptyset, \pi') \in \mathcal{F}\,[\![(\overline{\tau_{j,i}}^{\,n}) \to \tau_j]\!]$. De la manera en que π' está definida, se cumple que $\pi \equiv \pi'$ (módulo $\textbf{TypeVar} \setminus itv\left((\overline{\tau_{j,i}}^{\,n}) \to \tau_j\right)$), así que $(\emptyset, \pi) \in \mathcal{S}\,[\![\overline{\forall}\,(\overline{\tau_{j,i}}^{\,n}) \to \tau_j]\!]$. De forma equivalente, $(W_j, \pi) \in$

$\mathscr{S}[\![\overline{\forall}(\overline{\tau_{j,i}}^n) \to \tau_j]\!]$ ya que estamos suponiendo que W_j está vacío.

Ahora vamos a suponer que W_j no esté vacío. Supongamos una tupla $w \in W_j$ de la forma $((\overline{v_i}^n), v')$. Con (4.11) se cumple que $(\theta[\overline{x_i / v_i}^n], v') \in \mathscr{T}[\![\langle \tau_j; \Gamma[\overline{x_i : \tau_{j,i}}^n]\rangle]\!]$. Esto implica la existencia de una instanciación π_w tal que:

$$\left.\begin{array}{ll} (v_i, \pi_w) \in \mathscr{T}[\![\tau_{j,i}]\!] & \text{para cada } i \in \{1..n\} \\ (\theta(z), \pi_w) \in \mathscr{T}[\![\Gamma(z)]\!] & \text{para cualquier otra } z \notin \{\overline{x_i}^n\} \\ (v', \pi_w) \in \mathscr{T}[\![\tau_j]\!] & \\ \pi_w \vDash \Gamma|_C & \end{array}\right\} \tag{4.12}$$

Nuevamente, escribimos π_w para remarcar que cada tupla $w \in W_j$ tiene su propia π_w como testigo de que w pertenece a $\mathscr{T}[\![\langle \tau_j; \Gamma[\overline{x_i : \tau_{j,i}}^n]\rangle]\!]$. Para cada una de estas π_w, vamos a definir una instanciación π'_w de la siguiente manera:

$$\forall \alpha \in \textbf{TypeVar} : \pi'_w(\alpha) = \begin{cases} \pi_w(\alpha) & \text{si } \alpha \in itv\big((\overline{\tau_{j,i}}^n) \to \tau_j\big) \\ \pi(\alpha) & \text{en otro caso} \end{cases}$$

donde π es una instanciación que satisface (4.10). Ahora vamos a demostrar que $\pi_w \equiv \pi'_w$ (módulo $ftv\big((\overline{\tau_{j,i}}^n) \to \tau_j\big)$). Supongamos una variable de tipo α.

- Si $\alpha \in itv\big((\overline{\tau_{j,i}}^n) \to \tau_j\big)$, entonces $\pi_w(\alpha) = \pi'_w(\alpha)$ por la definición de π'_w.
- Si $\alpha \in ftv\big((\overline{\tau_{j,i}}^n) \to \tau_j\big)$ pero $\alpha \notin itv\big((\overline{\tau_{j,i}}^n) \to \tau_j\big)$, entonces $\alpha \in ftv\big(\overline{\forall}(\overline{\tau_{j,i}}^n) \to \tau_j\big)$. La regla [ABS] especifica que existe un y tal que $\Gamma(y) = \alpha$ y $y \notin \{\overline{x_i}^n\}$, es decir, α es una de las variables de tipo $\overline{\alpha_i}$ que han sido usadas al aplicar [ABS]. Por lo tanto, obtenemos que $(\theta(y), \pi_w) \in \mathscr{T}[\![\Gamma(y)]\!] = \mathscr{T}[\![\alpha]\!]$, lo que implica $\pi_w(\alpha) = \{\theta(y)\}$. Por otro lado, con (4.10) obtenemos que $\pi(\alpha) = \{\theta(y)\}$. Por consiguiente, $\pi'_w(\alpha) = \pi(\alpha) = \{\theta(y)\} = \pi_w(\alpha)$.

Como $\pi_w \equiv \pi'_w$ (módulo $ftv\big((\overline{\tau_{j,i}}^n) \to \tau_j\big)$) podemos aplicar el lema 1 a (4.12) para obtener:

$$\begin{array}{ll} (v_i, \pi'_w) \in \mathscr{T}[\![\tau_{j,i}]\!] & \text{para cada } i \in \{1..n\} \\ (v', \pi'_w) \in \mathscr{T}[\![\tau_j]\!] & \end{array}$$

Ahora vamos a definir $\Pi_j = \{\pi'_w \mid w \in W_j\}$ y $\pi_j = \bigcup \Pi_j$. Obviamente, por la definición de π'_w, se cumple que $\pi_j \equiv \pi$ (módulo $\textbf{TypeVar} \setminus itv\big((\overline{\tau_{j,i}}^n) \to \tau_j\big)$). Por lo tanto, $\pi_j \in Dcp\big(\Pi_j, (\overline{\tau_{j,i}}^n) \to \tau_j\big)$, lo que implica que $(W_j, \pi_j) \in \mathscr{F}[\![(\overline{\tau_{j,i}}^n) \to \tau_j]\!]$ para cada $j \in \{1..m\}$. Es más, obtenemos que $(W_j, \pi) \in \mathscr{S}[\![\overline{\forall}(\overline{\tau_{j,i}}^n) \to \tau_j]\!]$ para cada $j \in \{1..m\}$ (de nuevo como consecuencia de ser π_j igual a π módulo $\textbf{TypeVar} \setminus itv\big((\overline{\tau_{j,i}}^n) \to \tau_j\big)$). Por ello $(v, \pi) = (W_1 \cup \ldots \cup W_m, \pi) \in \mathscr{T}\left[\!\!\left[\bigsqcup_{j=1}^m \overline{\forall}(\overline{\tau_{j,i}}^n) \to \tau_j\right]\!\!\right]$, y junto a (4.10): $(\theta, v) \in \mathscr{T}\left[\!\!\left[\left\langle \bigsqcup_{j=1}^m \overline{\forall}(\overline{\tau_{j,i}}^n) \to \tau_j; \Gamma\right\rangle\right]\!\!\right]$.

- **Caso [APP1]**

En este caso $e = f(\overline{x_i}^n)$ para algún símbolo funcional f y algunas variables $\overline{x_i}^n$. Dado que $(\theta, v) \in \mathscr{E}[\![f(\overline{x_i}^n)]\!]$, obtenemos que $\theta(f)$ es un grafo—de una función de aridad n—que contiene la tupla $((\overline{\theta(x_i)}^n), v)$ dentro de él. Ya que $\theta(f)$ es una función de aridad n sabemos que se cumple que $\theta \in \mathscr{T}_{Env}[\![\lfloor f : (\overline{\mathsf{any}()}^n) \to \mathsf{any}() \rfloor]\!]$, y como $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$, obtenemos que $\theta \in \mathscr{T}_{Env}[\![\Gamma_0]\!]$, donde Γ_0 es el entorno que aparece en la regla [APP1]. Esta regla nos permite suponer que la función f tiene el siguiente esquema sobrecargado en Γ_0:

$$\Gamma_0(f) = \bigsqcup_{j=1}^{m} \forall \overline{\alpha_{j,i}}.(\overline{\tau_{j,i}}^n) \to \tau_j$$

Esto significa que existe una instanciación π tal que $(\theta(f), \pi) \in \mathscr{T}[\![\bigsqcup_{j=1}^m \forall \overline{\alpha_{j,i}}.(\overline{\tau_{j,i}}^n) \to \tau_j]\!]$ y $\pi(\alpha_{x_i}) = \{\theta(x_i)\}$ para cada parámetro x_i donde $i \in \{1..n\}$. Teniendo en cuenta la definición semántica de un tipo de esquema sobrecargado, se cumple que $\theta(f) = f_1 \cup \ldots \cup f_m$, donde cada f_i es el subgrafo correspondiente a cada rama del tipo sobrecargado. Es decir, para cada $j \in \{1..m\}$ tenemos que $(f_j, \pi) \in \mathscr{S}[\![\forall \overline{\alpha_{j,i}}.(\overline{\tau_{j,i}}^n) \to \tau_j]\!]$. Ya que sabemos que $((\overline{\theta(x_i)}^n), v)$ está dentro de $\theta(f)$, por lo que tiene que pertenecer a algún f_k donde $k \in \{1..m\}$. Con la definición de $\mathscr{S}[\![\_]\!]$ obtenemos que $(f_k, \pi') \in \mathscr{F}[\![(\overline{\tau_{k,i}}^n) \to \tau_k]\!]$ para un $\pi' \equiv \pi$ (módulo $\mathbf{TypeVar} \setminus \{\overline{\alpha_{k,i}}\}$). Es más, obtenemos que:

$$\forall i \in \{1..n\}.(\theta(x_i), \pi'') \in \mathscr{T}[\![\tau_{k,i}]\!] \quad \text{y} \quad (v, \pi'') \in \mathscr{T}[\![\tau_k]\!] \tag{4.13}$$

para un $\pi'' \subseteq \pi'$ tal que $\pi'' \equiv \pi'$ (módulo $\mathbf{TypeVar} \setminus itv((\overline{\tau_{k,i}}^n) \to \tau_k)$). Vamos a denotar con $\{\overline{\beta_i}\}$ el conjunto de variables en $itv((\overline{\tau_{k,i}}^n) \to \tau_k) \cup \{\overline{\alpha_{k,i}}\}$ y vamos a suponer otro conjunto $\{\overline{\beta'_i}\}$ de variables frescas tales que $\mu_k = [\overline{\beta_i / \beta'_i}]$ de acuerdo con la regla [APP1]. Como las $\overline{\beta'_i}$ son variables frescas, podemos aplicar el lema 6 a (4.13) para obtener:

$$\forall i \in \{1..n\}.(\theta(x_i), \pi''\mu_k) \in \mathscr{T}[\![\tau_{k,i}\mu_k]\!] \quad \text{y} \quad (v, \pi''\mu_k) \in \mathscr{T}[\![\tau_k\mu_k]\!] \tag{4.14}$$

Nótese que ninguna de las $\overline{\beta_i}$ aparece en $\tau_{k,i}\mu_k$ o $\tau_k\mu_k$, ya que todas estas variables de tipo han sido renombradas con la aplicación de μ_k. Esto nos permite definir la instanciación $\pi_\circ = (\pi''\mu_k)[\overline{\beta_i \mapsto \pi(\beta_i)}]$ para que $\pi_\circ \equiv \pi''\mu_k$ (módulo $ftv(\tau_{k,i}\mu_k)$) para cada $i \in \{1..n\}$ y $\pi_\circ \equiv \pi''\mu_k$ (módulo $ftv(\tau_k\mu_k)$). De esta manera podemos usar el lema 1 para transformar (4.14) de la siguiente forma:

$$\forall i \in \{1..n\}.(\theta(x_i), \pi_\circ) \in \mathscr{T}[\![\tau_{k,i}\mu_k]\!] \quad \text{y} \quad (v, \pi_\circ) \in \mathscr{T}[\![\tau_k\mu_k]\!] \tag{4.15}$$

Ahora denotaremos con $\pi_\bullet$ la instanciación $\pi[\overline{\beta'_i \mapsto \pi''(\beta_i)}]$ y demostraremos que $\pi_\circ = \pi_\bullet$. Suponemos un $\alpha \in \mathbf{TypeVar}$:

- Si $\alpha = \beta'_i$ para algún i tenemos que $\pi_\circ(\beta'_i) = (\pi''\mu_k)(\beta'_i) = \pi''(\beta_i) = \pi_\bullet(\beta'_i)$.
- Si $\alpha = \beta_i$ para algún i obtenemos que $\pi_\circ(\beta_i) = \pi(\beta_i) = \pi_\bullet(\beta_i)$.

- Si no α no pertenece a $itv\left(\left(\overline{\tau_{k,i}}^{n}\right) \to \tau_k\right) \cup \{\overline{\alpha_{k,i}}\}$. Esto significa que:

$$\pi_\circ(\alpha) = (\pi'' \mu_k)(\alpha) = \pi''(\alpha) = \pi'(\alpha) = \pi(\alpha) = \pi_\bullet(\alpha)$$

  ya que se cumple que $\pi'' \equiv \pi'$ (módulo **TypeVar**$\setminus itv\left(\left(\overline{\tau_{k,i}}^{n}\right) \to \tau_k\right)$) y también que $\pi'' \equiv \pi'$ (módulo **TypeVar**$\setminus itv\left(\left(\overline{\tau_{k,i}}^{n}\right) \to \tau_k\right)$).

Como consecuencia, podemos sustituir $\pi_\circ$ por $\pi_\bullet$ en (4.15) y obtener:

$$\forall i \in \{1..n\}.\, (\theta(x_i), \pi_\bullet) \in \mathcal{T}\,[\![\tau_{k,i}\mu_k]\!] \quad \text{y} \quad (v, \pi_\bullet) \in \mathcal{T}\,[\![\tau_k \mu_k]\!] \tag{4.16}$$

Ya que las variables $\overline{\beta'_i}$ son diferentes de las $\overline{\alpha_{x_i}}^{n}$, sabemos que $\pi_\bullet(\alpha_{x_i}) = \pi(\alpha_{x_i}) = \{\theta(x_i)\}$ y por ello con (4.16) sabremos que $\pi_\bullet \vDash \tau_{k,i}\mu_k \Leftarrow \alpha_{x_i}$ para cada $i \in \{1..n\}$. Es más, como $\pi \equiv \pi_\bullet$ (módulo **TypeVar**$\setminus \{\beta'_i\}$) y las variables β'_i no aparecen en Γ_0, obtenemos lo siguiente con el lema 1:

$$\forall y.\, \left(\theta(y), \pi_\bullet\right) \in \mathcal{T}\,[\![\Gamma_0(y)]\!] \quad \text{y} \quad \pi_\bullet \vDash \Gamma_0|_C \tag{4.17}$$

Ahora vamos a demostrar que $\pi_\bullet \vDash \beta\mu_k \subseteq \beta$ para cada $\beta \in itv\left(\forall \overline{\alpha_{k,i}}.\, \left(\overline{\tau_{k,i}}^{n}\right) \to \tau_k\right)$. Esto significa que β es alguna de las β_i definidas antes, pero no es ninguna de las $\{\overline{\alpha_{k,i}}\}$. Por ello tenemos que demostrar que $\pi_\bullet \vDash \beta'_i \subseteq \beta_i$. Suponemos que $v \in \pi_\bullet(\beta'_i)$. Tenemos que:

$$v \in \pi_\bullet(\beta'_i) = \pi''(\beta_i) \subseteq \pi'(\beta_i) = \pi(\beta_i) = \pi_\bullet(\beta_i)$$

así que $(v, [\beta_i \mapsto \{v\}]) \in \mathcal{T}\,[\![\beta_i]\!]$ y $[\beta_i \mapsto \{v\}] \subseteq \pi_\bullet$. Por lo tanto, $\pi_\bullet \vDash \beta_i\mu_k \subseteq \beta_i$ que es lo que queríamos demostrar. Dado este resultado y la ecuación (4.16) y (4.17) obtenemos:

$$\begin{aligned}
&(\theta, v) \in \mathcal{T}\,[\![\langle \tau_k\mu_k ; \Gamma_k \rangle]\!] \subseteq \mathcal{T}\,\left[\!\!\left[\overline{\langle \tau_j\mu_j ; \Gamma_j \rangle}^{m}\right]\!\!\right] \\
&\text{donde } \Gamma_k = \Gamma_0\left[\{\overline{\tau_{k,i}\mu_k \Leftarrow \alpha_{x_i}}^{n}\} \cup \{\beta\mu_k \subseteq \beta \mid \beta \in itv\left(\forall \overline{\alpha_{k,i}}.\, \left(\overline{\tau_{k,i}}^{n}\right) \to \tau_k\right)\}\right]
\end{aligned}$$

que demuestra el teorema.

- **Caso [APP2]**

  En este caso tenemos que demostrar que se cumple $\mathcal{E}\,[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}\,[\![\langle \mathsf{none}() ; \bot \rangle]\!]$ o, su equivalente, $\mathcal{E}\,[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} = \emptyset$. Vamos a demostrarlo por reducción al absurdo. Suponemos que $(\theta, v) \in \mathcal{E}\,[\![e]\!]$ y $\theta \in \mathcal{T}_{Env}\,[\![\Gamma]\!]$. Como en el caso de [APP1], esto implica que $\theta(f)$ es el grafo de una función de aridad n, así que $\theta \in \mathcal{T}_{Env}\left[\!\!\left[\left[f : \left(\overline{\mathsf{any}()}^{n}\right) \to \mathsf{any}()\right]\right]\!\!\right]$. Por lo tanto, con el paso anterior, obtenemos que $\theta \in \mathcal{T}_{Env}\left[\!\!\left[\Gamma \sqcap \left[f : \left(\overline{\mathsf{any}()}^{n}\right) \to \mathsf{any}()\right]\right]\!\!\right]$. Sin embargo, con la condición de la regla [APP2] obtenemos que $\theta \in \mathcal{T}_{Env}\,[\![\bot]\!] = \emptyset$, llevándonos a una contradicción.

- **Caso [LET]**

  Supongamos la tupla $(\theta, v) \in \mathcal{E}\,[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. Entonces existe un v_1 tal que $(\theta, v_1) \in \mathcal{E}\,[\![e_1]\!]$ y $(\theta[x/v_1], v) \in \mathcal{E}\,[\![e_2]\!]$. Como $\theta \in \mathcal{T}_{Env}\,[\![\Gamma]\!]$ obtenemos que $(\theta, v_1) \in \mathcal{E}\,[\![e_1]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$ y por ello

es cierto que $(\theta, v_1) \in \mathcal{T}\left[\!\left[\overline{\langle \tau_i ; \Gamma_i \rangle}^n\right]\!\right]$, donde $\overline{\langle \tau_i ; \Gamma_i \rangle}^n$ es la secuencia de pares usados al aplicar la regla [LET]. Esto significa que $(\theta, v_1) \in \mathcal{T}[\![\langle \tau_k ; \Gamma_k \rangle]\!]$ para un $k \in \{1..n\}$. En particular existe un π tal que:

$$\forall z \in \mathbf{Var}.(\theta(z), \pi) \in \mathcal{T}[\![\Gamma_k(z)]\!]$$
$$(v_1, \pi) \in \mathcal{T}[\![\tau_k]\!]$$
$$\pi \vDash \Gamma_k|_C$$

que podemos rescribir del siguiente modo:

$$\forall z \in \mathbf{Var} \setminus \{x\}.(\theta[x/v_1](z), \pi) \in \mathcal{T}[\![\Gamma_k[x : \tau_k](z)]\!]$$
$$(\theta[x/v_1](x), \pi) \in \mathcal{T}[\![\Gamma_k[x : \tau_k](x)]\!]$$
$$\pi \vDash \Gamma_k[x : \tau_k]|_C$$

Los dos primeros hechos pueden fusionarse para obtener:

$$\forall z \in \mathbf{Var}.(\theta[x/v_1](z), \pi) \in \mathcal{T}[\![\Gamma_k[x : \tau_k](z)]\!]$$
$$\pi \vDash \Gamma_k[x : \tau_k]|_C$$

y por lo tanto $\theta[x/v_1] \in \mathcal{T}_{Env}[\![\Gamma_k[x : \tau_k]]\!]$. Como $(\theta[x/v_1], v) \in \mathcal{E}[\![e_2]\!]$ podemos aplicar la hipótesis de inducción sobre la derivación de $\Gamma_k[x : \tau_k] \vdash e_2 : \rho_k$ para obtener que $(\theta[x/v_1], v) \in \mathcal{T}[\![\rho_k]\!]$. Con el lema 8 obtenemos $(\theta, v) \in \mathcal{T}[\![\rho_k \setminus \{x\}]\!] \subseteq \mathcal{T}\left[\!\left[\overline{\rho_i \setminus \{x\}}^n\right]\!\right]$, que demuestra el teorema.

- **Caso [CAS]**

Suponiendo que $(\theta, v) \in \mathcal{E}\left[\!\left[\textbf{case } x \textbf{ of } \overline{cls_i}^n\right]\!\right]$, existe un $k \in \{1..n\}$ y $\overline{v_j}$ tales que $\left(\theta\left[\overline{x_j/v_j}\right], v\right) \in \mathcal{E}[\![e']\!]$ y $\theta\left[\overline{x_j/v_j}\right] \in matches(\theta, \theta(x), cls_k)$ donde cls_k tiene la forma p **when** $e_g \mapsto e'$ y $\{\overline{x_j}\} = vars(p)$. Es más, ya que hemos aplicado la regla [CAS], sabemos que $\Gamma(x) = \alpha$ y que $\Gamma \Vdash_\alpha cls_k : \rho_k$ para un ρ_k. Desplegamos las definiciones de $matches(\theta, \theta(x), cls_k)$ y $\Gamma \Vdash_\alpha cls_k : \rho_k$ para obtener:

$$\left(\theta\left[\overline{x_j/v_j}\right], \theta(x)\right) \in \mathcal{E}[\![p]\!] \qquad \Gamma\left[\overline{x_j : \alpha_j}\right] \vdash p : \tau_p \tag{4.18}$$

$$\left(\theta\left[\overline{x_j/v_j}\right], \texttt{'true'}\right) \in \mathcal{E}[\![e_g]\!] \quad \Gamma\left[\overline{x_j : \alpha_j} \mid \tau_p \Leftarrow \alpha\right] \vdash e_g : \overline{\left\langle \tau'_j ; \Gamma'_j \right\rangle}^m \tag{4.19}$$

$$\left(\theta\left[\overline{x_j/v_j}\right], v\right) \in \mathcal{E}[\![e']\!] \qquad \Gamma'_j\left[\texttt{'true'} \subseteq \tau'_j\right] \vdash e : \rho'_j \quad \text{para cada } j \in \{1..n\} \tag{4.20}$$

Como sabemos que $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$, existe un π tal que $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$. Vamos a denotar con π' la instanciación $\pi\left[\overline{\alpha_j \mapsto \{v_j\}}\right]$. Como las variables $\overline{\alpha_j}$ son frescas, sabemos que $\theta \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma]\!]$ por el lema 1 y por ello $\theta\left[\overline{x_j/v_j}\right] \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma\left[\overline{x_j : \alpha_j}\right]]\!]$. Aparte de lo anterior, $(\theta(x), \pi') \in \mathcal{T}[\![\Gamma(x)]\!] = \mathcal{T}[\![\alpha]\!]$, así que $\pi'(\alpha) = \{\theta(x)\}$. Por lo tanto podemos usar lo anterior con (4.18) para aplicar el lema 7 y así obtener que $(\theta(x), \pi') \in \mathcal{T}[\![\tau_p]\!]$. Al saber que $\pi'(\alpha) = \pi(\alpha) = \{\theta(x)\}$ entonces se cumple que $\pi' \vDash \tau_p \Leftarrow \alpha$ y por consiguiente $\theta\left[\overline{x_j/v_j}\right] \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma\left[\overline{x_j : \alpha_j} \mid \tau_p \Leftarrow \alpha\right]]\!]$. Ahora podemos aplicar la hipótesis de inducción sobre la derivación de (4.19) para obtener que $(\theta\left[\overline{x_j/v_j}\right], \texttt{'true'}) \in$

$\mathcal{T}\left[\!\left[\overline{\left\langle \tau'_j;\Gamma'_j \right\rangle}^{m}\right]\!\right]$. Esto implica que existe un $l \in \{1..m\}$ tal que $(\theta\left[\overline{x_j/v_j}\right], \texttt{'true'}) \in \mathcal{T}\left[\!\left[\langle \tau'_l;\Gamma'_l\rangle\right]\!\right]$. También existe una instanciación π tal que $(\texttt{'true'},\pi) \in \mathcal{T}\left[\!\left[\tau'_l\right]\!\right]$ y $\theta \in \mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma'_l\right]\!\right]$. De lo primero se sigue que $\pi \vDash \texttt{'true'} \subseteq \tau'_l$ que podemos unir con lo segundo para obtener que $\theta\left[\overline{x_j/v_j}\right] \in \mathcal{T}\left[\!\left[\Gamma'_l\left[\texttt{'true'} \subseteq \tau'_l\right]\right]\!\right]$. Ahora podemos usar (4.20) para aplicar la hipótesis de inducción y obtener que $(\theta\left[\overline{x_j/v_j}\right], v) \in \mathcal{T}\left[\!\left[\rho'_l\right]\!\right]$. Aplicando el lema 8 obtendremos como resultado que $(\theta, v) \in \mathcal{T}\left[\!\left[\rho'_l \setminus \{\overline{x_i}\}\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\overline{\rho'_j \setminus \{\overline{x_i}\}}^{m}\right]\!\right] = \mathcal{T}\left[\!\left[\rho_k\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\overline{\rho_k}^{n}\right]\!\right]$, lo cual demuestra el teorema.

- **Caso [RCV]**

  Suponemos que $(\theta, v) \in \mathcal{E}\left[\!\left[\textbf{receive } \overline{cls_i}^{n} \textbf{ after } x_t \to e'\right]\!\right]$ con $\theta \in \mathcal{T}_{Env}\left[\!\left[\Gamma\right]\!\right]$. De acuerdo con la definición semántica de la expresión **receive**, tenemos dos posibilidades:

  - **Caso [RCV-1]:** Existe un $k \in \{1..n\}$ y unos valores $\overline{v_i}, v'$ tales que $\theta(x_t) \in \texttt{integer()} \cup \{\texttt{'infinity'}\}$, $\left(\theta\left[\overline{x_i/v_i}\right], v\right) \in \mathcal{E}\left[\!\left[e''\right]\!\right]$ y $\theta\left[\overline{x_i/v_i}\right] \in matches\left(\theta, v', cls_k\right)$, donde $\overline{x_i}$ son las variables que aparecen dentro del patrón de cls_k y e'_k es el cuerpo de la cláusula número k dentro de la expresión **receive** (es decir, cls_k).
  - **Caso [RCV-2]:** $\theta(x_t) \in \texttt{integer()}$ y $(\theta, v) \in \mathcal{E}\left[\!\left[e'\right]\!\right]$.

  Primero vamos a suponer [RCV-1]. Si $\theta(x_t) \in \texttt{integer()} \cup \{\texttt{'infinity'}\}$ entonces $\theta \in \mathcal{T}_{Env}\left[\!\left[\Gamma_t\right]\!\right]$, donde $\Gamma_t = [x_t : \texttt{integer()} \cup \texttt{'infinity'}]$, así que $\theta \in \mathcal{T}_{Env}\left[\!\left[\Gamma\right]\!\right] \cap \mathcal{T}_{Env}\left[\!\left[\Gamma_t\right]\!\right] = \mathcal{T}_{Env}\left[\!\left[\Gamma \sqcap \Gamma_t\right]\!\right]$. Equivalentemente, θ pertenece a $\mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma \sqcap \Gamma_t\right]\!\right]$ para una instanciación π. Ahora supondremos que la cláusula escogida cls_k tiene la forma p **when** $e_g \to e''$ y $\overline{x_i}$ son las variables que aparecen dentro de p. Por la definición de $matches\left(\theta, v', cls_k\right)$ se cumple que $\left(\theta\left[\overline{x_i/v_i}\right], v'\right) \in \mathcal{E}\left[\!\left[p\right]\!\right]$. Vamos a denotar con π' la instanciación $\pi\left[\alpha \mapsto \{v'\}, \overline{\alpha_i \mapsto \{v_i\}}\right]$, donde α es una variable fresca que ha sido usada en la aplicación de la regla [RCV], y $\overline{\alpha_i}$ son las variables frescas usadas en la regla [CLS] que debe haber sido aplicada para demostrar que $\Gamma \sqcap \Gamma_t \Vdash_{\alpha} cls_k : \rho_k$. Como π y π' se diferencian únicamente en dichas variables frescas, con el lema 1 obtenemos que $\theta \in \mathcal{T}^{\pi'}_{Env}\left[\!\left[\Gamma \sqcap \Gamma_t\right]\!\right]$. De hecho, si extendemos el entorno también tenemos que $\theta\left[\overline{x_i/v_i}\right]$ está contenido dentro de $\mathcal{T}^{\pi'}_{Env}\left[\!\left[(\Gamma \sqcap \Gamma_t)\left[\overline{x_i : \alpha_i}\right]\right]\!\right]$. Podemos entonces aplicar el lema 7 y proceder igual que con el caso de [CAS] para demostrar el teorema.

  Ahora suponemos [RCV-2]. Como $\theta(x_t) \in \texttt{integer()}$ obtenemos que $\theta \in \mathcal{T}_{Env}\left[\!\left[\Gamma'_t\right]\!\right]$, donde $\Gamma'_t = [x_t : \texttt{integer()}]$ y por ello $\theta \in \mathcal{T}_{Env}\left[\!\left[\Gamma \sqcap \Gamma'_t\right]\!\right]$. El teorema prosigue con la aplicación de la hipótesis de inducción a la derivación de $\Gamma \sqcap \Gamma'_t \vdash e' : \rho$.

- **Caso [LRC]**

  En este caso e tiene la forma **letrec** $\overline{x_i = f_i}^{n}$ **in** e'. Sin pérdida de generalidad vamos a suponer que $n = 1$, es decir, que la expresión **letrec** solo va a definir una asignación. La extensión a dos o más asignaciones es simple de demostrar. En particular, suponemos que **letrec** $x = f$ **in** e', donde f es una expresión de la forma **fun**$(\overline{x_i}^{m}) \to e''$. Según uno de los supuestos de la regla [LRC], tenemos que $\Gamma\left[x : \tau'\right] \vdash f : \langle \tau'; \Gamma\left[x : \tau'\right]\rangle$ para un τ'.

Vamos a definir la función $F_\theta : \mathbf{DVal} \to \mathbf{DVal}$ tal que para cada $v \in \mathbf{DVal}$,

$$F_\theta(v) = v' \quad \text{donde } \{v\} = \left\{v' \mid (\theta\,[x/v]\,, v') \in \mathscr{E}\,[\![f]\!]\right\}$$

Vamos a definir ahora la secuencia $(v_k)_{k\in\mathbb{N}}$ como:

$$\begin{array}{lcll} v_0 & = & \emptyset & \\ v_k & = & F_\theta(v_{k-1}) & \text{para cada } k > 0 \end{array}$$

Es fácil mostrar que la semántica es monótona en las funciones que aparecen en una sustitución fija θ. Es decir, si $\left(\theta\,[x/v_1]\,, v_1'\right), \left(\theta\,[x/v_2]\,, v_2'\right) \in \mathscr{E}\,[\![f]\!]$ y $v_1 \subseteq v_2$, entonces $v_1' \subseteq v_2'$. Por lo tanto, la secuencia $(v_k)_{\Bbbk\in\mathbb{N}}$ es, de hecho, una cadena ascendente: $v_0 \subseteq v_1 \subseteq v_2 \subseteq \ldots$. Vamos a demostrar que, para cada $k \in \mathbb{N}$, la sustitución $\theta\,[x/v_k]$ pertenece a $\mathscr{T}_{Env}\,[\![\Gamma\,[x:\tau']]\!]$. Procedemos ahora por inducción sobre k:

- **Caso base** ($k = 0$). Como f es una λ-abstracción, la regla [ABS] debe haber sido usada en algún lugar del árbol de derivación de $\Gamma\,[x:\tau'] \vdash f : \langle \tau'; \Gamma\,[x:\tau']\rangle$. Por lo tanto, la semántica de τ' contiene el grafo vacío $\emptyset$, y por ello $\theta\,[x/\emptyset] \in \mathscr{T}_{Env}\,[\![\Gamma\,[x:\tau']]\!]$.
- **Paso inductivo** ($k > 0$). Suponemos que $\theta\,[x/v_k] \in \mathscr{T}_{Env}\,[\![\Gamma_0]\!]$. Como $(\theta\,[x/v_k]\,, v_{k+1}) \in \mathscr{E}\,[\![f]\!]$ y $\Gamma\,[x:\tau'] \vdash f : \langle \tau'; \Gamma\,[x:\tau']\rangle$, obtenemos por hipótesis de inducción sobre esta derivación que $(\theta\,[x/v_k]\,, v_{k+1}) \in \mathscr{T}\,[\![\langle \tau'; \Gamma\,[x:\tau']\rangle]\!]$. Esto implica la existencia de una instanciación π tal que $\theta \in \mathscr{T}^{\pi}_{Env}\,[\![\Gamma\,[x:\tau']]\!]$ y $(v_{k+1}, \pi) \in \mathscr{T}\,[\![\tau']\!]$, así que $\theta\,[x/v_{k+1}] \in \mathscr{T}^{\pi}_{Env}\,[\![\Gamma\,[x:\tau']]\!] \subseteq \mathscr{T}_{Env}\,[\![\Gamma\,[x:\tau']]\!]$.

Como consecuencia de esto último, se puede demostrar que $\theta\left[x/\bigcup_{k=1}^{\infty} v_k\right] \in \mathscr{T}_{Env}\,[\![\Gamma\,[x:\tau']]\!]$. Con el teorema del punto fijo de Tarski, obtendremos que $lfp\, F_\theta = \bigcup_{k=1}^{\infty} v_k$, así que $\theta\left[x/lfp\, F_\theta\right] \in \mathscr{T}_{Env}\,[\![\Gamma\,[x:\tau']]\!]$. Por el otro lado, tenemos que $\left(\theta\left[x/lfp\, F_\theta\right], v\right) \in \mathscr{E}\,[\![e']\!]$, por lo que aplicaremos la hipótesis de inducción sobre la derivación de $\Gamma\,[x:\tau'] \vdash e' : \rho$ para obtener que $\left(\theta\left[x/lfp\, F_\theta\right], v\right) \in \mathscr{T}\,[\![\rho]\!]$, a lo que aplicaremos el lema 8 para obtener $(\theta, v) \in \mathscr{T}\,[\![\rho \setminus \{x\}]\!]$ demostrando el teorema.

□

Por último, y como ya sucedía en el caso monomórfico, del teorema anterior es fácil deducir el siguiente corolario donde se muestra que, en el caso particular de que nuestra expresión e sea cerrada, nuestras reglas derivan *success types* en el sentido de la definición 1.

Colorario 2. *Para cualquier expresión cerrada e y un tipo anotado ρ tal que $[\,] \vdash e : \overline{\langle \tau_i; [\,]\rangle}^n$, entonces la unión de $\tau_1, \ldots, \tau_n$ es un* success type, *es decir $\mathscr{E}\,[\![e]\!]|_2 \subseteq \mathscr{T}\,[\![\tau_1 \cup \ldots \cup \tau_n]\!]|_2$.*

Capítulo 5

Inferencia de success types polimórficos

En el capítulo 4 presentamos el sistema para derivar tipos polimórficos que formulamos para nuestra investigación y demostramos que, de acuerdo con los planteamientos de esta tesis, son en efecto *success types*. Sobre este último hemos desarrollado un algoritmo de inferencia que genera y transforma *success types* polimórficos que pueden ser derivados con el sistema del capítulo anterior. Por ello, a lo largo del siguiente capítulo vamos a introducir al lector en nuestro algoritmo de inferencia de *success types* polimórficos, explicando paso a paso su funcionamiento y formalización.

En la sección 5.1 presentaremos los cambios realizados sobre la sintaxis y la semántica de los tipos polimórficos para adecuarlos a la inferencia. En la sección 5.2 presentaremos las reglas de generación de restricciones que hacen falta para inferir un tipo polimórfico en el sistema. En la sección 5.3 presentaremos una colección de operaciones necesarias para poder manipular y simplificar tipos. En la sección 5.4 mostraremos cómo normalizar los tipos resultantes del proceso de generación de restricciones. En la sección 5.5 detallaremos cómo se transforman los tipos generados para funciones no recursivas, de cara a obtener tipos simplificados. En la sección 5.6 extenderemos estas técnicas para inferir tipos de funciones recursivas. En la sección 5.7 mostraremos ejemplos de inferencia de tipos usando las reglas expuestas anteriormente. También mostraremos los tipos inferidos por el algoritmo cuando se aplica a funciones básicas sobre números y listas, incluyendo ejemplos que muestran que, conforme a lo previsto, el polimorfismo permite obtener tipos más precisos y, en consecuencia, detectar más casos de error (que corresponden al *success type* `none()`). En la sección 5.8 detallaremos algunos aspectos relativos a la precisión de los *success types* obtenidos con la inferencia. Por último en la sección 5.9 presentaremos algunos detalles relevantes sobre la implementación de la herramienta que pone en práctica el algoritmo de inferencia.

$$\rho ::= \langle \tau ; \Gamma \rangle \mid \rho ; \rho \qquad \Gamma ::= \left[\overline{x_i : \alpha_{x_i}} \mid \overline{\varphi_j} \right]$$
$$C ::= \{\overline{\varphi_i}\} \qquad \varphi ::= \alpha \subseteq \tau \mid c \subseteq \tau \mid \tau \Leftarrow \alpha$$
$$\tau ::= \widetilde{\tau} \mid \bigcup_{i=1}^{n} \widetilde{\tau}_i \textbf{ when } C_i \qquad \sigma ::= \forall \overline{\alpha_j}.\left(\overline{\tau_i}\right) \to \tau \textbf{ when } C$$
$$\widetilde{\tau} ::= \texttt{none()} \mid \texttt{any()} \mid B \mid c \mid \alpha \mid \{\overline{\tau_i}\} \mid \texttt{nelist}(\tau_1 \textbf{ when } C, \tau_2) \mid \bigsqcup_{i=1}^{n} \sigma_i$$
donde $B \in \mathbb{B} c \in \textbf{DVal} \alpha \in \textbf{TypeVar}$

Figura 5.1: Sintaxis normalizada de los tipos

5.1. Adaptando los tipos para el algoritmo de inferencia

En esta sección vamos a introducir la sintaxis normalizada de los tipos para poder trabajar con el algoritmo de inferencia, así como las nuevas restricciones que necesitaremos para el sistema. También expondremos los cambios en la semántica que han sido necesarios para adaptar el sistema de tipos polimórfico al algoritmo de inferencia.

5.1.1. Sintaxis normalizada de los tipos

El primer paso a considerar de cara a la inferencia es el siguiente: a partir de los tipos polimórficos definidos en el capítulo anterior (figuras 4.1 y 4.4) y con el objeto de limitar el número de casos con los que trabajar durante el proceso de inferencia, establecemos una forma normal para los tipos de **Type**, de acuerdo con la nueva sintaxis normalizada de la figura 5.1. Al conjunto de tipos en forma normal lo denotamos por $\textbf{Type}^\star$ que es por tanto un subconjunto de **Type**. De forma análoga a los tipos, establecemos una forma normal para los tipos anotados ρ con $\textbf{TypeAn}^\star$, subconjunto de **TypeAn**, y para los entornos Γ con $\textbf{Env}^\star$, subconjunto de **Env**.

La principal diferencia entre la sintaxis normalizada y la expuesta en la sección 4.1 consiste en restringir dónde podemos usar los tipos condicionales, que tienen la forma τ **when** C. En esta forma normalizada solo permitimos adjuntar restricciones en las siguientes estructuras: en las componentes de los tipos unión, en el tipo que representa los elementos de las listas no vacías, y en el tipo que representa el resultado en los tipos funcionales. En la mayoría de los casos no perderemos generalidad restringiendo los tipos a esta forma normal, ya que—en muchos de los casos—podremos elevar los sufijos **when** C a posiciones superiores del árbol sintáctico correspondiente del tipo hasta alcanzar las formas normales que hemos indicado anteriormente, obteniendo así tipos equivalentes. Por ejemplo, se cumple que $\{\tau_1 \textbf{ when } C, \tau_2\}$ es equivalente a $\{\tau_1, \tau_2\}$ **when** C para τ_1, τ_2 y

C cualesquiera. Sin embargo, esta forma normal no es totalmente igual de expresiva que la sintaxis sin normalizar. En particular, la forma normal prohíbe el uso de tipos con **when** dentro de restricciones. Por ejemplo, el conjunto de restricciones $\{\alpha \subseteq \tau \textbf{ when } C\}$ tendría que normalizarse, obteniendo como resultado el conjunto $\{\alpha \subseteq \tau\} \cup C$, que no es semánticamente equivalente al anterior, porque suponiendo $\pi = [\alpha \mapsto \{1\}, \beta \mapsto \{1, a\}]$, se tiene que $\pi \vDash \{\alpha \subseteq \beta \textbf{ when } \beta \subseteq \texttt{integer()}\}$, pero $\pi \nvDash \{\alpha \subseteq \beta, \beta \subseteq \texttt{integer()}\}$. Las consecuencias de esta pérdida de expresividad no son tan preocupantes como podría parecer en un principio, ya que el algoritmo de inferencia no genera, por sí solo, tipos **when** dentro de restricciones. Los tipos de esta clase tendrían que haber sido introducidos manualmente por el usuario y, en ese caso, tendrían que ser transformados, asumiendo la consiguiente pérdida de precisión semántica. Otra posibilidad para evitar este tipo de situaciones consiste en limitar la sintaxis de los tipos que puede describir el usuario, para que este pueda introducir solamente tipos normalizados.

En cuanto a los tipos anotados, tenemos una diferencia importante en los entornos, ya que supondremos que para toda $x \in \textbf{Var}$ se cumple que $\Gamma(x) = \alpha_x$, siendo α_x una variable de tipo específica para x, y distinta de las variables de tipo correspondientes al resto de variables de programa. Esto no supone ningún problema, porque se supone que al analizar una expresión no habrá colisión de variables, y en caso de haberlas, se pueden renombrar para evitarlo. Además, cualquier entorno en el que una variable x esté ligada a un tipo τ que no sea una variable de tipo, puede convertirse en un entorno equivalente en el que x esté ligada a α_x, añadiendo la restricción $\tau \Leftarrow \alpha_x$. Todo entorno $\Gamma \in \textbf{Env}^\star$ cumplirá la condición que acabamos de explicar. Teniendo esto en cuenta, usaremos en las reglas de las siguientes secciones la notación $\langle \tau; C \rangle$ para representar tipos anotados, que pertenecen a $\textbf{TypeAn}^\star$, en vez de usar la notación $\langle \tau; \Gamma \rangle$, ya que la asignación de variables de programa x a sus correspondientes variables de tipo α_x se considera implícita.

Una de las clases de restricciones que no vamos a utilizar en el algoritmo de inferencia son las de la forma $\alpha_f \subseteq \bigsqcup_{j=1}^{m} \left(\forall \overline{\alpha_{j,i}}. \left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)$. Es decir, aquellas restricciones de inclusión que tienen un tipo funcional (sobrecargado o no) en el lado derecho. En su lugar, las restricciones que podemos usar son de la forma $\bigsqcup_{j=1}^{m} \left(\forall \overline{\alpha_{j,i}}. \left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \Leftarrow \alpha_f$. Normalmente, esta última es la clase de restricciones que se desean cuando introducimos el tipo de una función en un entorno, ya sea un tipo inferido, o especificado por el usuario.

Por convenio, en las reglas del algoritmo, las apariciones de variables de tipo δ corresponden a variables frescas generadas para esa regla en concreto, mientras que las variables de tipo α van acompañadas del nombre de la variable de programa a la que representan (por ejemplo: α_X), las variables de tipo δ se identifican con un valor numérico que reciben al ser generadas (por ejemplo: δ_1).

Usaremos la notación $\alpha = \beta$, como azúcar sintáctico, para expresar la aparición de las restricciones $\alpha \subseteq \beta$ y $\beta \subseteq \alpha$ dentro de un mismo conjunto. No usaremos esta notación para $\alpha \Leftarrow \beta$ o $\beta \Leftarrow \alpha$, porque la propia semántica de una restricción de encaje equivale a la igualdad entre instancias para cada variable de tipo en la restricción. Por tanto $\alpha \Leftarrow \beta$ equivale a $\beta \Leftarrow \alpha$.

5.1.2. Variables de tipo plurales y singulares

Un concepto importante para el algoritmo de inferencia es entender cuándo la semántica de una variable de tipo es plural o singular. Decimos que una variable α tiene semántica plural en un tipo τ si está contenida dentro del tipo τ_1 de los elementos de una lista no vacía `nelist(`τ_1`,`τ_2`)` o está contenida en un tipo funcional. En caso contrario, decimos que tiene semántica singular. Si α tiene semántica singular en τ, se cumple que para todo $(\upsilon,\pi) \in \mathscr{T}[\![\tau]\!]$, $\pi(\alpha)$ tiene cardinalidad uno o cero.

5.1.3. Nuevas restricciones: aplicaciones simbólicas

Uno de los principales problemas a los que nos enfrentamos, a la hora de inferir los tipos de una expresión, es saber cómo se tiene que tipar una aplicación de función. Hay que tener en cuenta que en muchas ocasiones no se va a disponer del tipo exacto de la función que se está aplicando. Por ello hay que añadir una nueva clase de restricciones que representen dicha aplicación para resolver el tipo de sus argumentos y de su resultado cuando tengamos más información sobre el tipo de la función aplicada. Esta clase de restricciones que necesitamos recibe el nombre de *aplicación simbólica*. Las restricciones de esta nueva clase tienen la siguiente forma:

$$Z_\alpha\left(\overline{\alpha_i}^n\right) \Leftarrow \beta$$

Donde β sería el tipo que representa el resultado de la aplicación, $\overline{\alpha_i}^n$ las variables utilizadas como argumentos en la aplicación y α la variable de tipo asociada al tipo funcional a aplicar.

Como hemos añadido un nuevo tipo de restricciones a nuestra sintaxis, hemos de proporcionar la siguiente definición semántica para las mismas:

Definición 4. Dada una instanciación para variables de tipo π y una restricción de la forma $Z_\alpha\left(\overline{\alpha_i}^n\right) \Leftarrow \beta$, diremos que $\pi \vDash Z_\alpha\left(\overline{\alpha_i}^n\right) \Leftarrow \beta$ si y solo si para cada $f \in \pi(\alpha)$, $\upsilon_i \in \pi(\alpha_i)$ (donde $i \in \{1..n\}$), $\upsilon \in \pi(\beta)$, se cumple que $\left(\left(\overline{\upsilon_i}^n\right), \upsilon\right) \in f$.

De esta manera podemos incorporar las aplicaciones simbólicas como parte del conjunto $\mathbf{Type}^\star$ de los tipos normalizados y usarlo en las reglas del algoritmo de inferencia. Como consecuencia de esta definición, tendremos el siguiente lema:

Lema 10. *Sea una instanciación π tal que $\pi \vDash \bigsqcup_{j=1}^m \left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \Leftarrow \alpha$. Se cumple lo siguiente:*

$$\pi \vDash \left\{Z_\alpha\left(\overline{\beta_i}^n\right) \Leftarrow \beta\right\}$$
$$\Updownarrow$$
$$\forall\, j \in \{1..m\}.\pi_j \vDash \left(\left\{\overline{\tau_{j,i}\mu_j \Leftarrow \beta_i}^n, \tau_j\mu_j \Leftarrow \beta\right\} \cup \left\{\beta'\mu_j \subseteq \beta' \mid \beta' \in itv\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)\right\}\right)$$

donde $\mu_j = freshRenaming\left(itv\left(\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \cup \left\{\overline{\alpha_{j,i}}\right\}\right)$ y la instanciación $\pi_j \equiv \pi$ (módulo $\setminus rng\,\mu_j$), para todo $j \in \{1..m\}$.

Demostración. Siguiendo la definición 4 para las aplicaciones simbólicas, tenemos que $\left(\left(\overline{v_i}^n\right), v\right) \in \pi(\alpha)$, $v_i \in \pi(\beta_i)$, para cada $i \in \{1..n\}$, y $v \in \pi(\beta)$, además de existir un tipo τ_f tal que $(\pi(\alpha), \pi) \in \mathcal{T}\,[\![\tau_f]\!]$, por lo que $\tau_f = \bigsqcup_{j=1}^{m}\left(\forall\overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)$ y entonces $(\pi(\alpha), \pi) \in \mathcal{T}\left[\!\!\left[\bigsqcup_{j=1}^{m}\left(\forall\overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)\right]\!\!\right]$. Suponemos las sustituciones $\mu_j = \mathit{freshRenaming}\left(itv\left(\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \cup \{\overline{\alpha_{j,i}}\}\right)$, para todo $j \in \{1..m\}$, de modo que el rango de μ_j será el conjunto $\left\{\overline{\beta_i''}\right\}$ con las variables de tipo frescas que se van a introducir en el resultado final. Entonces, de forma análoga a la demostración de la regla [APP1] en el teorema 2, para cada $j \in \{1..m\}$, obtendremos las siguientes restricciones:

$$\begin{array}{rl} \forall i \in \{1..n\}\,.\,\pi_j \vDash \tau_{j,i}\mu_j \Leftarrow \beta_i & \wedge \quad \pi_j \vDash \tau_j\mu_j \Leftarrow \beta \\ \wedge \quad \forall \beta' \in itv\left(\forall\overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)\,.\,\pi_j \vDash \beta'\mu_j \subseteq \beta' & \end{array}$$

donde π_j es una instanciación tal que $\pi_j \equiv \pi$ (módulo $\setminus \left\{\overline{\beta_i''}\right\}$). La unión de todas las restricciones en un solo conjunto da el resultado final. □

5.1.4. Tipos ∪-normalizados

Antes de explicar qué son los tipos ∪-normalizados, necesitamos repasar la semántica del sistema de tipos polimórfico que vimos en la sección 4.3 del capítulo anterior. Partiendo de la semántica de los tipos unión, si la generalizamos para n componentes obtendremos la siguiente definición:

$$\begin{array}{l} \mathcal{T}\left[\!\!\left[\bigcup_{i=1}^{n} \widetilde{\tau}_i \textbf{ when } C_i\right]\!\!\right] = \bigcup_{i=1}^{n} (\mathcal{T}\,[\![\widetilde{\tau}_i \textbf{ when } C_i]\!] \setminus E_i) \\ \qquad \text{donde } E_i = \left(\bigcup\left\{itv\left(\widetilde{\tau}_j \textbf{ when } C_j\right) \mid j \in \{1..n\}, j \neq i\right\}\right) \setminus ftv(\widetilde{\tau}_i \textbf{ when } C_i) \end{array} \tag{5.1}$$

Recordemos que la notación $\mathcal{T}\,[\![\tau]\!] \setminus E_i$ significa $\{(v, \pi\setminus E_i) \mid (v,\pi) \in \mathcal{T}\,[\![\tau]\!]\}$, donde $\pi\setminus E_i$ es una instanciación π' que cumple $\pi' \equiv [\,]$ (módulo E_i) y $\pi' \equiv \pi$ (módulo $\setminus E_i$).

Si analizamos la semántica de los tipos unión que acabamos de generalizar, tener que borrar de las instanciaciones la información dada por el conjunto E_i es bastante laborioso a la hora de razonar sobre tipos unión, y a la hora de manipularlos. No obstante, este borrado puede representarse explícitamente mediante restricciones de la forma `none()` $\Leftarrow \alpha$. Por ejemplo, los siguientes tipos son equivalentes entre sí:

$$\texttt{[]} \cup \texttt{nelist(}\alpha\texttt{,[])} \quad \equiv \quad (\texttt{[]} \textbf{ when } \texttt{none()} \Leftarrow \alpha) \cup \texttt{nelist(}\alpha\texttt{,[])}$$

Vemos que en la primera rama tenemos el tipo `[]` sin variables de tipo, mientras que en la segunda rama de la unión tenemos una lista no vacía propia de tipo α. Como α está en una posición instanciable, tendrá que ser eliminada de las instancias que obtengamos de `[]`, por lo que al introducir la restricción `none()` $\Leftarrow \alpha$ hacemos explícita la operación de eliminación.

Decimos que un tipo está ∪-normalizado si cada rama—dentro de la unión—indica explícitamente las variables E_i que han de ser eliminadas. Esto lo formalizamos con la siguiente definición:

Definición 5. Un tipo unión $\bigcup_{i=1}^{n} \widetilde{\tau}_i$ **when** C_i es $\cup$-normal si y solo si E_i, tal como está definido en (5.1), es vacío para cada $i \in \{1..n\}$.

Teniendo esta definición, vamos a necesitar ciertas proposiciones que podremos usar más adelante para razonar sobre la corrección del algoritmo de inferencia. La primera proposición establece que la semántica de un tipo unión $\cup$-normalizado es igual a la unión de las semánticas de aquellos tipos que componen la unión, sin necesidad de eliminar las variables en cada E_i de las instanciaciones correspondientes.

Proposición 12. *Supongamos un tipo unión* $\bigcup_{i=1}^{n} \widetilde{\tau}_i$ **when** C_i *que es* $\cup$*-normal, entonces:*

$$\mathscr{T}\left[\!\!\left[\bigcup_{i=1}^{n} \widetilde{\tau}_i \textbf{ when } C_i\right]\!\!\right] = \bigcup_{i=1}^{n} \mathscr{T}\left[\!\!\left[\widetilde{\tau}_i \textbf{ when } C_i\right]\!\!\right]$$

Demostración. Puesto que todo $E_i = \emptyset$ para cada $i \in \{1..n\}$, se cumple que $\pi = \pi \backslash E_i$ para toda instanciación de tipos π. Por lo tanto:

$$\begin{aligned}
\mathscr{T}\left[\!\!\left[\bigcup_{i=1}^{n} \widetilde{\tau}_i \textbf{ when } C_i\right]\!\!\right] &= \bigcup_{i=1}^{n} \left(\mathscr{T}\left[\!\!\left[\widetilde{\tau}_i \textbf{ when } C_i\right]\!\!\right] \backslash E_i\right) \\
&= \bigcup_{i=1}^{n} \{(v, \pi \backslash E_i) \mid (v,\pi) \in \mathscr{T}\left[\!\!\left[\widetilde{\tau}_i \textbf{ when } C_i\right]\!\!\right]\} \\
&= \bigcup_{i=1}^{n} \{(v,\pi) \mid (v,\pi) \in \mathscr{T}\left[\!\!\left[\widetilde{\tau}_i \textbf{ when } C_i\right]\!\!\right]\} \\
&= \bigcup_{i=1}^{n} \mathscr{T}\left[\!\!\left[\widetilde{\tau}_i \textbf{ when } C_i\right]\!\!\right]
\end{aligned}$$

□

Dado un tipo unión $\tau = \bigcup_{i=1}^{n} \widetilde{\tau}_i$ **when** C_i y los conjuntos de variables E_i, tal como se definen en (5.1), denotamos con $\widehat{\tau}$ la $\cup$-*normalización* de τ, que está definida de la siguiente manera:

$$\widehat{\tau} = \bigcup_{i=1}^{n} \widetilde{\tau}_i \textbf{ when } C_i \cup \{\mathsf{none}() \Leftarrow \alpha \mid \alpha \in E_i\}$$

Con esta definición vamos a demostrar a continuación que efectivamente $\widehat{\tau}$ es $\cup$-normal.

Proposición 13. *Suponiendo un tipo* τ*, se cumple que* $\widehat{\tau}$ *es un tipo* $\cup$*-normal.*

Demostración. Para cada $i \in \{1..n\}$, vamos a definir $\widehat{E_i}$ de la siguiente manera:

$$\begin{aligned}
\widehat{E_i} = \Big(\bigcup \big\{itv\big(\widetilde{\tau}_j \textbf{ when } C_j \cup \{\mathsf{none}() \Leftarrow \alpha \mid \alpha \in E_j\}\big) \mid j \in \{1..n\}, j \neq i\big\}\Big) \\
\backslash ftv(\widetilde{\tau}_i \textbf{ when } C_i \cup \{\mathsf{none}() \Leftarrow \alpha \mid \alpha \in E_i\})
\end{aligned}$$

Tenemos que demostrar que $\widehat{E_i} = \emptyset$ para cada i. Para toda $i, j \in \{1..n\}, j \neq i$ tenemos que demostrar que:

$$itv\big(\widetilde{\tau}_j \textbf{ when } C_j \cup \big\{\mathsf{none}\,() \Leftarrow \alpha \mid \alpha \in E_j\big\}\big) \subseteq ftv(\widetilde{\tau}_i \textbf{ when } C_i \cup \{\mathsf{none}\,() \Leftarrow \alpha \mid \alpha \in E_i\})$$

Supongamos una variable de tipo $\alpha \in itv\big(\widetilde{\tau}_j \textbf{ when } C_j \cup \big\{\mathsf{none}\,() \Leftarrow \alpha \mid \alpha \in E_j\big\}\big)$. Como $itv\big(\mathsf{none}\,() \Leftarrow \beta\big)$ es vacío, para toda variable de tipo β, obtenemos que $\alpha \in itv\big(\widetilde{\tau}_j \textbf{ when } C_j\big)$. Si $\alpha \in itv(\widetilde{\tau}_i \textbf{ when } C_i)$, tenemos también que $\alpha \in ftv(\widetilde{\tau}_i \textbf{ when } C_i \cup \{\mathsf{none}\,() \Leftarrow \alpha \mid \alpha \in E_i\})$. Si no, $\alpha \in E_i$, y por ello:

$$\alpha \in ftv(\mathsf{none}\,() \Leftarrow \alpha) \subseteq ftv(\widetilde{\tau}_i \textbf{ when } C_i \cup \{\mathsf{none}\,() \Leftarrow \alpha \mid \alpha \in E_i\})$$

□

A continuación demostraremos que la semántica de un tipo y la de su versión $\cup$-normal son iguales:

Proposición 14. *Dado un tipo τ, si $\hat{\tau}$ es su $\cup$-normalización, entonces se cumple que $\mathscr{T}[\![\tau]\!] = \mathscr{T}[\![\hat{\tau}]\!]$.*

Demostración. Empezaremos demostrando que $\mathscr{T}[\![\tau]\!] \subseteq \mathscr{T}[\![\hat{\tau}]\!]$. Supongamos un par $(v,\pi) \in \mathscr{T}[\![\tau]\!]$. Existe un $i \in \{1..n\}$ tal que $(v,\pi') \in \mathscr{T}[\![\widetilde{\tau}_i \textbf{ when } C_i]\!]$ donde $\pi = \pi' \backslash E_i$. Esto último implica que $\pi \equiv \pi'$ (módulo $\backslash E_i$). Por la definición de E_i, se cumple que $E_i \cap ftv(\widetilde{\tau}_i \textbf{ when } C_i) = \emptyset$, lo que implica que $ftv(\widetilde{\tau}_i \textbf{ when } C_i) \subseteq \backslash E_i$. Por lo tanto, $\pi \equiv \pi'$ (módulo $ftv(\widetilde{\tau}_i \textbf{ when } C_i)$). Podemos aplicar el lema 1 para transformar $(v,\pi') \in \mathscr{T}[\![\widetilde{\tau}_i \textbf{ when } C_i]\!]$ en $(v,\pi) \in \mathscr{T}[\![\widetilde{\tau}_i \textbf{ when } C_i]\!]$. Es más, como $\pi \equiv [\,]$ (módulo E_i) se cumple que $\pi(\alpha) = \emptyset$ para cualquier $\alpha \in E_i$. Por lo que $\pi \vDash \mathsf{none}\,() \Leftarrow \alpha$ para cada $\alpha \in E_i$. Por consiguiente:

$$(v,\pi) \in \mathscr{T}[\![\widetilde{\tau}_i \textbf{ when } C_i \cup \{\mathsf{none}\,() \Leftarrow \alpha \mid \alpha \in E_i\}]\!] \subseteq \mathscr{T}[\![\hat{\tau}]\!]$$

donde el último paso se obtiene del hecho de que $\hat{\tau}$ es $\cup$-normal.

Ahora demostraremos que $\mathscr{T}[\![\hat{\tau}]\!] \subseteq \mathscr{T}[\![\tau]\!]$. Supongamos un par $(v,\pi) \in \mathscr{T}[\![\hat{\tau}]\!]$. Como el tipo $\hat{\tau}$ es $\cup$-normal, existe un $i \in \{1..n\}$ tal que $(v,\pi) \in \mathscr{T}[\![\widetilde{\tau}_i \textbf{ when } C_i \cup \{\mathsf{none}\,() \Leftarrow \alpha \mid \alpha \in E_i\}]\!]$. Resulta que $(v,\pi) \in \mathscr{T}[\![\widetilde{\tau}_i \textbf{ when } C_i]\!]$ y $\pi \vDash \mathsf{none}\,() \Leftarrow \alpha$ para cada $\alpha \in E_i$. Esto último implica que $\pi(\alpha) = \emptyset$ para cada $\alpha \in E_i$ o, de forma equivalente, $\pi \equiv [\,]$ (módulo E_i). Por lo tanto, $\pi = \pi \backslash E_i$, así que $(v,\pi) \in \mathscr{T}[\![\widetilde{\tau}_i \textbf{ when } C_i]\!]$ implica $(v,\pi\backslash E_i) \in \mathscr{T}[\![\tau]\!]$ de lo que obtenemos que $(v,\pi) \in \mathscr{T}[\![\tau]\!]$. □

Ahora vamos a suponer dos tipos unión τ_1 y τ_2 definidos como sigue:

$$\tau_1 = \bigcup_{i=1}^{n} \widetilde{\tau}_i \textbf{ when } C_i \qquad \tau_2 = \bigcup_{j=1}^{m} \widetilde{\tau}'_j \textbf{ when } C'_j$$

Su ínfimo $\tau_1 \sqcap \tau_2$ se define de la siguiente manera:

$$\tau_1 \sqcap \tau_2 = \bigcup_{\substack{i \in \{1..n\} \\ j \in \{1..m\}}} \widetilde{\tau}_{i,j} \textbf{ when } C_i \cup C'_j \cup C_{i,j}$$

En la sección 5.3.3 se dará una definición del operador $\sqcap$ para tipos $\widetilde{\tau}$.

Proposición 15. *Si τ_1 y τ_2 son $\cup$-normales, también lo será $\tau_1 \sqcap \tau_2$ siempre que se cumplan las siguientes condiciones para cualquier $i \in \{1..n\}$, $j \in \{1..m\}$:*

1. $itv\left(\widetilde{\tau}_i \sqcap \widetilde{\tau}'_j\right) \subseteq itv(\widetilde{\tau}_i) \cup itv\left(\widetilde{\tau}'_j\right)$
2. $ftv\left(\widetilde{\tau}_i \sqcap \widetilde{\tau}'_j\right) \supseteq ftv(\widetilde{\tau}_i) \cup ftv\left(\widetilde{\tau}'_j\right)$

Demostración. Para cada $i \in \{1..n\}$, $j \in \{1..m\}$ vamos a definir el conjunto $E_{i,j}$ de la siguiente manera:

$$E_{i,j} = \bigcup \{itv(\widetilde{\tau}_{k,l} \textbf{ when } C_k \cup C'_l \cup C_{k,l}) \mid k \in \{1..n\}, l \in \{1..m\}, k \neq i, l \neq j\} \setminus ftv\left(\widetilde{\tau}_{i,j} \textbf{ when } C_i \cup C'_j \cup C_{i,j}\right)$$

Vamos a demostrar que $E_{i,j} = \emptyset$ por reducción al absurdo. Supongamos que existe un $k \in \{1..n\}$ y un $l \in \{1..m\}$ tales que $k \neq i, l \neq j$, y además:

$$\alpha \in itv(\widetilde{\tau}_{k,l} \textbf{ when } C_k \cup C'_l \cup C_{k,l}) \wedge \alpha \notin ftv\left(\widetilde{\tau}_{i,j} \textbf{ when } C_i \cup C'_j \cup C_{i,j}\right)$$

Como $\alpha \notin ftv\left(\widetilde{\tau}_{i,j} \textbf{ when } C_i \cup C'_j \cup C_{i,j}\right)$, podemos asegurar que $\alpha \notin ftv(\widetilde{\tau}_i) \cup ftv\left(\widetilde{\tau}'_j\right) \cup ftv(C_i) \cup ftv\left(C'_j\right)$.

Si $\alpha \in itv(C_k)$, obtendremos que $\alpha \in itv(\widetilde{\tau}_k \textbf{ when } C_k) \setminus ftv(\widetilde{\tau}_i \textbf{ when } C_i)$, contradiciendo el hecho de que τ_1 es $\cup$-normal. Análogamente, si $\alpha \in itv(C'_l)$ tendremos que $\alpha \in itv(\widetilde{\tau}'_l \textbf{ when } C'_l) \setminus ftv\left(\widetilde{\tau}'_j \textbf{ when } C'_j\right)$ contradiciendo que τ_2 es $\cup$-normal.

El único caso restante es aquel en el que $\alpha \in itv(\widetilde{\tau}_{k,l} \textbf{ when } C_{k,l})$. Esto implica que $\alpha \in itv(\widetilde{\tau}_k) \cup itv(\widetilde{\tau}'_l)$. Si $\alpha \in itv(\widetilde{\tau}_k)$ obtenemos que $\alpha \in itv(\widetilde{\tau}_k \textbf{ when } C_k) \setminus ftv(\widetilde{\tau}_i \textbf{ when } C_i)$, contradiciendo que τ_1 es $\cup$-normal. De forma similar, si suponemos que $\alpha \in itv(\widetilde{\tau}'_l)$ nos conducirá a una contradicción con τ_2 siendo $\cup$-normal. □

Por último, con la siguiente proposición tendremos como resultado que la semántica del ínfimo entre tipos unión será la intersección de la semántica de cada uno, siempre que se cumpla esta misma propiedad para el ínfimo aplicado a las componentes de la unión.

Proposición 16. *Para todo par de tipos unión τ_1 y τ_2, se cumple que $\mathscr{T}[\![\tau_1 \sqcap \tau_2]\!] = \mathscr{T}[\![\tau_1]\!] \cap \mathscr{T}[\![\tau_2]\!]$ siempre que $\mathscr{T}\left[\!\left[\widetilde{\tau}_i \sqcap \widetilde{\tau}'_j\right]\!\right] = \mathscr{T}[\![\widetilde{\tau}_i]\!] \cap \mathscr{T}\left[\!\left[\widetilde{\tau}'_j\right]\!\right]$ para cada $i \in \{1..n\}$ y cada $j \in \{1..m\}$.*

Demostración. Para esta demostración usaremos la siguiente notación. Si C es un conjunto de res-

tricciones, $[\![C]\!]$ denota aquel conjunto de pares (v,π) tales que $v \in \mathbf{DVal}$ y $\pi \vDash C$.

$$
\begin{aligned}
\mathscr{T}\,[\![\tau_1 \sqcap \tau_2]\!] &= \mathscr{T}\left[\!\!\left[\bigcup_{\substack{i\in\{1..n\}\\ j\in\{1..m\}}} \widetilde{\tau}_{i,j} \textbf{ when } C_i \cup C'_j \cup C_{i,j} \right]\!\!\right] \\
&= \bigcup_{\substack{i\in\{1..n\}\\ j\in\{1..m\}}} \mathscr{T}\left[\!\!\left[\widetilde{\tau}_{i,j} \textbf{ when } C_i \cup C'_j \cup C_{i,j} \right]\!\!\right] && \tau_1 \sqcap \tau_2 \text{ es } \cup\text{-normal} \\
&= \bigcup_{\substack{i\in\{1..n\}\\ j\in\{1..m\}}} \mathscr{T}\left[\!\!\left[\widetilde{\tau}_i \sqcap \widetilde{\tau}'_j \textbf{ when } C_i \cup C'_j \right]\!\!\right] && \text{definición de } \widetilde{\tau}_{i,j} \textbf{ when } C_{i,j} \\
&= \bigcup_{\substack{i\in\{1..n\}\\ j\in\{1..m\}}} \left(\mathscr{T}\left[\!\!\left[\widetilde{\tau}_i \sqcap \widetilde{\tau}'_j \right]\!\!\right] \cap \left[\!\!\left[C_i \cup C'_j \right]\!\!\right]\right) \\
&= \bigcup_{\substack{i\in\{1..n\}\\ j\in\{1..m\}}} \left(\mathscr{T}\,[\![\widetilde{\tau}_i]\!] \cap \mathscr{T}\left[\!\!\left[\widetilde{\tau}'_j \right]\!\!\right] \cap [\![C_i]\!] \cap \left[\!\!\left[C'_j \right]\!\!\right]\right) && \text{suposición} \\
&= \bigcup_{\substack{i\in\{1..n\}\\ j\in\{1..m\}}} \left(\mathscr{T}\,[\![\widetilde{\tau}_i \textbf{ when } C_i]\!] \cap \mathscr{T}\left[\!\!\left[\widetilde{\tau}'_j \textbf{ when } C'_j \right]\!\!\right]\right) \\
&= \left(\bigcup_{i\in\{1..n\}} \mathscr{T}\,[\![\widetilde{\tau}_i \textbf{ when } C_i]\!]\right) \cap \left(\bigcup_{j\in\{1..m\}} \mathscr{T}\left[\!\!\left[\widetilde{\tau}'_j \textbf{ when } C'_j \right]\!\!\right]\right) && \cup \text{ distribuye respecto } \cap \\
&= \mathscr{T}\left[\!\!\left[\bigcup_{i\in\{1..n\}} \widetilde{\tau}_i \textbf{ when } C_i \right]\!\!\right] \cap \mathscr{T}\left[\!\!\left[\bigcup_{j\in\{1..m\}} \widetilde{\tau}'_j \textbf{ when } C'_j \right]\!\!\right] && \tau_1 \text{ y } \tau_2 \text{ son } \cup\text{-normales} \\
&= \mathscr{T}\,[\![\tau_1]\!] \cap \mathscr{T}\,[\![\tau_2]\!]
\end{aligned}
$$

□

5.2. Reglas de generación de restricciones

En la sección 4.5 vimos las reglas con las que podíamos derivar *success types* polimórficos para una expresión e. En estas reglas tenemos un entorno de entrada Γ que contiene información relativa a las variables libres de la expresión e, para obtener un tipo anotado final ρ. Para implementar un algoritmo de inferencia, estas reglas presentan un problema inicial al requerir en Γ los tipos de las funciones que van a ser aplicadas. Esto hace que, en las expresiones **letrec**, tipadas mediante la regla [LRC], uno tenga que conocer de antemano el tipo de las funciones definidas en la expresión **letrec** para poder introducirlas en un entorno con el que construir una derivación de tipos para dicha expresión. Sin embargo, cuando queremos inferir el tipo de una expresión **letrec**, no podemos saber de antemano cuáles serán los tipos de las funciones definidas en ella. Por ello necesitamos un conjunto de reglas nuevas que definen un algoritmo para generar un *success type* para una expresión, que son las que tenemos en la figura 5.2. En estas reglas tenemos juicios de la forma $e \vdash \rho$, donde $\rho \in \mathbf{Type}^\star$, que

$$\frac{}{c \vdash \langle c; \emptyset \rangle}\ [\text{CNS}] \qquad \frac{}{x \vdash \langle \alpha_x; \emptyset \rangle}\ [\text{VAR}] \qquad \frac{\forall i \in \{1..n\}: \quad e_i \vdash \rho_i}{\{\overline{e_i}^n\} \vdash \rho_1 \otimes \cdots \otimes \rho_n}\ [\text{TPL}]$$

$$\frac{e_1 \vdash \rho_1 \qquad e_2 \vdash \rho_2 \qquad \rho_1 \otimes \rho_2 = \overline{\langle \{\tau_i, \tau'_i\}; C_i \rangle}^n}{[e_1 \mid e_2] \vdash \overline{\langle \texttt{nelist}(\tau_i, \tau'_i); C_i \rangle}^n}\ [\text{LST}]$$

$$\frac{\begin{array}{c} e \vdash \overline{\langle \tau_j; C_j \rangle}^m \qquad \forall j \in \{1..m\}: \quad \tau'_j = (\overline{\alpha_{x_i}}^n) \to \tau_j \ \textbf{when}\ C_j \\ \{\overline{\alpha_{ij}}\} = ftv\left(\tau'_j\right) \setminus \{\alpha_x \mid x \in freevars(\textbf{fun}(\overline{x_i}^n) \to e)\} \qquad \sigma_j = \forall \overline{\alpha_{ij}}.\tau'_j \end{array}}{\textbf{fun}(\overline{x_i}^n) \to e \vdash \left\langle \bigsqcup_{j=1}^m \sigma_j; \emptyset \right\rangle}\ [\text{ABS}]$$

$$\frac{}{f(\overline{x_i}^n) \vdash \left\langle \delta; \left\{ Z_{\alpha_f}(\overline{\alpha_{x_i}}^n) \Leftarrow \delta \right\} \right\rangle}\ [\text{APP}] \qquad \frac{\forall i \in \{1..n\}: \quad cls_i \vdash_{\alpha_x} \rho_i}{\textbf{case}\ x\ \textbf{of}\ \overline{cls_i}^n \vdash \overline{\rho_i}^n}\ [\text{CAS}]$$

$$\frac{\begin{array}{c} \forall i \in \{1..n\}: \quad cls_i \vdash_\delta \rho_i \qquad \rho_1; \ldots; \rho_n = \overline{\langle \tau_j; C_j \rangle}^m \qquad e \vdash \overline{\langle \tau_k; C_k \rangle}^l \\ C_t = \{\alpha_{x_t} \subseteq \texttt{integer()} \cup \texttt{'infinity'}\} \qquad C'_t = \{\alpha_{x_t} \subseteq \texttt{integer()}\} \end{array}}{\textbf{receive}\ \overline{cls_i}^n\ \textbf{after}\ x_t \to e \vdash \overline{\langle \tau_j; C_j \cup C_t \rangle}^m; \overline{\langle \tau_k; C_k \cup C'_t \rangle}^l}\ [\text{RCV}]$$

$$\frac{e_1 \vdash \rho_1 \qquad e_2 \vdash \rho_2 \qquad \rho_1 \otimes \rho_2 = \overline{\langle \{\tau'_i, \tau_i\}; C_i \rangle}^n \qquad \mu = [\alpha_x / \delta]}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \vdash \overline{\langle \tau_i; C_i \cup \{\tau'_i \Leftarrow \alpha_x\} \rangle \mu}^n}\ [\text{LET}]$$

$$\frac{\forall i \in \{1..n\}: \quad f_i \vdash \langle \tau_i; \emptyset \rangle \qquad e \vdash \overline{\langle \tau_j; C_j \rangle}^m \qquad \mu = \left[\overline{\alpha_{x_i} / \delta_i}^n\right]}{\textbf{letrec}\ \overline{x_i = f_i}^n\ \textbf{in}\ e \vdash \overline{\langle \tau_j; C_j \cup \{\overline{\tau_i \Leftarrow \alpha_{x_i}}^n\} \rangle \mu}^m}\ [\text{LRC}]$$

Figura 5.2: Reglas de generación de restricciones para expresiones

es el conjunto de tipos anotados que pueden contener aplicaciones simbólicas en sus conjuntos de restricciones.

Antes de comentar cada regla, recordemos que se ha decidido usar como notación $\langle \tau; C \rangle$ en vez de $\langle \tau; \Gamma \rangle$ en cada rama de los tipos anotados resultantes, porque para toda $x \in \textbf{Var}$ se supone que $\Gamma(x) = \alpha_x$. En las reglas haremos uso del operador $\otimes$ visto en la sección 4.4, pero la definición adaptada a la notación usada por el algoritmo de inferencia sería de la siguiente forma:

$$\langle \tau_1; C_1 \rangle \otimes \cdots \otimes \langle \tau_n; C_n \rangle = \langle \{\tau_1, \ldots, \tau_n\}; C_1 \cup \cdots \cup C_n \rangle$$

La notación $\langle \tau; C \rangle$ tiene como efecto colateral que todos los entornos Γ que representan los respectivos conjuntos C_i estarían normalizados. Además, las reglas de generación de restricciones, cuando añaden una variable de tipo nueva, es una variable fresca, por lo que no hay colisión entre las variables de tipo de las diferentes ramas del tipo anotado, excluyendo las α_x asociadas implícitamente a cada variable de programa x. Para extender el operador $\otimes$ a los tipos anotados de la forma $\rho; \rho'$, se haría de

$$\frac{\rho_g \otimes \rho_e = \overline{\langle \{\tau'_i, \tau_i\}; C_i \rangle}^{n} \qquad \begin{array}{c} p \vdash \langle \tau_p; \emptyset \rangle \qquad e_g \vdash \rho_g \qquad e \vdash \rho_e \\ vars(p) = \{\overline{x_j}^{m}\} \qquad \mu = \left[\overline{\alpha_{x_j}/\delta_j}^{m}\right] \end{array}}{p \textbf{ when } e_g \to e \vdash_\alpha \overline{\langle \tau_i; C_i \cup \{\tau_p \Leftarrow \alpha, \texttt{'true'} \subseteq \tau'_i\} \rangle \mu}^{n}} \text{ [CLS]}$$

Figura 5.3: Regla de generación de restricciones para cláusulas

la misma forma que se vio en la sección 4.4.

Pasamos a describir las reglas de generación de restricciones. Las reglas [CNS] y [VAR] no añaden restricciones al resultado. Además, en el caso de los literales, el tipo final es el propio literal, mientras que el tipo de una variable x es α_x por la suposición anterior. La regla [TPL] fusiona los tipos anotados de cada subexpresión con el operador $\otimes$ para obtener como resultado un tipo tupla. La regla [LST] hace lo mismo para constructores de listas. La diferencia es que convierte el tipo tupla de cada par obtenido con $\otimes$ en un tipo `nelist` para formar el tipo anotado del resultado.

Para la regla [ABS], las nuevas restricciones generadas—a partir del cuerpo de la λ-abstracción—no salen del esquema de tipo funcional que se construye. Ello provoca que el tipo anotado final sea un par sin restricciones. Para construir el esquema de tipos de la λ-abstracción tomamos cada par del tipo anotado generado a partir del cuerpo de la misma y generamos un tipo condicional con restricciones a partir de dicho par, que junto a las variables de tipo $\overline{\alpha_{x_i}}^{n}$—que representan el tipo para cada parámetro $\overline{x_i}^{n}$—conforma el tipo funcional sobre el que haremos el cierre de variables. Entonces, para cerrar el tipo funcional y obtener así un esquema, tomamos todas las variables libres dentro del tipo funcional, a excepción de las α_x asociadas a las variables de programa libres de la función $\textbf{fun}(\overline{x_i}^{n}) \to e$. Una vez conformados los esquemas de tipo funcional, se construye el tipo sobrecargado final, que es el tipo de la λ-abstracción. Por su parte, la regla [APP] indica que el tipo de una aplicación de función es una variable fresca δ, relacionada con los tipos de los argumentos mediante una restricción de aplicación simbólica.

En el caso de las cláusulas, usamos la regla vista en la figura 5.3. Los juicios de la regla [CLS]—que definen el proceso de generación de restricciones—aceptan como parámetro una variable de tipo, teniendo la forma $cls \vdash_\alpha \rho$. La variable α es el tipo que representa al discriminante del **case** al que pertenece la cláusula o una variable de tipo fresca suelta en el caso del **receive**. Esta variable α se utiliza para generar una restricción de encaje que va ligada al tipo que se obtiene con el patrón p, el cual no genera restricciones debido a la naturaleza de los patrones en el lenguaje *Mini Erlang*. Con la operación $\otimes$ fusionamos los tipos anotados que hemos obtenido de la guarda y del cuerpo de la cláusula, obteniendo otro tipo anotado con tipos tupla, donde la primera componente es el resultado de la guarda τ'_i y la segunda componente el tipo final de la cláusula τ_i. A cada conjunto de restricciones C_i obtenido, añadimos una restricción que encaja α con el tipo del patrón τ_p y otra que indica que τ'_i ha de contener el átomo `'true'`. Por último, se renombran las variables de tipo que representan las

variables de programa que se encuentran dentro del patrón, porque al ser las variables $\overline{x_j}^m$ ligadas en la cláusula, debemos evitar que colisionen con otras variables de igual nombre que puedan existir en un ámbito superior. En el sistema de derivación de tipos del capítulo 4 conseguíamos esto mediante la notación $\Gamma(x_j)$, que asignaba tipo `any()` a estas variables ligadas, pero en este caso no podemos hacer esto, pues estas variables x_j tienen implícitamente el tipo α_{x_j} en todos los ámbitos. Por tanto, aplicamos un renombramiento sobre las variables α_{x_j} para evitar que estos tipos interfieran con los de otras posibles variables homónimas en un ámbito superior.

Ahora que hemos explicado la regla para las cláusulas, la regla [CAS] de generación de restricciones para las expresiones **case** consiste en unir los tipos anotados de cada cláusula en una unión de tipos anotados. En la regla [RCV] se realiza algo similar, pero con algunas diferencias. La primera diferencia de la regla es que a cada conjunto de restricciones—obtenido a partir de las cláusulas—hay que añadir una restricción que indique que el tipo de la variable α_{x_t}, que es el tipo de la variable situada en la cláusula **after**, sea subconjunto del tipo `integer()` ∪ `'infinity'`. La segunda diferencia es que para la cláusula **after** obtendremos un tipo anotado correspondiente a la expresión e y a sus restricciones le añadiremos que la variable α_{x_t} ha de ser subconjunto de `integer()`. Por último, para analizar las cláusulas de una expresión **receive** pasaremos una variable fresca δ en los juicios de cada cláusula, porque no hay discriminante con el que poder ligar cada patrón.

Mediante la regla [LET] analizamos cada subexpresión de una expresión **let**, fusionando los resultados de ambas mediante el operador ⊗, cuyo resultado es una tupla donde la primera componente es el tipo τ_i', el resultado de analizar e_1 que se va a asignar a la variable ligada x, y la segunda componente es el tipo τ_i, el resultado de analizar e_2. Para el resultado final construimos, para cada par obtenido de la operación ⊗, un par tomando el tipo τ_i y el conjunto de restricciones C_i, a la que añadiremos una restricción para encajar α_x con el tipo τ_i'. Como vimos con las cláusulas, renombraremos la variable α_x para evitar colisiones con variables de tipo del mismo nombre utilizadas en ámbitos superiores.

Por último, la regla [LRC] genera un tipo para cada λ-abstracción, cuyas restricciones se encuentran dentro del tipo τ_i para cada f_i donde $i \in \{1..n\}$. También obtenemos un tipo anotado para la expresión e, donde a cada par le añadiremos restricciones de encaje que liguen cada variable α_{x_i} con el tipo τ_i. Al final, como hicimos con la regla [LET], haremos un renombramiento de las variables $\overline{\alpha_{x_i}}^n$.

5.2.1. Propiedades de la generación de restricciones

En la sección 4.5 hemos presentado las reglas de derivación de *success types* polimórficos de nuestro sistema de tipos y en la sección 4.7 hemos demostrado que son correctas mediante el teorema 2. Como ya hemos mencionado antes en esta sección, necesitamos las reglas de las figuras 5.2 y 5.3 para poder obtener *success types* de una expresión sin necesidad de conocer los tipos de las funciones que son aplicadas dentro de la misma. Para demostrar que estas reglas también son correctas, necesitaremos definir cuál es la relación que guardan con las reglas de derivación vistas en la figura 4.5 y demostrar que esta relación es correcta.

Al principio de esta sección hemos visto una definición alternativa para el operador ⊗ cuando los tipos anotados pertenecen a **TypeAn***. La siguiente propiedad aclara que el operador ⊗ sigue siendo correcto para todo $\rho \in$ **TypeAn***:

Proposición 17. *Para cada sustitución* θ*, valores* $v_1, \ldots, v_n$ *y tipos anotados* $\rho_1, \ldots, \rho_n$*, tales que* $(\theta, v_i) \in \mathcal{T}\,[\![\rho_i]\!]$ *y* $\rho_i \in$ **TypeAn*** *para cada* $i \in \{1..n\}$*, se tiene que* $\left(\theta, \left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right)\right) \in \mathcal{T}\,[\![\rho_1 \otimes \cdots \otimes \rho_n]\!]$.

Demostración. La demostración es sencilla, ya que cada ρ_i, que pertenece a **TypeAn***, para todo $i \in \{1..n\}$, tendrá la forma $\langle \tau_1; \Gamma_1 \rangle; \ldots; \langle \tau_m; \Gamma_m \rangle$, donde cada Γ_j pertenece a **Env***. Por lo tanto, todos los entornos están normalizados con la forma $\left[\overline{x_i : \alpha_{x_i}} \mid C\right]$ y se puede aplicar el ínfimo entre ellos, siguiendo la demostración de la propiedad 10 sin aplicar el paso de normalizar los entornos, al estar ya normalizados. □

Para determinar si las reglas de generación de restricciones son correctas, queremos demostrar que a partir de la derivación de un juicio de la forma $e \vdash \rho$, obtenida a partir de las reglas de generación de restricciones, es posible encontrar una derivación de tipo en la que e obtenga el tipo anotado ρ. Para lograrlo hemos de entender que los juicios para derivar tipos, vistos en la figura 4.5, tienen las siguientes tres formas: (a) $\Gamma \vdash e : \rho$, (b) $\Gamma \Vdash_\alpha cls : \rho$ y (c) $\Gamma \vdash p : \tau$, donde e es una expresión general del lenguaje, cls es una cláusula, p es una expresión patrón, α es una variable de tipo que representa la variable con la que encaja la cláusula, Γ es el entorno de entrada, ρ es el tipo anotado final y τ es el tipo final del el patrón, tales que $\rho \in$ **TypeAn**, $\tau \in$ **Type** y $\Gamma \in$ **Env**. Por otro lado, los juicios para la generación de restricciones, vistos en la figura 5.2, tienen las dos formas siguientes: (1) $e \vdash \rho'$ y (2) $cls \vdash_\alpha \rho'$, donde ρ' es un tipo anotado, pero a diferencia de los juicios para derivar, tenemos que $\rho' \in$ **TypeAn***.

Un factor relevante es que los juicios correspondientes a las reglas de generación de restricciones no contienen ningún entorno inicial, mientras que los juicios del sistema de tipos sí disponen de uno. La propiedad de corrección de las reglas de generación de restricciones que queremos demostrar establece que si ρ' es el resultado obtenido a partir de las reglas para una expresión, patrón o cláusula, entonces para todo entorno inicial Γ puede derivarse un tipo anotado ρ para esa misma expresión, patrón, o cláusula, y se cumple la relación $\rho \subseteq \rho' \sqcap \Gamma$. Para lograr esto tenemos que definir la operación de ínfimo entre tipos anotados y entornos. Definimos el ínfimo entre un tipo anotado $\rho \in$ **Type*** y un entorno $\Gamma \in$ **Env** de la siguiente manera:

$$\rho \sqcap \Gamma = \langle \tau_1; \Gamma_1 \sqcap \Gamma \rangle; \ldots; \langle \tau_n; \Gamma_n \sqcap \Gamma \rangle$$
$$\text{donde } \rho = \langle \tau_1; \Gamma_1 \rangle; \ldots; \langle \tau_n; \Gamma_n \rangle$$

Esto nos lleva a adaptar la operación de ínfimo vista en 4.4 aplicada a dos entornos $\Gamma \in$ **Env** y $\Gamma' \in$ **Env***,

haciéndolo de la siguiente forma:

$$\begin{aligned}\Gamma \sqcap \Gamma' &= \left[\overline{x_i : \alpha_{x_i}} \mid C \cup C'\right] \\ \text{donde } \mathit{norm}(\Gamma) &= \left[\overline{x_i : \alpha_{x_i}} \mid C\right] \text{ y } \Gamma' = \left[\overline{x_i : \alpha_{x_i}} \mid C'\right] \\ \text{y } \mathit{ftv}(\Gamma) \cap \mathit{ftv}\left(\Gamma'\right) &= \emptyset\end{aligned}$$

La diferencia—con respecto a la definición del capítulo anterior—es que aquí Γ' ya está normalizado por su propia naturaleza y como las variables de tipo que no están ligadas a una variable de programa son frescas, no existe la posibilidad de una colisión entre los dos entornos. Por abuso de notación también usaremos en la demostración la notación $\rho \sqcap C$, que equivale a $\rho \sqcap \Gamma$ para un entorno $\Gamma \in \mathbf{Env}^\star$, tal que $\Gamma|_C = C$ y $\Gamma(x) = \alpha_x$ para toda variable $x \in \mathbf{Var}$. Dado que Γ y los entornos de ρ están normalizados, el ínfimo entre $\rho \sqcap C$ es, simplemente, la unión de C a todos los conjuntos de restricciones que hay en ρ.

Ahora podemos empezar demostrando la relación entre las reglas para derivar tipos para patrones y las reglas de generación de restricciones, con la siguiente proposición:

Proposición 18. *Suponemos un patrón p, un entorno $\Gamma' \in \mathbf{Env}^\star$ y un tipo $\tau' \in \mathbf{Type}^\star$. Se obtiene que:*

$$p \vdash \langle \tau'; \Gamma' \rangle \implies \forall \Gamma \in \mathbf{Env}.\ \exists\, \tau \in \mathbf{Type}.\ \Gamma \vdash p : \tau \ \wedge\ \mathscr{T}\,[\![\langle \tau; \Gamma \rangle]\!] \subseteq \mathscr{T}\,\left[\!\left[\langle \tau'; \Gamma' \rangle \sqcap \Gamma\right]\!\right]$$

Demostración. El primer paso para demostrar la propiedad es aclarar que las reglas de generación que vamos a usar ([CNS], [VAR], [TPL] y [LST]) no generan restricciones nuevas. Sabemos que [CNS] y [VAR] devuelven un conjunto vacío de restricciones, y dado que [TPL] y [LST] dependen del resultado obtenido para cada una de sus componentes, sabemos que también devuelven un conjunto vacío de restricciones, ya que [TPL] y [LST] no añaden restricciones nuevas y en última instancia el caso base para los patrones son expresiones literales o variables, cuyas reglas [CNS] y [VAR] tampoco devuelven restricciones. También sabemos que ninguna de las cuatro reglas genera una secuencia de pares como tipo anotado, por lo que el resultado siempre será un par con el conjunto de restricciones vacío.

Podemos deducir ahora que cada juicio $p \vdash \rho$ tiene en realidad la forma $p \vdash \langle \tau'; \emptyset \rangle$. El entorno de salida, al que nos vamos a referir como Γ', tiene la forma $\left[\overline{x_i : \alpha_{x_i}}\right]$ ya que esa es la forma que tienen los entornos vacíos dentro de $\mathbf{Env}^\star$. Al hacer el ínfimo con Γ y Γ' el resultado es:

$$\begin{aligned}\Gamma \sqcap \left[\overline{x_i : \alpha_{x_i}}\right] = \left[\overline{x_i : \alpha_{x_i}} \mid C\right] &= \mathit{norm}(\Gamma) \\ \text{donde } \mathit{norm}(\Gamma) = \left[\overline{x_i : \alpha_{x_i}} \mid C\right] &\text{ y } \mathit{ftv}(\Gamma) \cap \mathit{ftv}\left(\Gamma'\right) = \emptyset\end{aligned}$$

Como $\mathit{ftv}\left(\Gamma'\right) = \{\overline{\alpha_{x_i}}\}$, porque $\Gamma'\big|_C = \emptyset$, suponiendo un $\theta \in \mathscr{T}^{\pi}_{Env}[\![\Gamma]\!]$, existirá una instanciación π' tal que $\pi' \equiv \pi$ (módulo $\setminus \{\overline{\alpha_{x_i}}\}$) y $\theta \in \mathscr{T}^{\pi'}_{Env}[\![\mathit{norm}(\Gamma)]\!]$. Con la instanciación π', al hacer el ínfimo entre Γ y Γ', tendremos que $\theta \in \mathscr{T}^{\pi'}_{Env}[\![\mathit{norm}(\Gamma)]\!]$ y $\theta \in \mathscr{T}^{\pi'}_{Env}\left[\!\left[\Gamma'\right]\!\right]$, que se queda en $\theta \in \mathscr{T}^{\pi'}_{Env}[\![\mathit{norm}(\Gamma)]\!]$ porque Γ'

no tiene restricciones. Por lo tanto:

$$\mathscr{T}\left[\!\left[\langle \tau'; \Gamma' \rangle \sqcap \Gamma\right]\!\right] = \mathscr{T}\left[\!\left[\langle \tau'; norm(\Gamma) \rangle\right]\!\right]$$

Continuaremos la demostración caso por caso con las reglas para derivar tipos para los patrones:

- **Caso [LIT$_P$]**

  Suponemos el juicio $\Gamma \vdash c : c$, obtenido con la regla [LIT$_P$], y el juicio $c \vdash \langle c; \emptyset \rangle$, obtenido con la regla de generación [CNS], entonces:

$$\begin{array}{rcl} \mathscr{T}\left[\!\left[\langle c; \emptyset \rangle \sqcap \Gamma\right]\!\right] & = & \mathscr{T}\left[\!\left[\langle c; norm(\Gamma) \rangle\right]\!\right] \\ & = & \mathscr{T}\left[\!\left[\langle c; \Gamma \rangle\right]\!\right] \end{array}$$

- **Caso [VAR$_P$]**

  Suponemos el juicio $\Gamma\,[x : \alpha] \vdash x : \alpha$, obtenido con la regla [VAR$_P$], y el juicio $x \vdash \langle \alpha_x; \emptyset \rangle$, obtenido con la regla de generación [VAR], entonces:

$$\begin{array}{rcl} \mathscr{T}\left[\!\left[\langle \alpha_x; \emptyset \rangle \sqcap \Gamma\,[x : \alpha]\right]\!\right] & = & \mathscr{T}\left[\!\left[\langle \alpha_x; norm(\Gamma\,[x : \alpha]) \rangle\right]\!\right] \\ & = & \mathscr{T}\left[\!\left[\left\langle \alpha_x; \left[\overline{x_i : \alpha_{x_i}} \mid C \cup \{\alpha \Leftarrow \alpha_x\}\right] \right\rangle\right]\!\right] \\ & = & \mathscr{T}\left[\!\left[\left\langle \alpha; \left[\overline{x_i : \alpha_{x_i}} \mid C \cup \{\alpha \Leftarrow \alpha_x\}\right] \right\rangle\right]\!\right] \\ & = & \mathscr{T}\left[\!\left[\langle \alpha; norm(\Gamma\,[x : \alpha]) \rangle\right]\!\right] \\ & = & \mathscr{T}\left[\!\left[\langle \alpha; \Gamma\,[x : \alpha] \rangle\right]\!\right] \end{array}$$

- **Caso [TPL$_P$]**

  Suponemos el juicio $\Gamma \vdash \{\overline{p_i}^n\} : \{\overline{\tau_i}^n\}$, obtenido con la regla [TPL$_P$], y el juicio $\{\overline{p_i}^n\} \vdash \rho_1 \otimes \cdots \otimes \rho_n$, obtenido con la regla de generación [TPL]. Sabemos por la regla de generación [TPL] que el juicio $p_i \vdash \rho_i$ es $p_i \vdash \langle \tau'_i; \emptyset \rangle$ para cada $i \in \{1..n\}$. Aplicando la operación $\otimes$, el juicio final de la regla de generación [TPL] se transforma en lo siguiente:

$$\begin{array}{c} \{\overline{p_i}^n\} \vdash \rho_1 \otimes \cdots \otimes \rho_n \\ \Updownarrow \\ \{\overline{p_i}^n\} \vdash \langle \tau'_1; \emptyset \rangle \otimes \cdots \otimes \langle \tau'_n; \emptyset \rangle \\ \Updownarrow \\ \{\overline{p_i}^n\} \vdash \left\langle \{\overline{\tau'_i}^n\}; \emptyset \right\rangle \end{array}$$

  Por hipótesis de inducción, para cada $i \in \{1..n\}$, el juicio $p_i \vdash \langle \tau'_i; \emptyset \rangle$ obtiene la existencia de un tipo τ_i, tal que se cumple el juicio $\Gamma \vdash p_i : \tau_i$ y $\mathscr{T}\left[\!\left[\langle \tau_i; \Gamma \rangle\right]\!\right] \subseteq \mathscr{T}\left[\!\left[\langle \tau'_i; \emptyset \rangle \sqcap \Gamma\right]\!\right]$. Siguiendo la

semántica de los tipos anotados, tenemos que:

$$\begin{array}{ll} & \mathscr{T}\left[\!\left[\left\langle\{\overline{\tau'_i}^n\};\emptyset\right\rangle \sqcap \Gamma\right]\!\right] \\ = & \mathscr{T}\left[\!\left[\left\langle\{\overline{\tau'_i}^n\};[\overline{x_i:\alpha_{x_i}}]\sqcap\Gamma\right\rangle\right]\!\right] \\ = & \left\{\left(\theta,\left(\{\cdot^n\cdot\},\overline{v_i}^n\right)\right) \mid \theta\in\mathscr{T}^{\pi}_{Env}[\![[\overline{x_i:\alpha_{x_i}}]\sqcap\Gamma]\!], \left(\left(\{\cdot^n\cdot\},\overline{v_i}^n\right),\pi\right)\in\mathscr{T}\left[\!\left[\{\overline{\tau'_i}^n\}\right]\!\right]\right\} \\ = & \left\{\left(\theta,\left(\{\cdot^n\cdot\},\overline{v_i}^n\right)\right) \mid \theta\in\mathscr{T}^{\pi}_{Env}[\![[\overline{x_i:\alpha_{x_i}}]\sqcap\Gamma]\!], \forall i\in\{1..n\}.(v_i,\pi)\in\mathscr{T}[\![\tau'_i]\!]\right\} \\ \supseteq & \quad \{\text{ Porque } \mathscr{T}[\![\langle\tau_i;\Gamma\rangle]\!]\subseteq\mathscr{T}\left[\!\left[\left\langle\tau'_i;\emptyset\right\rangle\sqcap\Gamma\right]\!\right] \text{ para cada } i\} \\ & \left\{\left(\theta,\left(\{\cdot^n\cdot\},\overline{v_i}^n\right)\right) \mid \theta\in\mathscr{T}^{\pi}_{Env}[\![\Gamma]\!], \forall i\in\{1..n\}.(v_i,\pi)\in\mathscr{T}[\![\tau_i]\!]\right\} \\ = & \mathscr{T}[\![\langle\{\overline{\tau_i}^n\};\Gamma\rangle]\!] \end{array}$$

- **Caso [LST$_P$]**

  Suponemos el juicio $\Gamma\vdash[p_1\mid p_2]:\texttt{nelist}(\tau_1,\tau_2)$, obtenido con la regla [LST$_P$], y el juicio $[p_1\mid p_2]\vdash\overline{\left\langle\texttt{nelist}(\tau'_{i,1},\tau'_{i,2});C_i\right\rangle}^n$, obtenido con la regla de generación [LST]. Sabemos por la regla de generación [LST] que los juicios $p_1\vdash\rho_1$ y $p_2\vdash\rho_2$ son en realidad $p_1\vdash\langle\tau'_1;\emptyset\rangle$ y $p_2\vdash\langle\tau'_2;\emptyset\rangle$. También sabemos por la regla de generación [LST] que $\rho_1\otimes\rho_2=\overline{\left\langle\{\tau'_{i,1},\tau'_{i,2}\};C_i\right\rangle}^n$, pero como sabemos el valor para ρ_1 y ρ_2, tenemos que:

$$\begin{array}{c} \rho_1\otimes\rho_2=\overline{\left\langle\{\tau'_{i,1},\tau'_{i,2}\};C_i\right\rangle}^n \\ \Updownarrow \\ \langle\tau'_1;\emptyset\rangle\otimes\langle\tau'_2;\emptyset\rangle=\overline{\left\langle\{\tau'_{i,1},\tau'_{i,2}\};C_i\right\rangle}^n \\ \Updownarrow \\ \langle\{\tau'_1,\tau'_2\};\emptyset\rangle=\left\langle\{\tau'_{1,1},\tau'_{1,2}\};C_1\right\rangle \end{array}$$

  Por lo tanto el juicio se transforma en $[p_1\mid p_2]\vdash\langle\texttt{nelist}(\tau'_1,\tau'_2);\emptyset\rangle$. Luego por hipótesis de inducción sobre $p_1\vdash\langle\tau'_1;\emptyset\rangle$ y $p_2\vdash\langle\tau'_2;\emptyset\rangle$, sabemos que existen unos tipos τ_1 y τ_2 respectivamente, tales que se cumple $\Gamma\vdash p_1:\tau_1$, $\mathscr{T}[\![\langle\tau_1;\Gamma\rangle]\!]\subseteq\mathscr{T}\left[\!\left[\left\langle\tau'_1;\emptyset\right\rangle\sqcap\Gamma\right]\!\right]$, $\Gamma\vdash p_2:\tau_2$ y $\mathscr{T}[\![\langle\tau_2;\Gamma\rangle]\!]\subseteq\mathscr{T}\left[\!\left[\left\langle\tau'_2;\emptyset\right\rangle\sqcap\Gamma\right]\!\right]$. Siguiendo la semántica de los tipos anotados, tenemos que:

$$\begin{array}{ll} & \mathscr{T}\left[\!\left[\left\langle\texttt{nelist}(\tau'_1,\tau'_2);\emptyset\right\rangle\sqcap\Gamma\right]\!\right] \\ = & \mathscr{T}\left[\!\left[\left\langle\texttt{nelist}(\tau'_1,\tau'_2);[\overline{x_i:\alpha_{x_i}}]\sqcap\Gamma\right\rangle\right]\!\right] \\ = & \{(\theta,([\_\mid\_],\overline{v_i}^n,v')) \mid \theta\in\mathscr{T}^{\pi}_{Env}[\![[\overline{x_i:\alpha_{x_i}}]\sqcap\Gamma]\!], ((\,[\_\mid\_],\overline{v_i}^n,v'),\pi)\in\mathscr{T}\left[\!\left[\texttt{nelist}(\tau'_1,\tau'_2)\right]\!\right]\} \\ = & \{(\theta,([\_\mid\_],\overline{v_i}^n,v')) \mid \theta\in\mathscr{T}^{\pi}_{Env}[\![[\overline{x_i:\alpha_{x_i}}]\sqcap\Gamma]\!], n\geq 1, \forall i\in\{1..n\}.(v_i,\pi_i)\in\mathscr{T}[\![\tau'_1]\!], \\ & \quad \pi\in Dcp(\{\overline{\pi_i}^n\},\tau'_1),(v',\pi)\in\mathscr{T}[\![\tau'_2]\!]\} \\ \supseteq & \quad \{\text{ Porque } \mathscr{T}[\![\langle\tau_i;\Gamma\rangle]\!]\subseteq\mathscr{T}\left[\!\left[\left\langle\tau'_i;\emptyset\right\rangle\sqcap\Gamma\right]\!\right] \text{ para cada } i\} \\ & \{(\theta,([\_\mid\_],\overline{v_i}^n,v')) \mid \theta\in\mathscr{T}^{\pi}_{Env}[\![\Gamma]\!], n\geq 1, \forall i\in\{1..n\}.(v_i,\pi_i)\in\mathscr{T}[\![\tau_1]\!], \\ & \quad \pi\in Dcp(\{\overline{\pi_i}^n\},\tau_1),(v',\pi)\in\mathscr{T}[\![\tau_2]\!]\} \\ = & \mathscr{T}[\![\langle\texttt{nelist}(\tau_1,\tau_2);\Gamma\rangle]\!] \end{array}$$

□

Tras demostrar la relación entre las reglas para derivar tipos para los patrones y las reglas de generación de restricciones, vamos ahora a demostrar la relación entre las reglas para derivar tipos para expresiones y las reglas de generación de restricciones, junto a las reglas para derivar tipos para cláusulas y las reglas de generación de restricciones para cláusulas. Esta demostración la tenemos que hacer conjunta entre estos dos tipos de juicios, ya que dependen recursivamente entre sí.

Teorema 3. *Suponemos una expresión e, una cláusula cls, una variable de tipo α y un tipo anotado $\rho \in \mathbf{TypeAn}^\star$. Se obtiene que:*

- $e \vdash \rho \implies \forall \Gamma \in \mathbf{Env}.\ \exists\, \rho' \in \mathbf{TypeAn}.\ \Gamma \vdash e : \rho' \ \wedge\ \mathscr{T}\,[\![\rho']\!] \subseteq \mathscr{T}\,[\![\rho \sqcap \Gamma]\!]$
- $cls \vdash_\alpha \rho \implies \forall \Gamma \in \mathbf{Env}.\ \exists\, \rho' \in \mathbf{TypeAn}.\ \Gamma \Vdash_\alpha cls : \rho' \ \wedge\ \mathscr{T}\,[\![\rho']\!] \subseteq \mathscr{T}\,[\![\rho \sqcap \Gamma]\!]$

Demostración. Por inducción sobre la derivación de los juicios $e \vdash \rho$ y $cls \vdash_\alpha \rho$ de generación de restricciones, vamos a demostrar caso por caso cada regla:

- **Caso [CNS]**

  Suponemos el juicio $c \vdash \langle c; \emptyset \rangle$, obtenido con la regla de generación [CNS]. Dado un Γ cualquiera, podemos obtener el juicio $\Gamma \vdash c : \langle c; \Gamma \rangle$ con la regla de tipado [CNS]. El resultado de $\langle c; \emptyset \rangle \sqcap \Gamma$ es $\langle c; norm(\Gamma) \rangle$ y sabemos que $\mathscr{T}\,[\![\langle c; norm(\Gamma)\rangle]\!] = \mathscr{T}\,[\![\langle c; \Gamma \rangle]\!]$, demostrando por lo tanto el teorema para las reglas [CNS].

- **Caso [VAR]**

  Suponemos el juicio $x \vdash \langle \alpha_x; \emptyset \rangle$, obtenido con la regla de generación [VAR]. Dado un Γ cualquiera, podemos obtener el juicio $\Gamma \vdash x : \langle \Gamma(x); \Gamma \rangle$ con la regla de tipado [VAR]. Entonces:

  $$\begin{array}{rcl} \langle \alpha_x; \emptyset \rangle \sqcap \Gamma & = & \langle \alpha_x; norm(\Gamma) \rangle \\ & = & \left\langle \alpha_x; \left[\overline{x_i : \alpha_{x_i}} \mid C \cup \{\Gamma(x) \Leftarrow \alpha_x\} \right] \right\rangle \end{array}$$

  Por lo tanto podemos sustituir α_x por $\Gamma(x)$, obteniendo $\mathscr{T}\,[\![\langle \Gamma(x); norm(\Gamma)\rangle]\!] = \mathscr{T}\,[\![\langle \Gamma(x); \Gamma\rangle]\!]$, ya que $norm(\Gamma) \approx \Gamma$, demostrando por lo tanto el teorema para las reglas [VAR].

- **Caso [TPL]**

  Suponemos el juicio $\{\overline{e_i}^{\,n}\} \vdash \rho_1 \otimes \cdots \otimes \rho_n$, obtenido con la regla de generación [TPL]. Dado un Γ cualquiera, podemos obtener el juicio $\Gamma \vdash \{\overline{e_i}^{\,n}\} : \rho'_1 \otimes \cdots \otimes \rho'_n$ con la regla de tipado [TPL]. Por hipótesis de inducción sabemos que, para cada $i \in \{1..n\}$, con el entorno Γ y el juicio $e_i \vdash \rho_i$, existe un tipo anotado ρ'_i tal que $\Gamma \vdash e_i : \rho'_i$ y $\mathscr{T}\,[\![\rho'_i]\!] \subseteq \mathscr{T}\,[\![\rho_i \sqcap \Gamma]\!]$.

  Por la proposición 17 sabemos que $\rho_1 \otimes \cdots \otimes \rho_n$ equivale a $\overline{\langle \{\overline{\tau_{j,i}}^{\,n}\}; \Gamma_j \rangle}^{\,m}$. Si suponemos un ínfimo $(\rho_1 \otimes \cdots \otimes \rho_n) \sqcap \Gamma$, tendremos que $\overline{\langle \{\overline{\tau_{j,i}}^{\,n}\}; \Gamma_j \rangle}^{\,m} \sqcap \Gamma$, que por definición es $\overline{\langle \{\overline{\tau_{j,i}}^{\,n}\}; \Gamma_j \sqcap \Gamma \rangle}^{\,m}$, lo que quiere decir que cada sustitución θ que pertenece a la semántica de Γ_j, también pertenece a la semántica de Γ, bajo una instanciación π. Cada Γ_j es el resultado del ínfimo entre n

entornos tomados de cada ρ_i, donde $i \in \{1..n\}$, por lo que todos los entornos de salida que hay en cada ρ_i tienen que hacer el ínfimo con Γ, que aplicando las definiciones a la inversa obtendríamos $\rho_i \sqcap \Gamma$ y por lo tanto $(\rho_1 \sqcap \Gamma) \otimes \cdots \otimes (\rho_n \sqcap \Gamma)$.

Por la proposición 17 sabemos que $\left(\theta, \left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right)\right) \in \mathscr{T}[\![(\rho_1 \sqcap \Gamma) \otimes \cdots \otimes (\rho_n \sqcap \Gamma)]\!]$, para una sustitución θ y unos valores $v_1, \ldots, v_n$, obteniendo que $(\theta, v_i) \in \mathscr{T}[\![\rho_i \sqcap \Gamma]\!]$ para cada $i \in \{1..n\}$. Como tenemos que $\mathscr{T}[\![\rho'_i]\!] \subseteq \mathscr{T}[\![\rho_i \sqcap \Gamma]\!]$, entonces $(\theta, v_i) \in \mathscr{T}[\![\rho'_i]\!]$, y por lo tanto también tenemos $\mathscr{T}[\![\rho_1 \otimes \cdots \otimes \rho'_n]\!]$, obteniendo así que $\mathscr{T}[\![\rho_1 \otimes \cdots \otimes \rho'_n]\!] \subseteq \mathscr{T}[\![(\rho_1 \otimes \cdots \otimes \rho_n) \sqcap \Gamma]\!]$, demostrando por lo tanto el teorema para las reglas [TPL].

- **Caso [LST]**

  Suponemos el juicio $[e_1 \mid e_2] \vdash \overline{\langle \texttt{nelist}(\tau_{i,1}, \tau_{i,2}); C_i \rangle}^n$, obtenido con la regla de generación [LST]. Dado un Γ cualquiera, podemos obtener el juicio $\Gamma \vdash [e_1 \mid e_2] : \overline{\langle \texttt{nelist}(\tau'_{i,1}, \tau'_{i,2}); \Gamma'_i \rangle}^m$ con la regla de tipado [LST]. De modo similar a la demostración con la regla [TPL], al tomar el tipo obtenido por la regla de generación y aplicar el ínfimo con Γ, se lo tenemos que aplicar a cada entorno de salida del tipo anotado. De este modo sabemos que con $\overline{\langle \texttt{nelist}(\tau_{i,1}, \tau_{i,2}); C_i \sqcap \Gamma \rangle}^n$ podemos deducir que tendremos $\overline{\langle \{\tau_{i,1}, \tau_{i,2}\}; C_i \sqcap \Gamma \rangle}^n$ y por consiguiente $\rho_1 \sqcap \Gamma$ y $\rho_2 \sqcap \Gamma$. Entonces, aplicando la hipótesis por inducción sobre los juicios $e_1 \vdash \rho_1$ y $e_2 \vdash \rho_2$, obtenemos la existencia de unos tipos anotados ρ'_1 y ρ'_2 respectivamente, tales que $\Gamma \vdash e_1 : \rho'_1$, $\mathscr{T}[\![\rho'_1]\!] \subseteq \mathscr{T}[\![\rho_1 \sqcap \Gamma]\!]$, $\Gamma \vdash e_2 : \rho'_2$ y $\mathscr{T}[\![\rho'_2]\!] \subseteq \mathscr{T}[\![\rho_2 \sqcap \Gamma]\!]$, obteniendo que $\mathscr{T}[\![\rho'_1 \otimes \rho'_2]\!] \subseteq \mathscr{T}[\![(\rho_1 \otimes \rho_2) \sqcap \Gamma]\!]$ y por lo tanto que $\mathscr{T}\left[\!\left[\overline{\langle \texttt{nelist}(\tau'_{i,1}, \tau'_{i,2}); \Gamma'_i \rangle}^m\right]\!\right] \subseteq \mathscr{T}\left[\!\left[\overline{\langle \texttt{nelist}(\tau_{i,1}, \tau_{i,2}); C_i \rangle}^n \sqcap \Gamma\right]\!\right]$, demostrando por lo tanto el teorema para las reglas [LST].

- **Caso [ABS]**

  Suponemos el juicio $\mathbf{fun}(\overline{x_i}^n) \to e \vdash \left\langle \bigsqcup_{j=1}^s \forall \overline{\alpha_{j,k}}.(\overline{\alpha_{x_i}}^n) \to \tau_j \ \mathbf{when}\ C_j; \emptyset \right\rangle$, obtenido con la regla de generación [ABS]. Dado un Γ cualquiera, podemos obtener el juicio $\Gamma \vdash \mathbf{fun}(\overline{x_i}^n) \to e : \left\langle \bigsqcup_{j=1}^m \forall \overline{\alpha'_{j,k}}.\left(\overline{\tau'_{j,i}}^n\right) \to \tau'_j; \Gamma \right\rangle$ con la regla de tipado [ABS], tal que el conjunto de variables de tipo $\left\{\overline{\alpha'_{j,k}}\right\} = itv\left(\left(\overline{\tau'_{j,i}}^n\right) \to \tau'_j\right)$ para cada $j \in \{1..m\}$. Sabemos por la regla de tipado [ABS] que $\Gamma = \Gamma_0\left[\overline{x_i : \texttt{any}()}^n, \overline{y_i : \alpha_i}\right]$ y $ftv\left(\forall \overline{\alpha'_{j,k}}.\left(\overline{\tau'_{j,i}}^n\right) \to \tau'_j\right) \subseteq \{\overline{\alpha_i}\}$ para cada $j \in \{1..m\}$. Por hipótesis de inducción, tomando Γ y el juicio $e \vdash \overline{\langle \tau_j; C_j \rangle}^s$, obtendremos que existe un tipo anotado $\overline{\langle \tau'_j; \Gamma'_j \rangle}^m$ tal que $\Gamma \vdash e : \overline{\langle \tau'_j; \Gamma'_j \rangle}^m$ y $\mathscr{T}\left[\!\left[\overline{\langle \tau'_j; \Gamma'_j \rangle}^m\right]\!\right] \subseteq \mathscr{T}\left[\!\left[\overline{\langle \tau_j; C_j \rangle}^s \sqcap \Gamma\right]\!\right]$. Suponemos que para cada $j \in \{1..m\}$ el entorno Γ'_j tiene la forma $\Gamma\left[\overline{x_i : \tau'_{j,i}}^n\right]$.

  Las variables de tipo $\overline{\alpha_{j,k}}$, que cerramos al construir el esquema de tipos funcional, son todas aquellas variables de tipo libres que no están asociadas a las variables de programa que están libres en la λ-abstracción, como se ve en la condición de la regla de generación [ABS]:

$$\forall\, j \in \{1..s\}: \quad \left\{\overline{\alpha_{j,k}}\right\} = ftv\left(\left(\overline{\alpha_{x_i}}^n\right) \to \tau_j \ \mathbf{when}\ C_j\right) \setminus \left\{\alpha_x \mid x \in freevars\left(\mathbf{fun}(\overline{x_i}^n) \to e\right)\right\}$$

  Las variables de tipo devueltas por $\left\{\alpha_x \mid x \in freevars\left(\mathbf{fun}(\overline{x_i}^n) \to e\right)\right\}$ son el conjunto $\left\{\overline{\alpha_{x_i}}\right\}$, que

son las variables de tipo asociadas a las variables de programa $\overline{x_i}$ que están libres en la λ-abstracción. Mientras que en la regla de derivación, las variables de tipo $\{\overline{\alpha_i}\}$ asociadas a las variables de programa $\overline{y_i}$ en Γ representan las condiciones necesarias sobre las variables libres de la λ-abstracción, teniendo que cumplirse la condición $ftv\left(\forall\overline{\alpha'_{j,k}}.\left(\overline{\tau'_{j,i}}^{\,n}\right)\to\tau'_j\right)\subseteq\{\overline{\alpha_i}\}$ para cada $j\in\{1..m\}$, que quiere decir que las únicas variables de tipo libres—que pueden existir en cada esquema de tipo funcional—son exactamente las que están asociadas a las variables de programa $\overline{y_i}$. Por lo tanto $\{\overline{y_i}\}=\mathit{freevars}(\mathbf{fun}(\overline{x_i}^{\,n})\to e)$. Entonces, al normalizar Γ se generan las restricciones de encaje $\overline{\alpha_i\Leftarrow\alpha_{y_i}}$, por lo que se cumple que necesariamente $\{\overline{\alpha_{y_i}}\}=\{\overline{\alpha_{x_i}}\}$, porque ambas reglas hacen referencia a las mismas variables de programa libres.

Por la semántica sabemos que $\mathscr{T}\left[\!\left[\overline{\langle\tau_j;C_j\rangle}^{\,s}\sqcap\Gamma\right]\!\right]$ equivale a $\mathscr{T}\left[\!\left[\overline{\langle\tau_j\ \mathbf{when}\ C_j;\emptyset\rangle}^{\,s}\sqcap\Gamma\right]\!\right]$, por lo tanto $\mathscr{T}\left[\!\left[\overline{\left\langle\tau'_j;\Gamma\left[\overline{x_i:\tau'_{j,i}}^{\,n}\right]\right\rangle}^{\,m}\right]\!\right]\subseteq\mathscr{T}\left[\!\left[\overline{\langle\tau_j\ \mathbf{when}\ C_j;\emptyset\rangle}^{\,s}\sqcap\Gamma\right]\!\right]$. Cuando se normaliza Γ no solo generamos las restricciones $\overline{\alpha_i\Leftarrow\alpha_{y_i}}$, sino que también tenemos que $\mathit{norm}(\Gamma)(x_i)=\alpha_{x_i}$, por lo que para cada $j\in\{1..s\}$ podemos transformar $\left(\overline{\alpha_{x_i}}^{\,n}\right)\to\tau_j\ \mathbf{when}\ C_j$ en $\left(\overline{\alpha_{x_i}}^{\,n}\right)\to\tau'_j$ para cada $j\in\{1..m\}$, que junto a $\Gamma\left[\overline{x_i:\tau'_{j,i}}^{\,n}\right]$ normalizado obtenemos las restricciones $\overline{\tau'_{j,i}\Leftarrow\alpha_{x_i}}^{\,n}$, transformando el tipo funcional en $\left(\overline{\tau'_{j,i}}^{\,n}\right)\to\tau'_j$ para cada $j\in\{1..m\}$. Hacemos el cierre con las variables de tipo $\overline{\alpha'_{j,k}}$ y obtenemos que:

$$\mathscr{T}\left[\!\left[\left\langle\bigsqcup_{j=1}^{m}\forall\overline{\alpha'_{j,k}}.\left(\overline{\tau'_{j,i}}^{\,n}\right)\to\tau'_j;\Gamma\right\rangle\right]\!\right]\quad\subseteq\quad\mathscr{T}\left[\!\left[\left\langle\bigsqcup_{j=1}^{s}\forall\overline{\alpha_{j,k}}.\left(\overline{\alpha_{x_i}}^{\,n}\right)\to\tau_j\ \mathbf{when}\ C_j;\emptyset\right\rangle\sqcap\Gamma\right]\!\right]$$

demostrando por lo tanto el teorema para las reglas [ABS].

- **Caso [APP]**

  Suponemos el juicio $f(\overline{x_i}^{\,n})\vdash\left\langle\beta;\left\{Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{\,n}\right)\Leftarrow\beta\right\}\right\rangle$, obtenido con la regla de generación [APP], tal que β es una variable de tipo fresca. Dado un Γ cualquiera. Distinguimos dos casos:

  - Si $\Gamma\sqcap\left[f:\left(\overline{\mathsf{any}()}^{\,n}\right)\to\mathsf{any}()\right]\approx\bot$, entonces es posible obtener una derivación de tipos $\Gamma\vdash f(\overline{x_i}^{\,n}):\langle\mathsf{none}();\bot\rangle$ utilizando la regla [APP2] del sistema de tipos. Aplicando el ínfimo entre $\left\langle\beta;\left\{Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{\,n}\right)\Leftarrow\beta\right\}\right\rangle$ y Γ, y siguiendo la definición 4 para las aplicaciones simbólicas, encontramos que no se cumple que $\left(\left(\overline{v_i}^{\,n}\right),v\right)\in\pi(\alpha_f)$, porque no es un tipo funcional, por lo tanto las instancias para β y $\overline{\alpha_{x_i}}^{\,n}$ serán vacías y el resultado será $\langle\mathsf{none}();\bot\rangle$, demostrando así el teorema para las reglas [APP] y [APP2].
  - Si $\Gamma\sqcap\left[f:\left(\overline{\mathsf{any}()}^{\,n}\right)\to\mathsf{any}()\right]\approx\Gamma_0$, para un Γ_0 tal que $\Gamma_0(f)=\bigsqcup_{j=1}^{m}\left(\forall\overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^{\,n}\right)\to\tau_j\right)$ y $\Gamma_0(x_i)=\beta_{x_i}$ para cada $i\in\{1..n\}$, entonces es posible obtener una derivación de tipos $\Gamma\vdash f(\overline{x_i}^{\,n}):\overline{\left\langle\tau'_j;\Gamma'_j\right\rangle}^{\,m}$ utilizando la regla [APP1] del sistema de tipos. Cuando Γ_0 sea normalizado, para cada $i\in\{1..n\}$, la variable de programa x_i estará asociada a la variable de tipo α_{x_i} y existirá una restricción de encaje $\beta_{x_i}\Leftarrow\alpha_{x_i}$, por lo que $\pi(\alpha_{x_i})=\pi(\beta_{x_i})$. Por la

regla [APP1] sabemos que:

$$\Gamma'_j = \Gamma_0 \left[\left\{ \overline{\tau_{j,i}\mu_j \Leftarrow \beta_{x_i}}^{\,n} \right\} \cup \{ \beta'\mu_j \subseteq \beta' \mid \beta' \in itv\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right) \} \right]$$

donde $\mu_j = freshRenaming\left(itv\left(\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right) \cup \{\overline{\alpha_{j,i}}\}\right)$, para cada $j \in \{1..m\}$. Aplicando el ínfimo entre $\left\langle \beta; \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{\,n}\right) \Leftarrow \beta \right\} \right\rangle$ y Γ_0, sabemos que una vez normalizado Γ_0 el tipo para f es α_f y que además existe la restricción $\bigsqcup_{j=1}^{m}\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right) \Leftarrow \alpha_f$. Siguiendo la definición 4 para las aplicaciones simbólicas, tenemos que $\left(\left(\overline{v_i}^{\,n}\right), v\right) \in \pi(\alpha_f)$, $v_i \in \pi(\alpha_{x_i})$, para cada $i \in \{1..n\}$, y $v \in \pi(\beta)$, además de existir un tipo τ_f tal que $\left(\pi(\alpha_f), \pi\right) \in \mathcal{T}\,[\![\tau_f]\!]$, por lo que $\tau_f = \bigsqcup_{j=1}^{m}\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right)$ y entonces $\left(\pi(\alpha_f), \pi\right) \in \mathcal{T}\left[\!\!\left[\bigsqcup_{j=1}^{m}\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right)\right]\!\!\right]$. Aplicando el lema 10 obtendremos que:

$$\pi \vDash \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{\,n}\right) \Leftarrow \beta \right\}$$
$$\Updownarrow$$
$$\forall i \in \{1..m\}.\pi_j \vDash \left(\left\{ \overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^{\,n}, \tau_j\mu_j \Leftarrow \beta \right\} \cup \{ \beta'\mu_j \subseteq \beta' \mid \beta' \in itv\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right) \}\right)$$

donde la instanciación $\pi_j \equiv \pi$ (módulo $\setminus rng\,\mu_j$), para todo $j \in \{1..m\}$. Como sabemos que $\pi \vDash \beta_{x_i} \Leftarrow \alpha_{x_i}$, tendremos también que $\pi \vDash \tau_{j,i}\mu_j \Leftarrow \beta_{x_i}$.

Vamos a definir para cada $j \in \{1..m\}$ el conjunto $V_j^{itv} = itv\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right)$. De este modo obtendremos lo siguiente:

$$\begin{array}{ll}
 & \mathcal{T}\left[\!\!\left[\overline{\left\langle \tau_j\mu_j; \Gamma'_j \right\rangle}^{\,m} \right]\!\!\right] \\
= & \bigcup_{j=1}^{m} \{(\theta, v) \mid (v,\pi) \in \mathcal{T}\,[\![\tau_j\mu_j]\!], \forall x \in \mathbf{Var}.(\theta(x),\pi) \in \mathcal{T}\,[\![\Gamma_0(x)]\!], \\
 & \quad \pi \vDash \Gamma_0|_C, \pi \vDash \left\{ \overline{\tau_{j,i}\mu_j \Leftarrow \beta_{x_i}}^{\,n} \right\}, \pi \vDash \left\{ \beta'\mu_j \subseteq \beta' \mid \beta' \in V_j^{itv} \right\} \} \\
= & \quad \left\{ \text{porque } \{\overline{\alpha_{x_i}}, \beta\} \cap ftv\left(\overline{\left\langle \tau_j\mu_j; \Gamma'_j \right\rangle}^{\,m} \right) = \emptyset \right\} \\
 & \bigcup_{j=1}^{m} \{(\theta, v) \mid (v,\pi) \in \mathcal{T}\,[\![\beta]\!], \forall x \in \mathbf{Var}.((\theta(x),\pi) \in \mathcal{T}\,[\![\alpha_x]\!], \pi \vDash \Gamma_0(x) \Leftarrow \alpha_x), \\
 & \quad \pi \vDash \Gamma_0|_C, \pi \vDash \left\{ \overline{\tau_{j,i}\mu_j \Leftarrow \beta_{x_i}}^{\,n}, \tau_j\mu_j \Leftarrow \beta \right\}, \pi \vDash \left\{ \beta'\mu_j \subseteq \beta' \mid \beta' \in V_j^{itv} \right\} \} \\
= & \quad \{ \text{porque } \Gamma_0(x_i) = \beta_{x_i} \text{ y } \beta_{x_i} \Leftarrow \alpha_{x_i} \text{ para algunas } i \} \\
 & \bigcup_{j=1}^{m} \{(\theta, v) \mid (v,\pi) \in \mathcal{T}\,[\![\beta]\!], \forall x \in \mathbf{Var}.((\theta(x),\pi) \in \mathcal{T}\,[\![\alpha_x]\!], \pi \vDash \Gamma_0(x) \Leftarrow \alpha_x), \\
 & \quad \pi \vDash \Gamma_0|_C, \pi \vDash \{ \overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^{\,n}, \tau_j\mu_j \Leftarrow \beta \}, \pi \vDash \left\{ \beta'\mu_j \subseteq \beta' \mid \beta' \in V_j^{itv} \right\} \} \\
= & \quad \{ \text{por el lema 10} \} \\
 & \{(\theta, v) \mid (v,\pi) \in \mathcal{T}\,[\![\beta]\!], \forall x \in \mathbf{Var}.((\theta(x),\pi) \in \mathcal{T}\,[\![\alpha_x]\!], \pi \vDash \Gamma_0(x) \Leftarrow \alpha_x), \\
 & \quad \pi \vDash \Gamma_0|_C, \pi \vDash \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{\,n}\right) \Leftarrow \beta \right\} \} \\
= & \mathcal{T}\left[\!\!\left[\left\langle \beta; \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{\,n}\right) \Leftarrow \beta \right\} \right\rangle \sqcap \Gamma_0 \right]\!\!\right]
\end{array}$$

demostrando por lo tanto el teorema para las reglas [APP] y [APP1].

- **Caso [CAS]**

Suponemos el juicio **case** x **of** $\overline{cls_i}^{\,n} \vdash \overline{\rho_i}^{\,n}$, obtenido con la regla de generación [CAS]. Dado un

$\Gamma[x:\alpha_x]$ cualquiera, podemos obtener el juicio $\Gamma[x:\alpha_x] \vdash$ **case** x **of** $\overline{cls_i}^n : \overline{\rho'_i}^n$ con la regla de tipado [CAS], tal que α_x es la variable de tipo específica para la variable de programa x. Por hipótesis de inducción sabemos que para cada $i \in \{1..n\}$, con el entorno $\Gamma[x:\alpha_x]$ y el juicio $cls_i \vdash_{\alpha_x} \rho_i$, existe un tipo anotado ρ'_i tal que $\Gamma[x:\alpha_x] \Vdash_{\alpha_x} cls_i : \rho'_i$ y $\mathcal{T}[\![\rho'_i]\!] \subseteq \mathcal{T}[\![\rho_i \sqcap \Gamma[x:\alpha_x]]\!]$. Por lo tanto $\mathcal{T}\left[\!\left[\overline{\rho'_i}^n\right]\!\right] \subseteq \mathcal{T}[\![\overline{\rho_i}^n \sqcap \Gamma[x:\alpha_x]]\!]$, demostrando así el teorema para las reglas [CAS].

- **Caso [RCV]**

  Suponemos el juicio **receive** $\overline{cls_i}^n$ **after** $x_t \to e \vdash \overline{\langle \tau_j; C_j \cup C_t \rangle}^m ; \overline{\langle \tau_k; C_k \cup C'_t \rangle}^h$, obtenido con la regla de generación [RCV], donde los conjuntos de restricciones $C'_t = \{\alpha_{x_t} \subseteq \texttt{integer()}\}$ y $C_t = \{\alpha_{x_t} \subseteq \texttt{integer()} \cup \texttt{'infinity'}\}$. Dado un Γ cualquiera, podemos obtener el juicio $\Gamma \vdash$ **receive** $\overline{cls_i}^n$ **after** $x_t \to e : \overline{\rho'_i}^n ; \rho'$ con la regla de tipado [RCV]. Supongamos una variable de tipo β que es fresca, por hipótesis de inducción, para cada $i \in \{1..n\}$, con el entorno Γ_c resultante de $\Gamma \sqcap [x_t : \texttt{integer()} \cup \texttt{'infinity'}]$ y el juicio $cls_i \vdash_\beta \rho_i$, obtenemos que existe un tipo anotado ρ'_i tal que $\Gamma_c \Vdash_\beta cls_i : \rho'_i$ y $\mathcal{T}[\![\rho'_i]\!] \subseteq \mathcal{T}[\![\rho_i \sqcap \Gamma_c]\!]$. De modo similar, por hipótesis de inducción, con el entorno Γ_t resultante de $\Gamma \sqcap [x_t : \texttt{integer()}]$ y el juicio $e \vdash \overline{\langle \tau_k; C_k \rangle}^h$, obtenemos que existe un tipo anotado ρ' tal que $\Gamma_t \vdash e : \rho'$ y $\mathcal{T}[\![\rho']\!] \subseteq \mathcal{T}\left[\!\left[\overline{\langle \tau_k; C_k \rangle}^h \sqcap \Gamma_t\right]\!\right]$. Teniendo en cuenta, por la regla de generación [RCV], que $\rho_1; \ldots; \rho_n = \overline{\langle \tau_j; C_j \rangle}^m$, vamos a seguir la semántica de los tipos anotados:

$$
\begin{array}{ll}
 & \mathcal{T}\left[\!\left[\overline{\langle \tau_j; C_j \cup C_t \rangle}^m ; \overline{\langle \tau_k; C_k \cup C'_t \rangle}^h \sqcap \Gamma\right]\!\right] \\
= & \mathcal{T}\left[\!\left[\overline{\langle \tau_j; C_j \cup C_t \rangle}^m \sqcap \Gamma\right]\!\right] \cup \mathcal{T}\left[\!\left[\overline{\langle \tau_k; C_k \cup C'_t \rangle}^h \sqcap \Gamma\right]\!\right] \\
= & \quad \{ \text{porque } C_t \approx [x_t : \texttt{integer()} \cup \texttt{'infinity'}] \text{ y } C'_t \approx [x_t : \texttt{integer()}] \} \\
 & \mathcal{T}\left[\!\left[\overline{\langle \tau_j; C_j \rangle}^m \sqcap \Gamma_c\right]\!\right] \cup \mathcal{T}\left[\!\left[\overline{\langle \tau_k; C_k \rangle}^h \sqcap \Gamma_t\right]\!\right] \\
= & \quad \left\{ \text{porque } \rho_1; \ldots; \rho_n = \overline{\langle \tau_j; C_j \rangle}^m \right\} \\
 & \mathcal{T}[\![(\rho_1 \sqcap \Gamma_c); \ldots; (\rho_n \sqcap \Gamma_c)]\!] \cup \mathcal{T}\left[\!\left[\overline{\langle \tau_k; C_k \rangle}^h \sqcap \Gamma_t\right]\!\right] \\
\supseteq & \quad \left\{ \text{porque } \mathcal{T}[\![\rho'_i]\!] \subseteq \mathcal{T}[\![\rho_i \sqcap \Gamma_c]\!] \text{ y } \mathcal{T}[\![\rho']\!] \subseteq \mathcal{T}\left[\!\left[\overline{\langle \tau_k; C_k \rangle}^h \sqcap \Gamma_t\right]\!\right] \right\} \\
 & \mathcal{T}[\![\rho'_1; \ldots; \rho'_n]\!] \cup \mathcal{T}[\![\rho']\!] \\
= & \mathcal{T}[\![\rho'_1; \ldots; \rho'_n; \rho']\!]
\end{array}
$$

  demostrando por lo tanto el teorema para las reglas [RCV].

- **Caso [CLS]**

  Suponemos una α y el juicio p **when** $e_g \to e \vdash_\alpha \overline{\langle \tau_{e,j}; C_j \cup \{\tau_p \Leftarrow \alpha, \texttt{'true'} \subseteq \tau_{g,j}\} \rangle \mu}^m$, obtenido con la regla de generación [CLS], donde $\{\overline{x_i}\} = vars(p)$ y $\mu = \left[\overline{\alpha_{x_i}/\beta_i}\right]$, siendo $\left\{\overline{\beta_i}\right\}$ un conjunto de variables de tipo frescas. Dado un $\Gamma\left[\overline{x_i : \texttt{any()}}\right]$ cualquiera, podemos obtener el juicio $\Gamma\left[\overline{x_i : \texttt{any()}}\right] \Vdash_\alpha p$ **when** $e_g \to e : \overline{\rho'_i \setminus \{\overline{x_i}\}}^n$ con la regla de tipado [CLS]. Usando la propiedad 18, suponiendo el entorno de entrada a $\Gamma[\overline{x_i : \alpha_i}]$, donde $\{\overline{\alpha_i}\}$ son un conjunto de variables de tipo frescas (siendo así equivalente semánticamente a $\Gamma\left[\overline{x_i : \texttt{any()}}\right]$), junto al juicio $p \vdash$

$\langle \tau_p; \emptyset \rangle$, obtenemos la existencia del tipo τ'_p tal que $\Gamma\left[\overline{x_i : \alpha_i}\right] \vdash p : \tau'_p$ y $\mathcal{T}\left[\!\left[\left\langle \tau'_p; \Gamma\left[\overline{x_i : \alpha_i}\right]\right\rangle\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\langle \tau_p; \emptyset\rangle \sqcap \Gamma\left[\overline{x_i : \alpha_i}\right]\right]\!\right]$. Suponiendo el entorno de entrada $\Gamma\left[\overline{x_i : \alpha_i} \mid \tau'_p \Leftarrow \alpha\right]$, junto al juicio $e_g \vdash \rho_g$, por hipótesis de inducción obtendremos la existencia del tipo anotado $\overline{\langle \tau'_i; \Gamma'_i \rangle}^n$ tal que $\Gamma\left[\overline{x_i : \alpha_i} \mid \tau'_p \Leftarrow \alpha\right] \vdash e_g : \overline{\langle \tau'_i; \Gamma'_i \rangle}^n$ y $\mathcal{T}\left[\!\left[\overline{\langle \tau'_i; \Gamma'_i \rangle}^n\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\rho_g \sqcap \Gamma\left[\overline{x_i : \alpha_i} \mid \tau'_p \Leftarrow \alpha\right]\right]\!\right]$. Luego, para cada $i \in \{1..n\}$, suponiendo el entorno de entrada $\Gamma'_i\left[\texttt{'true'} \subseteq \tau'_i\right]$, junto al juicio $e \vdash \rho_e$, por hipótesis de inducción obtendremos la existencia del tipo anotado ρ'_i tal que $\Gamma'_i\left[\texttt{'true'} \subseteq \tau'_i\right] \vdash e : \rho'_i$ y $\mathcal{T}\left[\!\left[\rho'_i\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\rho_e \sqcap \Gamma'_i\left[\texttt{'true'} \subseteq \tau'_i\right]\right]\!\right]$.

Con los resultados de los juicios de generación de la regla [CLS], $e_g \vdash \rho_g$ y $e \vdash \rho_e$, obtenemos que $\rho_g \otimes \rho_e = \overline{\langle \{\tau_{g,j}, \tau_{e,j}\}; C_j \rangle}^m$. Sabemos que ρ_g tiene la forma $\overline{\langle \tau_{g,k}; C_{g,k} \rangle}^s$ y ρ_e tiene la forma $\overline{\langle \tau_{e,k}; C_{e,k} \rangle}^t$. Usaremos Γ° como alias del entorno $\Gamma\left[\overline{x_i : \alpha_i}\right]$. La normalización de Γ°, que produce el siguiente resultado:

$$\begin{array}{rcl}
\mathcal{T}^{\pi}_{Env}\left[\!\left[norm(\Gamma^\circ)\right]\!\right] & = & \left\{\theta \mid \forall x \in \mathbf{Var}.\, ((\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \pi \vDash \Gamma^\circ(x) \Leftarrow \alpha_x), \pi \vDash \Gamma^\circ|_C\right\} \\
& = & \left\{\theta \mid \forall x \in \mathbf{Var}.\, (\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \forall x \in \mathbf{Var} \setminus \left\{\overline{x_i}\right\}.\pi \vDash \Gamma^\circ(x) \Leftarrow \alpha_x,\right. \\
& & \left. \quad \forall x \in \left\{\overline{x_i}\right\}.\pi \vDash \alpha_i \Leftarrow \alpha_{x_i}, \pi \vDash \Gamma^\circ|_C\right\}
\end{array}$$

donde las $\overline{x_k}$ son todas las variables en **Var**, entre las que se encuentran las $\overline{x_i}$. Siguiendo la semántica de los tipos anotados tenemos:

$$\begin{array}{rl}
& \mathcal{T}\left[\!\left[\overline{\langle \tau_{e,j}; C_j \cup \{\tau_p \Leftarrow \alpha, \texttt{'true'} \subseteq \tau_{g,j}\}\rangle}^m \sqcap \Gamma\left[\overline{x_i : \texttt{any()}}\right]\right]\!\right] \\
= & \mathcal{T}\left[\!\left[\overline{\langle \tau_{e,j}; C_j \cup \{\tau_p \Leftarrow \alpha, \texttt{'true'} \subseteq \tau_{g,j}\}\rangle}^m \sqcap \Gamma^\circ\right]\!\right] \\
= & \bigcup_{j=1}^m \mathcal{T}\left[\!\left[\langle \tau_{e,j}; C_j \cup \{\tau_p \Leftarrow \alpha, \texttt{'true'} \subseteq \tau_{g,j}\}\rangle \sqcap \Gamma^\circ\right]\!\right] \\
= & \bigcup_{j=1}^m \{(\theta, \upsilon) \mid (\upsilon, \pi) \in \mathcal{T}\left[\!\left[\tau_{e,j}\right]\!\right], \pi \vDash C_j \cup \{\{\tau_p \Leftarrow \alpha, \texttt{'true'} \subseteq \tau_{g,j}\}\}, \pi \vDash \Gamma^\circ|_C, \\
& \quad \forall x \in \mathbf{Var}.\, (\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \forall x \in \mathbf{Var} \setminus \left\{\overline{x_i}\right\}.\pi \vDash \Gamma^\circ(x) \Leftarrow \alpha_x, \forall x \in \left\{\overline{x_i}\right\}.\pi \vDash \alpha_i \Leftarrow \alpha_{x_i}\} \\
\supseteq & \quad \left\{ \text{porque } \mathcal{T}\left[\!\left[\left\langle \tau'_p; \Gamma\left[\overline{x_i : \alpha_i}\right]\right\rangle\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\langle \tau_p; \emptyset\rangle \sqcap \Gamma\left[\overline{x_i : \alpha_i}\right]\right]\!\right] \right\} \\
& \bigcup_{j=1}^m \left\{(\theta, \upsilon) \mid \theta \in \mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma^\circ\right]\!\right], \pi \vDash \tau'_p \Leftarrow \alpha, (\upsilon, \pi) \in \mathcal{T}\left[\!\left[\tau_{e,j}\right]\!\right], \pi \vDash C_j, \right. \\
& \quad \left. \pi \vDash \texttt{'true'} \subseteq \tau_{g,j}, \forall x \in \mathbf{Var}.\, ((\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \pi \vDash \Gamma^\circ(x) \Leftarrow \alpha_x)\right\} \\
\supseteq & \quad \left\{ \text{con } (\upsilon', \pi) \in \mathcal{T}\left[\!\left[\tau_{g,j}\right]\!\right] \text{ tenemos el resultado de } \rho_g \otimes \rho_e \right\} \\
& \bigcup_{j=1}^m \left\{(\theta, \upsilon) \mid \theta \in \mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma^\circ\right]\!\right], \pi \vDash \tau'_p \Leftarrow \alpha, \left(\left(\{\overset{n}{\cdots}\}, \upsilon', \upsilon\right), \pi\right) \in \mathcal{T}\left[\!\left[\{\tau_{g,j}, \tau_{e,j}\}\right]\!\right], \pi \vDash C_j, \right. \\
& \quad \left. \pi \vDash \texttt{'true'} \subseteq \tau_{g,j}, \forall x \in \mathbf{Var}.\, ((\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \pi \vDash \Gamma^\circ(x) \Leftarrow \alpha_x)\right\} \\
= & \quad \left\{ \text{por la definición de } \rho_g \otimes \rho_e \text{ y la propiedad 17} \right\} \\
& \bigcup_{j=1}^s \bigcup_{k=1}^t \left\{(\theta, \upsilon) \mid \theta \in \mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma^\circ\right]\!\right], \pi \vDash \tau'_p \Leftarrow \alpha, \left(\upsilon', \pi\right) \in \mathcal{T}\left[\!\left[\tau_{g,j}\right]\!\right], \pi \vDash C_{g,j}, (\upsilon, \pi) \in \mathcal{T}\left[\!\left[\tau_{e,k}\right]\!\right], \right. \\
& \quad \left. \pi \vDash C_{e,k}, \pi \vDash \texttt{'true'} \subseteq \tau_{g,j}, \forall x \in \mathbf{Var}.\, ((\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \pi \vDash \Gamma^\circ(x) \Leftarrow \alpha_x)\right\} \\
\supseteq & \quad \left\{ \text{porque } \mathcal{T}\left[\!\left[\overline{\langle \tau'_i; \Gamma'_i \rangle}^n\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\rho_g \sqcap \Gamma\left[\overline{x_i : \alpha_i} \mid \tau'_p \Leftarrow \alpha\right]\right]\!\right] \right\} \\
& \bigcup_{i=1}^n \bigcup_{k=1}^t \{(\theta, \upsilon) \mid \left(\upsilon', \pi\right) \in \mathcal{T}\left[\!\left[\tau'_i\right]\!\right], \theta \in \mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma'_i\right]\!\right], \pi \vDash \texttt{'true'} \subseteq \tau'_i, (\upsilon, \pi) \in \mathcal{T}\left[\!\left[\tau_{e,k}\right]\!\right], \\
& \quad \pi \vDash C_{e,k}, \forall x \in \mathbf{Var}.\, ((\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \pi \vDash \Gamma^\circ(x) \Leftarrow \alpha_x)\} \\
\supseteq & \quad \left\{ \text{porque } \mathcal{T}\left[\!\left[\rho'_i\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\rho_e \sqcap \Gamma'_i\left[\texttt{'true'} \subseteq \tau'_i\right]\right]\!\right] \right\} \\
& \bigcup_{i=1}^n \mathcal{T}\left[\!\left[\rho'_i\right]\!\right]
\end{array}$$

Entonces, usando el renombramiento de variables de tipo $\mu = \left[\overline{\alpha_{x_i}/\beta_i}\right]$, separamos estas variables de tipo del tipo que cada $\overline{x_i}$ tiene en el entorno normalizado. Además, aplicando $\overline{\rho'_i \setminus \{\overline{x_i}\}}^n$, el resultado es:

$$\mathcal{T}\left[\!\left[\overline{\rho'_i \setminus \{\overline{x_i}\}}^n\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\overline{\langle \tau_{e,j}; C_j \cup \{\tau_p \Leftarrow \alpha, \texttt{'true'} \subseteq \tau_{g,j}\}\rangle\mu}^m \sqcap \Gamma\left[\overline{x_i : \text{any}()}\right]\right]\!\right]$$

demostrando así el teorema para las reglas [CLS].

- **Caso [LET]**

Suponemos el juicio **let** $x = e_1$ **in** $e_2 \vdash \overline{\langle \tau_{i,2}; C_i \cup \{\tau_{i,1} \Leftarrow \alpha_x\}\rangle\mu}^n$, obtenido con la regla de generación [LET], donde $\mu = [\alpha_x/\beta]$, siendo β una variable de tipo fresca. Dado un $\Gamma[x : \text{any}()]$ cualquiera, podemos obtener el juicio $\Gamma[x : \text{any}()] \vdash$ **let** $x = e_1$ **in** $e_2 : \overline{\rho'_i \setminus \{x\}}^m$ con la regla de tipado [LET]. Por hipótesis de inducción, con el entorno $\Gamma[x : \text{any}()]$ y el juicio $e_1 \vdash \rho_1$, obtenemos que existe un tipo anotado $\overline{\langle \tau'_i; \Gamma_i\rangle}^m$ tal que $\Gamma[x : \text{any}()] \vdash e_1 : \overline{\langle \tau'_i; \Gamma_i\rangle}^m$ y $\mathcal{T}\left[\!\left[\overline{\langle \tau'_i; \Gamma_i\rangle}^m\right]\!\right] \subseteq \mathcal{T}[\![\rho_1 \sqcap \Gamma[x : \text{any}()]]\!]$. De nuevo por hipótesis de inducción, para $i \in \{1..m\}$, con el entorno $\Gamma_i[x : \tau'_i]$ y el juicio $e_2 \vdash \rho_2$, obtenemos que existe un tipo anotado ρ'_i tal que $\Gamma_i[x : \tau'_i] \vdash e_2 : \rho'_i$ y $\mathcal{T}[\![\rho'_i]\!] \subseteq \mathcal{T}[\![\rho_2 \sqcap \Gamma_i[x : \tau'_i]]\!]$. También sabemos por la regla de generación [LET] que $\rho_1 \otimes \rho_2 = \overline{\langle \{\tau_{i,1}, \tau_{i,2}\}; C_i\rangle}^n$. Sabemos que ρ_1 tiene la forma $\overline{\langle \tau^1_k; C^1_k\rangle}^s$ y ρ_2 tiene la forma $\overline{\langle \tau^2_k; C^2_k\rangle}^t$. Vamos a denominar al entorno $\Gamma[x : \text{any}()]$ como Γ'. Siguiendo la semántica de los tipos anotados tenemos:

$$\begin{aligned}
& \mathcal{T}\left[\!\left[\overline{\langle \tau_{i,2}; C_i \cup \{\tau_{i,1} \Leftarrow \alpha_x\}\rangle}^n \sqcap \Gamma'\right]\!\right] \\
= \;& \bigcup_{i=1}^n \mathcal{T}[\![\langle \tau_{i,2}; C_i \cup \{\tau_{i,1} \Leftarrow \alpha_x\}\rangle \sqcap \Gamma']\!] \\
= \;& \bigcup_{i=1}^n \{(\theta, \nu) \mid \theta \in \mathcal{T}^\pi_{Env}[\![\Gamma']\!], (\nu, \pi) \in \mathcal{T}[\![\tau_{i,2}]\!], \pi \vDash C_i, \pi \vDash \tau_{i,1} \Leftarrow \alpha_x, \\
& \qquad \forall x \in \textbf{Var}.\left((\theta(x), \pi) \in \mathcal{T}[\![\alpha_x]\!], \pi \vDash \Gamma'(x) \Leftarrow \alpha_x\right)\} \\
= \;& \qquad \{\text{ con } (\nu', \pi) \in \mathcal{T}[\![\tau_{i,1}]\!] \text{ tenemos el resultado de } \rho_g \otimes \rho_e \} \\
& \bigcup_{i=1}^n \left\{(\theta, \nu) \mid \theta \in \mathcal{T}^\pi_{Env}[\![\Gamma']\!], (\left(\{\overset{n}{\ldots}\}, \nu', \nu\right), \pi) \in \mathcal{T}[\![\{\tau_{i,1}, \tau_{i,2}\}]\!], \pi \vDash C_i, \right. \\
& \qquad \left. \pi \vDash \tau_{i,1} \Leftarrow \alpha_x, \forall x \in \textbf{Var}.\left((\theta(x), \pi) \in \mathcal{T}[\![\alpha_x]\!], \pi \vDash \Gamma'(x) \Leftarrow \alpha_x\right)\right\} \\
= \;& \qquad \{\text{ por la definición de } \rho_1 \otimes \rho_2 \text{ y la propiedad 17 } \} \\
& \bigcup_{j=1}^s \bigcup_{k=1}^t \left\{(\theta, \nu) \mid \theta \in \mathcal{T}^\pi_{Env}[\![\Gamma']\!], (\nu', \pi) \in \mathcal{T}\left[\!\left[\tau^1_j\right]\!\right], \pi \vDash C^1_j, (\nu, \pi) \in \mathcal{T}[\![\tau^2_k]\!], \right. \\
& \qquad \left. \pi \vDash C^2_k, \pi \vDash \tau^1_j \Leftarrow \alpha_x, \forall x \in \textbf{Var}.\left((\theta(x), \pi) \in \mathcal{T}[\![\alpha_x]\!], \pi \vDash \Gamma'(x) \Leftarrow \alpha_x\right)\right\} \\
\supseteq \;& \qquad \left\{\text{ porque } \mathcal{T}\left[\!\left[\overline{\langle \tau'_i; \Gamma_i\rangle}^m\right]\!\right] \subseteq \mathcal{T}[\![\rho_1 \sqcap \Gamma[x : \text{any}()]]\!] \right\} \\
& \bigcup_{i=1}^m \bigcup_{k=1}^t \{(\theta, \nu) \mid \theta \in \mathcal{T}^\pi_{Env}[\![\Gamma_i]\!], \pi \vDash \tau'_i \Leftarrow \alpha_x, (\nu', \pi) \in \mathcal{T}[\![\tau'_i]\!], (\nu, \pi) \in \mathcal{T}[\![\tau^2_k]\!], \\
& \qquad \pi \vDash C^2_k, \forall x \in \textbf{Var}.\left((\theta(x), \pi) \in \mathcal{T}[\![\alpha_x]\!], \pi \vDash \Gamma'(x) \Leftarrow \alpha_x\right)\} \\
\supseteq \;& \qquad \{\text{ porque } \mathcal{T}[\![\rho'_i]\!] \subseteq \mathcal{T}[\![\rho_2 \sqcap \Gamma_i[x : \tau'_i]]\!] \text{ para cada } i \in \{1..m\} \} \\
& \bigcup_{i=1}^m \mathcal{T}[\![\rho'_i]\!]
\end{aligned}$$

Con el renombramiento de variables μ y el borrado de variables en el entorno de salida, obten-

dremos lo siguiente:

$$\mathcal{T}\left[\!\left[\overline{\rho'_i \setminus \{x\}}^{m}\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\overline{\langle \tau_{i,2}; C_i \cup \{\tau_{i,1} \Leftarrow \alpha_x\}\rangle\mu}^{n} \sqcap \Gamma\,[x : \mathsf{any}()]\right]\!\right]$$

demostrando así el teorema para las reglas [LET].

- **Caso [LRC]**

Suponemos el juicio **letrec** $\overline{x_i = f_i}^{n}$ **in** $e \vdash \overline{\langle \tau_j; C_j \cup \{\overline{\tau_i \Leftarrow \alpha_{x_i}}^{n}\}\rangle\mu}^{m}$, obtenido con la regla de generación [LRC], donde $\mu = \left[\overline{\alpha_{x_i}/\beta_i}^{n}\right]$, siendo $\beta_1, \ldots, \beta_n$ variables de tipo frescas. Dado un entorno $\Gamma\left[\overline{x_i : \mathsf{any}()}^{n}\right]$ cualquiera, podemos obtener el juicio $\Gamma\left[\overline{x_i : \mathsf{any}()}^{n}\right] \vdash$ **letrec** $\overline{x_i = f_i}^{n}$ **in** $e : \rho' \setminus \{\overline{x_i}^{n}\}$ con la regla de tipado [LRC]. Por hipótesis de inducción, para cada $i \in \{1..n\}$, con el entorno $\Gamma\left[\overline{x_i : \tau'_i}^{n}\right]$ y el juicio $f_i \vdash \langle \tau_i; \emptyset\rangle$, obtenemos que existe un tipo anotado $\left\langle \tau'_i; \Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right\rangle$ tal que $\Gamma\left[\overline{x_i : \tau'_i}^{n}\right] \vdash f_i : \left\langle \tau'_i; \Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right\rangle$ y $\mathcal{T}\left[\!\left[\left\langle \tau'_i; \Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right\rangle\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\langle \tau_i; \emptyset\rangle \sqcap \Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right]\!\right]$. Por hipótesis de inducción, de nuevo con el entorno $\Gamma\left[\overline{x_i : \tau'_i}^{n}\right]$ y el juicio $e \vdash \overline{\langle \tau_j; C_j\rangle}^{m}$, obtenemos que existe un tipo anotado ρ' tal que $\Gamma\left[\overline{x_i : \tau'_i}^{n}\right] \vdash e : \rho'$ y $\mathcal{T}\left[\!\left[\rho'\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\overline{\langle \tau_j; C_j\rangle}^{m} \sqcap \Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right]\!\right]$. Siguiendo la semántica de los tipos anotados tenemos:

$$\begin{array}{ll}
 & \mathcal{T}\left[\!\left[\overline{\langle \tau_j; C_j \cup \{\overline{\tau_i \Leftarrow \alpha_{x_i}}^{n}\}\rangle}^{m} \sqcap \Gamma\left[\overline{x_i : \mathsf{any}()}^{n}\right]\right]\!\right] \\
= & \bigcup_{j=1}^{m} \left\{(\theta, v) \mid \theta \in \mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma\left[\overline{x_i : \mathsf{any}()}^{n}\right]\right]\!\right], \pi \vDash \{\overline{\tau_i \Leftarrow \alpha_{x_i}}^{n}\}, (v, \pi) \in \mathcal{T}\left[\!\left[\tau_j\right]\!\right],\right. \\
 & \qquad \left. \pi \vDash C_j, \forall x \in \mathbf{Var}.\left((\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \pi \vDash \left(\Gamma\left[\overline{x_i : \mathsf{any}()}^{n}\right]\right)(x) \Leftarrow \alpha_x\right)\right\} \\
\supseteq & \qquad \left\{ \text{porque } \Gamma\left[\overline{x_i : \tau'_i}^{n}\right] \subseteq \Gamma\left[\overline{x_i : \mathsf{any}()}^{n}\right] \right\} \\
 & \bigcup_{j=1}^{m} \left\{(\theta, v) \mid \theta \in \mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right]\!\right], \pi \vDash \{\overline{\tau_i \Leftarrow \alpha_{x_i}}^{n}\}, (v, \pi) \in \mathcal{T}\left[\!\left[\tau_j\right]\!\right],\right. \\
 & \qquad \left. \pi \vDash C_j, \forall x \in \mathbf{Var}.\left((\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \pi \vDash \left(\Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right)(x) \Leftarrow \alpha_x\right)\right\} \\
\supseteq & \qquad \left\{ \text{porque } \mathcal{T}\left[\!\left[\left\langle \tau'_i; \Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right\rangle\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\langle \tau_i; \emptyset\rangle \sqcap \Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right]\!\right] \right\} \\
 & \bigcup_{j=1}^{m} \left\{(\theta, v) \mid \theta \in \mathcal{T}^{\pi}_{Env}\left[\!\left[\Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right]\!\right], \pi \vDash \left\{\overline{\tau'_i \Leftarrow \alpha_{x_i}}^{n}\right\}, (v, \pi) \in \mathcal{T}\left[\!\left[\tau_j\right]\!\right],\right. \\
 & \qquad \left. \pi \vDash C_j, \forall x \in \mathbf{Var}.\left((\theta(x), \pi) \in \mathcal{T}\left[\!\left[\alpha_x\right]\!\right], \pi \vDash \left(\Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right)(x) \Leftarrow \alpha_x\right)\right\} \\
\supseteq & \qquad \left\{ \text{porque } \mathcal{T}\left[\!\left[\rho'\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\overline{\langle \tau_j; C_j\rangle}^{m} \sqcap \Gamma\left[\overline{x_i : \tau'_i}^{n}\right]\right]\!\right] \right\} \\
 & \mathcal{T}\left[\!\left[\rho'\right]\!\right]
\end{array}$$

Con el renombramiento de variables μ y el borrado de variables en el entorno de salida, obtendremos lo siguiente:

$$\mathcal{T}\left[\!\left[\overline{\langle \tau_j; C_j \cup \{\overline{\tau_i \Leftarrow \alpha_{x_i}}^{n}\}\rangle\mu}^{m} \sqcap \Gamma\left[\overline{x_i : \mathsf{any}()}^{n}\right]\right]\!\right] \subseteq \mathcal{T}\left[\!\left[\rho' \setminus \{\overline{x_i}^{n}\}\right]\!\right]$$

demostrando así el teorema para las reglas [LRC].

□

5.3. Operaciones fundamentales con los tipos

En esta sección vamos a describir una serie de operaciones que son necesarias para poder trabajar con el algoritmo de inferencia, sobre todo a la hora de transformar los tipos.

5.3.1. Cotas superiores *ground*

En algunos casos necesitamos conocer la estructura de los valores contenidos en un tipo polimórfico, independientemente de su relación con el contexto en el que se calculan. Por ejemplo, a veces necesitamos saber si un tipo polimórfico dado contiene solamente valores de tipo lista, de tipo tupla, o de tipo entero. Para ello nos puede ser útil dar una cota superior monomórfica de un tipo polimórfico cualquiera, suponiendo que conocemos las cotas de las variables de tipo inducidas por las restricciones de los tipos anotados.

Introducimos el concepto de *función de cota ground* como una función parcial $\phi : \mathbf{TypeVar} \to \mathbf{Type}^\bullet$, donde $\mathbf{Type}^\bullet$ denota el conjunto de tipos monomórficos (tipo *ground*). La semántica de una función de cota *ground* se define como la mayor instanciación π que asocia cada variable de tipo con valores contenidos dentro del tipo *ground* correspondiente:

$$[\![\phi]\!] = \pi \text{ tal que } \forall \alpha \in \mathbf{TypeVar} : \pi(\alpha) = \mathcal{T}\,[\![\phi(\alpha)]\!]\big|_1$$

Además definimos una relación de orden entre funciones de cota *ground* mediante la relación de orden habitual entre funciones parciales. Para definir funciones de cota *ground* utilizamos la notación habitual de funciones parciales. En particular, utilizamos $[\,]$ para denotar la función ϕ tal que $\phi(\alpha) =$ `any()` para todo $\alpha \in \mathbf{TypeVar}$, y utilizamos $\phi[\alpha \mapsto \tau]$ para denotar la función ϕ' tal que $\phi'(\alpha) = \tau$ y $\phi'(\beta) = \phi(\beta)$ para cualquier otro $\beta \neq \alpha$.

Para poder trabajar con estos nuevos conceptos definimos dos funciones:

- **Menor cota *ground* (*lug*)**. La expresión $lug\ [\![\tau]\!]\ \phi$ devuelve la menor cota *ground* de un tipo τ bajo la suposición de que las variables contenidas en τ están acotadas por ϕ. Su definición está en la figura 5.4.
- **Mayor solución (*gsol*)**. La expresión $gsol\ [\![C]\!]\ \phi$ devuelve la mayor función de cota *ground* que satisface un conjunto de restricciones y que sea menor que la función de cota *ground* ϕ pasada como parámetro. Su definición está en la figura 5.5.

Ambas funciones cumplen la siguiente propiedad:

Proposición 19. *Supongamos unos tipos τ y τ', una restricción φ, un conjunto de restricciones C, y una función de cota* ground *ϕ y ϕ', tales que $lug\ [\![\tau]\!]\ \phi = \tau'$, $gsol\ [\![\varphi]\!]\ \phi = \phi'$ y $gsol\ [\![C]\!]\ \phi = \phi'$. Entonces, para cada $v \in \mathbf{DVal}$ y $\pi \in \mathbf{TypeInst}$:*

$$
\begin{array}{rcl}
lug\ [\![\texttt{none()}]\!]\ \phi & = & \texttt{none()} \\
lug\ [\![\texttt{any()}]\!]\ \phi & = & \texttt{any()} \\
lug\ [\![B]\!]\ \phi & = & B \\
lug\ [\![c]\!]\ \phi & = & c \\
lug\ [\![\alpha]\!]\ \phi & = & \phi(\alpha) \\
lug\ [\![\tau\ \textbf{when}\ C]\!]\ \phi & = & \begin{cases} \texttt{none()} & \text{si } gsol\ [\![C]\!]\ \phi = \bot \\ lug\ [\![\tau]\!]\ (gsol\ [\![C]\!]\ \phi) & \text{e.o.c.} \end{cases} \\
lug\ [\![\tau_1 \cup \tau_2]\!]\ \phi & = & lug\ [\![\tau_1]\!]\ \phi \cup lug\ [\![\tau_2]\!]\ \phi \\
lug\ [\![\{\tau_1,\ldots,\tau_n\}]\!]\ \phi & = & \{lug\ [\![\tau_1]\!]\ \phi,\ldots,lug\ [\![\tau_n]\!]\ \phi\} \\
lug\ [\![\texttt{nelist}(\tau_1,\tau_2)]\!]\ \phi & = & \texttt{nelist}(lug\ [\![\tau_1]\!]\ \phi, lug\ [\![\tau_2]\!]\ \phi) \\
lug\ \left[\!\left[\bigsqcup_{i=1}^{n} \sigma_i\right]\!\right]\ \phi & = & \bigsqcup_{i=1}^{n} lug\ [\![\sigma_i]\!]\ \phi \\
lug\ \left[\!\left[\forall \overline{\alpha_j}.(\overline{\tau_i}) \to \tau\right]\!\right]\ \phi & = & \left(\overline{lug\ [\![\tau_i]\!]\ \phi'}\right) \to lug\ [\![\tau]\!]\ \phi' \text{ donde } \phi' = \phi\left[\overline{\alpha_i \mapsto \texttt{any()}}\right]
\end{array}
$$

Figura 5.4: Definición de la función del menor tipo *ground* superior

$$
\begin{array}{rcl}
gsol\ [\![\{\varphi_1,\ldots,\varphi_n\}]\!]\ \phi & = & fix_\phi\left(gsol\ [\![\varphi_n]\!] \circ \ldots \circ gsol\ [\![\varphi_1]\!]\ \right) \\
gsol\ [\![\alpha \subseteq \tau]\!]\ \phi & = & \phi \sqcap \left[\alpha \mapsto lug\ [\![\tau]\!]\ \phi\right] \\
gsol\ [\![c \subseteq \tau]\!]\ \phi & = & \begin{cases} \bot & \text{si } c \notin lug\ [\![\tau]\!]\ \phi \\ \phi & \text{e.o.c.} \end{cases} \\
gsol\ [\![\tau \Leftarrow \alpha]\!]\ \phi & = & \phi \sqcap \left[\alpha \mapsto lug\ [\![\tau]\!]\ \phi\right]
\end{array}
$$

Figura 5.5: Definición de la *mayor solución* para restricciones

- $(v,\pi) \in \mathcal{T}\,[\![\tau]\!] \wedge \pi \subseteq [\![\phi]\!] \Longrightarrow (v,\pi) \in \mathcal{T}\,[\![\tau']\!]$
- $\pi \vDash \varphi \wedge \pi \subseteq [\![\phi]\!] \Longrightarrow \pi \subseteq [\![\phi']\!]$
- $\pi \vDash C \wedge \pi \subseteq [\![\phi]\!] \Longrightarrow \pi \subseteq [\![\phi']\!]$

Demostración. Por inducción sobre la estructura de τ, C o φ. Demostraremos la propiedad caso por caso:

- **Caso** $\tau = \texttt{none()}$

  En este caso no existe ningún valor v ni ninguna instanciación π tal que $(v,\pi) \in \mathcal{T}\,[\![\texttt{none()}]\!]$, por lo que la proposición se cumple trivialmente.

- **Caso** $\tau = \texttt{any()}$

  En este caso $\tau' = \texttt{any()}$, por lo que siempre se cumple que $(v,\pi) \in \mathcal{T}\,[\![\tau']\!]$, demostrando así la proposición.

- **Caso** $\tau = B$

  Suponemos un par $(v,\pi) \in \mathcal{T}\,[\![B]\!]$. Por lo tanto, se cumple que $(v,\pi) \in \mathcal{T}\,[\![B]\!] = \mathcal{T}\,[\![\tau']\!]$, demostrando así la proposición.

- **Caso** $\tau = c$

  Suponemos un par $(c,\pi) \in \mathcal{T}\,[\![c]\!]$. Por lo tanto, se cumple que $(c,\pi) \in \mathcal{T}\,[\![c]\!] = \mathcal{T}\,[\![\tau']\!]$, demostrando así la proposición.

- **Caso** $\tau = \alpha$

  Sea un par $(v,\pi) \in \mathcal{T}\,[\![\alpha]\!]$, tal que $\pi \subseteq [\![\phi]\!]$. De la suposición $\pi \subseteq [\![\phi]\!]$, se obtiene que $\pi(\alpha) \subseteq \mathcal{T}\,[\![\phi(\alpha)]\!]\big|_1$. Como sabemos que $\pi(\alpha) = \{v\}$, entonces $v \in \mathcal{T}\,[\![\phi(\alpha)]\!]\big|_1$ y por lo tanto se cumple que $(v,\pi) \in \mathcal{T}\,[\![\phi(\alpha)]\!] = \mathcal{T}\,[\![\tau']\!]$, demostrando así la proposición.

- **Caso** $\tau = \tau''$ **when** C

  Sea un par $(v,\pi) \in \mathcal{T}\,[\![\tau''$ **when** $C]\!]$, tal que $\pi \subseteq [\![\phi]\!]$. Tenemos dos casos posibles:

  - **Caso** $gsol\ [\![C]\!]\ \phi = \bot$

    En este caso no existen un valor v y una instanciación π tal que $(v,\pi) \in \mathcal{T}\,[\![\tau]\!]$, por lo que se demuestra trivialmente que $\mathcal{T}\,[\![\tau']\!]$ es $\mathcal{T}\,[\![\texttt{none()}]\!]$.

  - **Caso** $gsol\ [\![C]\!]\ \phi \neq \bot$

    Entonces, tenemos que $(v,\pi) \in \mathcal{T}\,[\![\tau']\!]$ y $\pi \vDash C$. Por hipótesis de inducción, con $\pi \vDash C$ obtenemos que $\pi \subseteq [\![gsol\ [\![C]\!]\ \phi]\!]$, por lo tanto, aplicando la hipótesis de inducción de nuevo, junto a $(v,\pi) \in \mathcal{T}\,[\![\tau']\!]$ obtenemos que se cumple $(v,\pi) \in \subseteq \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ (gsol\ [\![C]\!]\ \phi)]\!] = \mathcal{T}\,[\![\tau']\!]$, demostrando así la proposición.

- **Caso** $\tau = \tau_1 \cup \tau_2$

  Sea un par $(v,\pi) \in \mathcal{T}\,[\![\tau_1 \cup \tau_2]\!]$, tal que $\pi \subseteq [\![\phi]\!]$. Tenemos dos casos posibles para (v,π) dentro de la unión $\tau_1 \cup \tau_2$:

  - **Caso** $(v,\pi') \in \mathcal{T}\,[\![\tau_1]\!]$

    Donde $\pi' = \pi \setminus \big(itv(\tau_2) \setminus ftv(\tau_1)\big)$, cumpliéndose que $\pi' \subseteq \pi$. Por lo tanto, se cumple que $\pi' \subseteq [\![\phi]\!]$. Si aplicamos la hipótesis de inducción, obtenemos que $(v,\pi') \in \mathcal{T}\,[\![lug\ [\![\tau_1]\!]\ \phi]\!]$. Como sabemos que $lug\ [\![\tau_1]\!]\ \phi$ es *ground*, tenemos que $(v,\pi) \in \mathcal{T}\,[\![lug\ [\![\tau_1]\!]\ \phi]\!]$

  - **Caso** $(v,\pi') \in \mathcal{T}\,[\![\tau_2]\!]$

    Donde $\pi' = \pi \setminus \big(itv(\tau_1) \setminus ftv(\tau_2)\big)$, cumpliéndose que $\pi' \subseteq \pi$. Por lo tanto, se cumple que $\pi' \subseteq [\![\phi]\!]$. Si aplicamos la hipótesis de inducción, obtenemos que $(v,\pi') \in \mathcal{T}\,[\![lug\ [\![\tau_2]\!]\ \phi]\!]$. Como sabemos que $lug\ [\![\tau_2]\!]\ \phi$ es *ground*, tenemos que $(v,\pi) \in \mathcal{T}\,[\![lug\ [\![\tau_2]\!]\ \phi]\!]$

  Ahora con la unión de $\mathcal{T}\,[\![lug\ [\![\tau_1]\!]\ \phi]\!]$ y $\mathcal{T}\,[\![lug\ [\![\tau_2]\!]\ \phi]\!]$ obtenemos que

  $$(v,\pi) \in \mathcal{T}\,[\![lug\ [\![\tau_1]\!]\ \phi \cup lug\ [\![\tau_2]\!]\ \phi]\!] = \mathcal{T}\,[\![\tau']\!]$$

  demostrando así la proposición.

- **Caso** $\tau = \{\tau_1, \ldots, \tau_n\}$

  Sea un par $(v,\pi) \in \mathcal{T}\,[\![\{\tau_1, \ldots, \tau_n\}]\!]$, tal que $\pi \subseteq [\![\phi]\!]$. Sabemos que $v = \left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right)$ por la definición semántica de las tuplas, por lo tanto:

  $$\begin{array}{c} \forall i \in \{1..n\}\,.\,(v_i,\pi) \in \mathcal{T}\,[\![\tau_i]\!] \\ \Downarrow \qquad \{\text{ por h.i. }\} \\ \forall i \in \{1..n\}\,.\,(v_i,\pi) \in \mathcal{T}\,[\![lug\ [\![\tau_i]\!]\ \phi]\!] \\ \Updownarrow \\ (v,\pi) \in \mathcal{T}\,[\![\{lug\ [\![\tau_1]\!]\ \phi, \ldots, lug\ [\![\tau_n]\!]\ \phi\}]\!] = \mathcal{T}\,[\![\tau']\!] \end{array}$$

  demostrando así la proposición.

- **Caso** $\tau = \texttt{nelist}(\tau_1, \tau_2)$

  Sea un par $(v,\pi) \in \mathcal{T}\,[\![\texttt{nelist}(\tau_1,\tau_2)]\!]$, tal que $\pi \subseteq [\![\phi]\!]$. Sabemos que $v = [v_1, \ldots, v_n \mid v']$ por la definición semántica de las listas no vacías, y que existen las instanciaciones $\pi_1, \ldots, \pi_n$, tales que $(v_i,\pi_i) \in \mathcal{T}\,[\![\tau_1]\!]$, para cada $i \in \{1..n\}$, y $(v',\pi) \in \mathcal{T}\,[\![\tau_2]\!]$, donde $\pi \in Dcp\big(\{\overline{\pi_i}^n\}, \tau_1\big)$. Sabemos que, para cada $i \in \{1..n\}$, se cumple que $\pi_i \subseteq \pi$ porque $\pi = \bigcup_{i=1}^{n} \pi_i$, por lo tanto $\pi_i \subseteq \pi \subseteq [\![\phi]\!]$. Entonces:

  $$\begin{array}{c} \forall i \in \{1..n\}\,.\,(v_i,\pi_i) \in \mathcal{T}\,[\![\tau_1]\!] \qquad \wedge \qquad \pi_i \subseteq [\![\phi]\!] \\ \Downarrow \qquad \{\text{ por h.i. }\} \\ \forall i \in \{1..n\}\,.\,(v_i,\pi_i) \in \mathcal{T}\,[\![lug\ [\![\tau_1]\!]\ \phi]\!] \end{array}$$

Como $lug\ [\![\tau_1]\!]\ \phi$ es un tipo *ground*, obtenemos que $(v_i, \pi) \in \mathcal{T}\ [\![lug\ [\![\tau_1]\!]\ \phi]\!]$, para cada $i \in \{1..n\}$. Con esto último se cumple que $\pi \in Dcp(\{\pi\}, lug\ [\![\tau_1]\!]\ \phi)$. Además, por hipótesis de inducción, con $(v', \pi) \in \mathcal{T}\ [\![\tau_2]\!]$ y $\pi \subseteq [\![\phi]\!]$, obtenemos que $(v', \pi) \in \mathcal{T}\ [\![lug\ [\![\tau_2]\!]\ \phi]\!]$. Por lo tanto:

$$([v_1, \ldots, v_n \mid v'], \pi) \in \mathcal{T}\ [\![\texttt{nelist(}lug\ [\![\tau_1]\!]\ \phi\texttt{,}\ lug\ [\![\tau_2]\!]\ \phi\texttt{)}]\!] = \mathcal{T}\ [\![\tau']\!]$$

demostrando así la proposición.

- **Caso** $\tau = (\overline{\tau_i}^n) \to \tau_0$

  Sea un par $(v, \pi) \in \mathcal{T}\ [\![(\overline{\tau_i}^n) \to \tau_0]\!]$, tal que $\pi \subseteq [\![\phi]\!]$. Hay dos casos posibles. El primero, si $v = \emptyset$, la demostración es trivial, demostrando la proposición con:

  $$(\emptyset, \pi) \in \mathcal{T}\ \left[\!\left[\left(\overline{lug\ [\![\tau_i]\!]\ \phi}^n\right) \to lug\ [\![\tau_0]\!]\ \phi\right]\!\right] = \mathcal{T}\ [\![\tau']\!]$$

  El segundo caso, si $v \neq \emptyset$, existe un conjunto de instanciaciones $\{\pi_w \mid \pi_w \in v\}$, tales que, para todo $w \in v$, si $w = ((v_1, \ldots, v_n), v_0)$, tendremos que:

  $$\forall i \in \{1..n\}.\, (v_i, \pi_w) \in \mathcal{T}\ [\![\tau_i]\!] \qquad (v_0, \pi_w) \in \mathcal{T}\ [\![\tau_0]\!] \qquad \pi \in Dcp(\{\pi_w \mid \pi_w \in v\}, (\overline{\tau_i}^n) \to \tau_0) \quad (5.2)$$

  Como $\pi = \bigcup_{w \in v} \pi_w$, se cumple que $\pi_w \subseteq \pi \subseteq [\![\phi]\!]$, para cada $w \in v$. Por lo tanto podemos aplicar la hipótesis de inducción con (5.2), y como para todo $w \in v$, si $w = ((v_1, \ldots, v_n), v_0)$, obtendremos:

  $$\forall i \in \{1..n\}.\, (v_i, \pi_w) \in \mathcal{T}\ [\![lug\ [\![\tau_i]\!]\ \phi]\!] \qquad (v_0, \pi_w) \in \mathcal{T}\ [\![lug\ [\![\tau_0]\!]\ \phi]\!]$$
  $$\pi \in Dcp\left(\{\pi_w \mid \pi_w \in v\}, \left(\overline{lug\ [\![\tau_i]\!]\ \phi}^n\right) \to lug\ [\![\tau_0]\!]\ \phi\right)$$

  y, por ser $lug\ [\![\tau_i]\!]\ \phi$, para cada $i \in \{1..n\}$, y $lug\ [\![\tau_0]\!]\ \phi$ tipos *ground*, obtendremos lo siguiente:

  $$\forall i \in \{1..n\}.\, (v_i, \pi) \in \mathcal{T}\ [\![lug\ [\![\tau_i]\!]\ \phi]\!] \qquad (v_0, \pi) \in \mathcal{T}\ [\![lug\ [\![\tau_0]\!]\ \phi]\!]$$
  $$\pi \in Dcp\left(\{\pi\}, \left(\overline{lug\ [\![\tau_i]\!]\ \phi}^n\right) \to lug\ [\![\tau_0]\!]\ \phi\right)$$

  Este último resultado nos permite tener

  $$(v, \pi) \in \mathcal{T}\ \left[\!\left[\left(\overline{lug\ [\![\tau_i]\!]\ \phi}^n\right) \to lug\ [\![\tau_0]\!]\ \phi\right]\!\right] = \mathcal{T}\ [\![\tau']\!]$$

  demostrando así la proposición.

- **Caso** $\tau = \forall \overline{\alpha_i}.\tau''$

  Sea un par $(v, \pi) \in \mathcal{T}\ [\![\forall \overline{\alpha_i}.\tau'']\!]$, tal que $\pi \subseteq [\![\phi]\!]$, entonces tenemos que $\exists \pi' \equiv \pi$ (módulo $\{\overline{\alpha_i}\}$), tal que $(v, \pi') \in \mathcal{T}\ [\![\tau'']\!]$. Sea una función $\phi' = \phi\left[\overline{\alpha_i \mapsto \texttt{any()}}\right]$ y la variable de tipo α:

  - Si $\alpha \notin \{\overline{\alpha_i}\}$, entonces se cumple que $\pi'(\alpha) = \pi(\alpha) \subseteq \mathcal{T}\ [\![\phi(\alpha)]\!]\big|_1 = \mathcal{T}\ [\![\phi'(\alpha)]\!]\big|_1$.
  - Si $\alpha \in \{\overline{\alpha_i}\}$, entonces se cumple que $\pi'(\alpha) \subseteq \mathcal{T}\ [\![\texttt{any()}]\!]|_1 = \mathcal{T}\ [\![\phi'(\alpha)]\!]\big|_1$.

Por lo tanto, se cumple que $\pi' \subseteq [\![\phi']\!]$ y podemos aplicar la hipótesis de inducción sobre $(v,\pi') \in \mathcal{T}\,[\![\tau'']\!]$, obteniendo que $(v,\pi') \in \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi']\!]$. Al ser $lug\ [\![\tau'']\!]\ \phi'$ un tipo *ground*, tenemos que $(v,\pi) \in \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi']\!] = \mathcal{T}\,[\![\tau']\!]$, demostrando así la proposición.

- **Caso** $\tau = \bigsqcup_{i=1}^{n} \sigma_i$

  Sea un par $(v,\pi) \in \mathcal{T}\,[\![\bigsqcup_{i=1}^{n} \sigma_i]\!]$, tal que $\pi \subseteq [\![\phi]\!]$. Esto implica que $(v,\pi) \in \mathcal{T}\,[\![\sigma_i]\!]$ para algún $i \in \{1..n\}$. Por hipótesis por inducción, para cada $i \in \{1..n\}$, con $(v,\pi) \in \mathcal{T}\,[\![\sigma_i]\!]$ obtendremos $(v,\pi) \in \mathcal{T}\,[\![lug\ [\![\sigma_i]\!]\ \phi]\!]$. Por lo tanto, si volvemos a unir los tipos que componían la unión, tendremos $(v,\pi) \in \mathcal{T}\,[\![\bigsqcup_{i=1}^{n} lug\ [\![\sigma_i]\!]\ \phi]\!] = \mathcal{T}\,[\![\tau']\!]$, demostrando así la proposición.

- **Caso** $\varphi = \alpha \subseteq \tau''$

  Suponemos que $\pi \vDash \alpha \subseteq \tau''$ y $\pi \subseteq [\![\phi]\!]$. Por la semántica de la restricción tenemos que:

  $$\pi(\alpha) \subseteq \{v \mid (v,\pi') \in \mathcal{T}\,[\![\tau'']\!], \pi' \subseteq \pi\}$$

  Sea un valor $v \in \pi(\alpha)$, entonces $\exists \pi' \subseteq \pi$ tal que $(v,\pi') \in \mathcal{T}\,[\![\tau'']\!]$. Además se cumple que $\pi' \subseteq \pi \subseteq [\![\phi]\!]$. Por lo tanto, aplicamos la hipótesis de inducción sobre el tipo τ'' y obtenemos $(v,\pi') \in \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]$. Tenemos entonces que $\pi(\alpha) \subseteq \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]\big|_1$. Por lo tanto, sea una variable de tipo β:

  - Si $\beta \neq \alpha$, entonces tenemos que:

    $$\pi(\beta) \subseteq \mathcal{T}\,[\![\phi(\beta)]\!]\big|_1 \quad \wedge \quad \pi(\beta) \subseteq \mathcal{T}\,[\![\mathsf{any}()]\!]|_1 = \mathcal{T}\,[\![[\alpha \mapsto lug\ [\![\tau'']\!]\ \phi]\,(\beta)]\!]\big|_1$$
    $$\Downarrow$$
    $$\pi(\beta) \subseteq \mathcal{T}\,[\![(\phi \sqcap [\alpha \mapsto lug\ [\![\tau'']\!]\ \phi])\,(\beta)]\!]\big|_1$$

  - Si $\beta = \alpha$, entonces tenemos que:

    $$\pi(\alpha) \subseteq \mathcal{T}\,[\![\phi(\alpha)]\!]\big|_1 \quad \wedge \quad \pi(\alpha) \subseteq \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]\big|_1$$
    $$\Downarrow$$
    $$\pi(\alpha) \subseteq \mathcal{T}\,[\![(\phi \sqcap [\alpha \mapsto lug\ [\![\tau'']\!]\ \phi])\,(\alpha)]\!]\big|_1$$

  Por consiguiente se cumple que $\pi \subseteq [\![\phi \sqcap [\alpha \mapsto lug\ [\![\tau'']\!]\ \phi]]\!]$, demostrando así la proposición.

- **Caso** $\varphi = c \subseteq \tau''$

  Suponemos que $\pi \vDash c \subseteq \tau''$ y $\pi \subseteq [\![\phi]\!]$. Por la semántica de la restricción tenemos que:

  $$c \in \{v \mid (v,\pi') \in \mathcal{T}\,[\![\tau'']\!], \pi' \subseteq \pi\}$$

  Sea un valor $v \in \mathbf{DVal}$, entonces $\exists \pi' \subseteq \pi$ tal que $(v,\pi') \in \mathcal{T}\,[\![\tau'']\!]$. Además se cumple que $\pi' \subseteq \pi \subseteq [\![\phi]\!]$. Por lo tanto, aplicamos la hipótesis de inducción sobre el tipo τ'' y obtenemos $(v,\pi') \in \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]$. Tenemos entonces dos posibles situaciones:

- Si $c \notin \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]\big|_1$, entonces el resultado será $\bot$, ya que no existe ninguna π que pueda satisfacer la restricción para c.
- Si $c \in \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]\big|_1$, entonces el resultado será que $\pi \subseteq [\![\phi]\!]$.

De este modo se demuestra la proposición.

- **Caso** $\varphi = \tau'' \Leftarrow \alpha$

Suponemos que $\pi \vDash \tau'' \Leftarrow \alpha$ y $\pi \subseteq [\![\phi]\!]$. Por la semántica de la restricción tenemos que:

$$\exists (\pi_v)_{v \in \pi(\alpha)} : \pi = \bigcup_{v \in \pi(\alpha)} \pi_v \wedge \left(\forall v \in \pi(\alpha) : (v, \pi_v) \in \mathcal{T}\,[\![\tau'']\!]\right)$$

Sea un valor $v \in \pi(\alpha)$, como $\pi_v \subseteq \pi$, se tiene que $\pi_v \subseteq [\![\phi]\!]$. Por lo tanto, aplicamos la hipótesis de inducción sobre el tipo τ'' y obtenemos $(v, \pi') \in \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]$. Tenemos entonces que $\pi(\alpha) \subseteq \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]\big|_1$. Por lo tanto, sea una variable de tipo β:

- Si $\beta \neq \alpha$, entonces tenemos que:

$$\begin{array}{c} \pi(\beta) \subseteq \mathcal{T}\,[\![\phi(\beta)]\!]\big|_1 \quad \wedge \quad \pi(\beta) \subseteq \mathcal{T}\,[\![\mathsf{any}\,()]\!]|_1 = \mathcal{T}\,[\![[\alpha \mapsto lug\ [\![\tau'']\!]\ \phi]\,(\beta)]\!]\big|_1 \\ \Downarrow \\ \pi(\beta) \subseteq \mathcal{T}\,[\![(\phi \sqcap [\alpha \mapsto lug\ [\![\tau'']\!]\ \phi])\,(\beta)]\!]\big|_1 \end{array}$$

- Si $\beta = \alpha$, entonces tenemos que:

$$\begin{array}{c} \pi(\alpha) \subseteq \mathcal{T}\,[\![\phi(\alpha)]\!]\big|_1 \quad \wedge \quad \pi(\alpha) \subseteq \mathcal{T}\,[\![lug\ [\![\tau'']\!]\ \phi]\!]\big|_1 \\ \Downarrow \\ \pi(\alpha) \subseteq \mathcal{T}\,[\![(\phi \sqcap [\alpha \mapsto lug\ [\![\tau'']\!]\ \phi])\,(\alpha)]\!]\big|_1 \end{array}$$

Por consiguiente se cumple que $\pi \subseteq [\![\phi \sqcap [\alpha \mapsto lug\ [\![\tau'']\!]\ \phi]]\!]$, demostrando así la proposición.

- **Caso** $C = \{\varphi_1, \ldots, \varphi_n\}$

Sea F una función definida del siguiente modo:

$$F(\phi) = (gsol\ [\![\varphi_n]\!]\ \circ \ldots \circ gsol\ [\![\varphi_1]\!]\ )\,(\phi)$$

Para todo ϕ', si $\pi \subseteq [\![\phi']\!]$, entonces tendremos:

$$\begin{array}{lcl} \pi \vDash \varphi_1 & \overset{\text{h.i.}}{\Longrightarrow} & \pi \subseteq [\![gsol\ [\![\varphi_1]\!]\ \phi']\!] \\ \pi \vDash \varphi_2 & \overset{\text{h.i.}}{\Longrightarrow} & \pi \subseteq [\![gsol\ [\![\varphi_2]\!]\ (gsol\ [\![\varphi_1]\!]\ \phi')]\!] \\ & \vdots & \\ \pi \vDash \varphi_n & \overset{\text{h.i.}}{\Longrightarrow} & \pi \subseteq [\![(gsol\ [\![\varphi_n]\!]\ \circ \ldots \circ gsol\ [\![\varphi_1]\!]\ )\,\phi']\!] = [\![F(\phi')]\!] \end{array}$$

Por lo tanto, para toda ϕ', si $\pi \subseteq [\![\phi']\!]$, entonces se cumple que $\pi \subseteq [\![F(\phi')]\!]$. Por hipótesis de inducción, sabemos que tenemos $\pi \subseteq [\![\phi]\!]$ y por inducción sobre n podemos demostrar que:

$$\pi \subseteq [\![F^n(\phi)]\!] = [\![F(F^{n-1}(\phi))]\!] = [\![F(F(\ldots(F(\phi))\ldots))]\!]$$

Por lo tanto, $\pi \subseteq \left[\!\left[fix_\phi F\right]\!\right]$, demostrando así la proposición.

□

5.3.2. Sustitución de variables de tipo

Las reglas de simplificación de tipos anotados (que se describirán en la sección 5.5) disponen de una regla de transformación que nos permite, para un tipo τ, tomar una restricción de la forma $\tau' \Leftarrow \alpha$ y reemplazar apariciones de la variable α en τ con el tipo τ'. Supongamos que la notación $\tau[\alpha/\tau']$ denota el operador de sustitución, de modo que *algunas* apariciones de α en τ son reemplazadas por τ'. Nótese que $\tau[\alpha/\tau']$ no es una sustitución en el sentido habitual. Por ello decimos que *algunas* apariciones son reemplazadas y no necesariamente todas. En la figura 5.6 presentamos una definición apropiada para $\tau[\alpha/\tau']$, de modo que se cumpla el siguiente lema:

Lema 11 (Lema de sustitución)**.** *Si tenemos un par* $(v,\pi) \in \mathcal{T}[\![\tau]\!]$ *y una instanciación* $\pi \vDash \tau' \Leftarrow \alpha$*, entonces se cumple que* $(v,\pi) \in \mathcal{T}[\![\tau[\alpha/\tau']]\!]$.

El operador de sustitución puede generar variables frescas por el camino. La siguiente generalización del lema de sustitución tiene en cuenta estas nuevas variables:

Lema 12 (Lema de sustitución con variables frescas)**.** *Si tenemos un par* $(v,\pi) \in \mathcal{T}[\![\tau]\!]$*, una instanciación* $\pi \vDash \tau' \Leftarrow \alpha$ *y un conjunto* X *de variables frescas generadas por la sustitución* $\tau[\alpha/\tau']$*, entonces existe algún* π' *tal que* $(v,\pi') \in \mathcal{T}[\![\tau[\alpha/\tau']]\!]$ *y* $\pi' \equiv \pi$ *(módulo* $\setminus X$*).*

Usaremos como abuso de notación $\pi' \equiv \pi$ (módulo $\setminus X$), empleada en el lema anterior, para denotar la notación $\pi' \equiv \pi$ (módulo **TypeVar**$\setminus X$), que ya vimos definida en la sección 4.2. Debido a la complejidad que tiene demostrar el último lema, puede resultar más fácil pensar en la sustitución como una secuencia de operaciones más simples. Esas operaciones son:

1. Hundir la restricción de encaje $\tau' \Leftarrow \alpha$ en el interior del árbol sintáctico abstracto del tipo τ. Por ejemplo, el tipo $\{\tau_1, \tau_2\}$ **when** $\tau' \Leftarrow \alpha$ se convertiría en $\{\tau_1$ **when** $\tau' \Leftarrow \alpha, \tau_2$ **when** $\tau' \Leftarrow \alpha\}$.

2. Cuando una restricción alcanza la variable de tipo que quiere sustituir, entonces llevaremos a cabo la sustitución. Es decir, α **when** $\tau' \Leftarrow \alpha$ se convierte en τ'.

Este último paso queda justificado con la siguiente proposición:

$$
\begin{aligned}
\tau\left[\alpha/\tau'\right] &= \tau \\
&\quad \text{si } \tau \text{ es igual a } \texttt{any()}, \texttt{none()}, c \text{ o } B \\
\{\tau_1,\ldots,\tau_n\}\left[\alpha/\tau'\right] &= \{\tau_1\left[\alpha/\tau'\right],\ldots,\tau_n\left[\alpha/\tau'\right]\} \\
\texttt{nelist}(\tau_1,\tau_2)\left[\alpha/\tau'\right] &= \texttt{nelist}(\tau_1\mu\left[\alpha/\tau'\right],\tau_2\left[\alpha/\tau'\right]) \textbf{ when } C \\
&\quad \text{donde } \{\overline{\alpha_i}\} = ftv(\tau_1) \cap ftv(\tau') \\
&\quad\quad \mu = \left[\overline{\alpha_i/\delta_i}\right] \text{ siendo } \overline{\delta_i} \text{ frescas} \\
&\quad\quad C = \{\beta \Leftarrow \mu(\beta) \mid \beta \in dom\,\mu, \beta \in itv(\tau_1)\} \\
&\quad\quad\quad \cup \{\beta = \mu(\beta) \mid \beta \in dom\,\mu, \beta \notin itv(\tau_1)\} \\
((\tau_1,\ldots,\tau_n) \to \tau)\left[\alpha/\tau'\right] &= \left((\tau_1\left[\alpha/\tau'\mu\right],\ldots,\tau_n\left[\alpha/\tau'\mu\right]) \to \tau\left[\alpha/\tau'\mu\right]\right) \textbf{ when } \overline{\delta_i \subseteq \alpha_i} \\
&\quad \text{donde } \{\overline{\alpha_i}\} = ftv(\tau') \\
&\quad\quad \mu = \left[\overline{\alpha_i/\delta_i}\right] \text{ siendo } \overline{\delta_i} \text{ frescas} \\
(\tau_1 \cup \cdots \cup \tau_n)\left[\alpha/\tau'\right] &= \left(\bigcup_{i=1}^{n} \tau_i\left[\alpha/\tau'\mu_i\right] \textbf{ when } \left\{\texttt{none()} \Leftarrow \beta \mid \beta \in \bigcup\left\{\overline{\delta_{k,j}} \mid k \in \{1..n\}, k \neq i\right\}\right\}\right) \\
&\quad\quad \textbf{when } \left\{\overline{\delta_{1,j} \subseteq \alpha_{1,j}},\ldots,\overline{\delta_{1,j} \subseteq \alpha_{1,j}}\right\} \\
&\quad \text{donde } \forall i \in \{1..n\}.\left\{\overline{\alpha_{i,j}}\right\} = ftv(\tau') \setminus \bigcup\{itv(\tau_k) \mid k \in \{1..n\}, k \neq i\} \\
&\quad\quad \forall i \in \{1..n\}.\mu_i = \left[\overline{\alpha_{i,j}/\delta_{i,j}}\right] \text{ siendo } \overline{\delta_{i,j}} \text{ frescas} \\
\left(\forall\overline{\alpha_i}.\tau\right)\left[\alpha/\tau'\right] &= \forall\overline{\alpha_i}.\tau\left[\alpha/\tau'\right] \\
&\quad \text{si } \left(ftv(\tau') \cup \{\alpha\}\right) \cap \{\overline{\alpha_i}\} = \emptyset \\
\left(\bigsqcup_{i=1}^{n} \sigma_i\right)\left[\alpha/\tau'\right] &= \bigsqcup_{i=1}^{n} \sigma_i\left[\alpha/\tau'\right]
\end{aligned}
$$

Figura 5.6: Definición de la sustitución de variables de tipo

Proposición 20. *Si tenemos* $(v,\pi) \in \mathcal{T}\left[\!\left[\alpha \textbf{ when } \tau' \Leftarrow \alpha\right]\!\right]$, *entonces* $(v,\pi) \in \mathcal{T}\left[\!\left[\tau'\right]\!\right]$.

Demostración. Supongamos un par $(v,\pi) \in \mathcal{T}[\![\alpha]\!]$ y una instanciación $\pi \vDash \tau' \Leftarrow \alpha$. De lo primero se deduce que $\pi(\alpha) = \{v\}$. De lo segundo se deduce la existencia de un π_v tal que $(v,\pi_v) \in \mathcal{T}\left[\!\left[\tau'\right]\!\right]$. Sin embargo, como $\pi \in Dcp\big(\{\pi_v\},\tau'\big)$, se cumple que $\pi = \pi_v$, y por lo tanto $(v,\pi) \in \mathcal{T}\left[\!\left[\tau'\right]\!\right]$. □

Lo opuesto no se cumple, por norma general, porque la variable α no está restringida en τ', pero aún con ello podemos dar el siguiente resultado:

Proposición 21. *Si tenemos* $(v,\pi) \in \mathcal{T}\left[\!\left[\tau'\right]\!\right]$, *entonces* $(v,\pi[\alpha \mapsto \{v\}]) \in \mathcal{T}\left[\!\left[\alpha \textbf{ when } \tau' \Leftarrow \alpha\right]\!\right]$, *donde* α *es una variable de tipo fresca.*

Demostración. Supongamos un par $(v,\pi) \in \mathcal{T}\left[\!\left[\tau'\right]\!\right]$ y una variable de tipo fresca α. Como $\alpha \notin ftv\big(\tau'\big)$, porque es fresca, sabemos que existe una instanciación $\pi' = \pi[\alpha \mapsto \{v\}]$ tal que $(v,\pi') \in \mathcal{T}\left[\!\left[\tau'\right]\!\right]$ y por lo tanto tendremos también que $(v,\pi') \in \mathcal{T}[\![\alpha]\!]$. Por la semántica de las restricciones, al tener que $\pi'(\alpha) = \{v\}$ y $(v,\pi') \in \mathcal{T}\left[\!\left[\tau'\right]\!\right]$, obtendremos que se cumple $\pi' \vDash \tau' \Leftarrow \alpha$. Por lo tanto, con $(v,\pi') \in \mathcal{T}[\![\alpha]\!]$ y $\pi' \vDash \tau' \Leftarrow \alpha$, tendremos que $(v,[\alpha \mapsto \{v\}]) \in \mathcal{T}\left[\!\left[\alpha \textbf{ when } \tau' \Leftarrow \alpha\right]\!\right]$. □

Ahora vamos a ver bajo qué condiciones podemos hundir las restricciones de encaje en el interior de un árbol sintáctico abstracto. El caso más sencillo es el de las tuplas:

Proposición 22. *Para todo* $i \in \{1..n\}$*:*

$$\mathcal{T}\left[\!\left[\{\tau_1,\ldots,\tau_n\} \textbf{ when } \tau' \Leftarrow \alpha\right]\!\right] = \mathcal{T}\left[\!\left[\{\tau_1,\ldots,\big(\tau_i \textbf{ when } \tau' \Leftarrow \alpha\big),\ldots,\tau_n\}\right]\!\right]$$

Demostración.

$$\begin{aligned}
& (v,\pi) \in \mathcal{T}\left[\!\left[\{\tau_1,\ldots,\tau_n\} \textbf{ when } \tau' \Leftarrow \alpha\right]\!\right] \\
\Leftrightarrow\ & \exists v_1,\ldots,v_n : v = \{v_1,\ldots,v_n\} \wedge (v_1,\pi) \in \mathcal{T}[\![\tau_1]\!] \wedge \cdots \wedge (v_n,\pi) \in \mathcal{T}[\![\tau_n]\!] \wedge \pi \vDash \tau' \Leftarrow \alpha \\
\Leftrightarrow\ & \exists v_1,\ldots,v_n : v = \{v_1,\ldots,v_n\} \wedge (v_1,\pi) \in \mathcal{T}[\![\tau_1]\!] \wedge \cdots \wedge (v_i,\pi) \in \mathcal{T}\left[\!\left[\tau_i \textbf{ when } \tau' \Leftarrow \alpha\right]\!\right] \\
& \qquad\qquad \wedge \cdots \wedge (v_n,\pi) \in \mathcal{T}[\![\tau_n]\!] \\
\Leftrightarrow\ & (v,\pi) \in \mathcal{T}\left[\!\left[\{\tau_1,\ldots,\tau_i \textbf{ when } \tau' \Leftarrow \alpha,\ldots,\tau_n\}\right]\!\right]
\end{aligned}$$

□

En el lema anterior hemos movido la restricción a uno de los componentes del tipo tupla, pero podríamos haber movido dicha restricción dentro de varios componentes. Por ejemplo, podemos demostrar

que se cumple lo siguiente:

$$\begin{aligned}
& \mathcal{T}\,[\![\{\tau_1,\tau_2\}\ \textbf{when}\ \tau' \Leftarrow \alpha]\!] \\
= \;& \mathcal{T}\,[\![(\{\tau_1,\tau_2\}\ \textbf{when}\ \tau' \Leftarrow \alpha)\ \textbf{when}\ \tau' \Leftarrow \alpha]\!] \\
= \;& \mathcal{T}\,[\![\{\tau_1\ \textbf{when}\ \tau' \Leftarrow \alpha,\tau_2\}\ \textbf{when}\ \tau' \Leftarrow \alpha]\!] \\
= \;& \mathcal{T}\,[\![\{\tau_1\ \textbf{when}\ \tau' \Leftarrow \alpha,\tau_2\ \textbf{when}\ \tau' \Leftarrow \alpha\}]\!]
\end{aligned}$$

Ahora que hemos terminado con las tuplas, vamos a demostrar qué casos existen para las listas no vacías. Primero, podemos hundir una restricción dentro del tipo que hay en la continuación de la lista:

Proposición 23.

$$\mathcal{T}\,[\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}\ \textbf{when}\ \tau' \Leftarrow \alpha]\!] = \mathcal{T}\,[\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\ \textbf{when}\ \tau' \Leftarrow \alpha\texttt{)}]\!]$$

Demostración.

$$\begin{aligned}
& (v,\pi) \in \mathcal{T}\,[\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}\ \textbf{when}\ \tau' \Leftarrow \alpha]\!] \\
\Leftrightarrow\ & \exists v_1,\ldots,v_n,v',\pi_1,\ldots,\pi_n : v = [v_1,\ldots,v_n|v'] \wedge (v_1,\pi_1) \in \mathcal{T}\,[\![\tau_1]\!] \wedge \cdots \wedge \\
& \quad (v_n,\pi_n) \in \mathcal{T}\,[\![\tau_1]\!] \wedge (v',\pi) \in \mathcal{T}\,[\![\tau_2]\!] \wedge \pi \in Dcp\big(\{\overline{\pi_i}^n\},\tau_1\big) \wedge \pi \vDash \tau' \Leftarrow \alpha \\
\Leftrightarrow\ & \exists v_1,\ldots,v_n,v',\pi_1,\ldots,\pi_n : v = [v_1,\ldots,v_n|v'] \wedge (v_1,\pi_1) \in \mathcal{T}\,[\![\tau_1]\!] \wedge \cdots \wedge \\
& \quad (v_n,\pi_n) \in \mathcal{T}\,[\![\tau_1]\!] \wedge (v',\pi) \in \mathcal{T}\,[\![\tau_2\ \textbf{when}\ \tau' \Leftarrow \alpha]\!] \wedge \pi \in Dcp\big(\{\overline{\pi_i}^n\},\tau_1\big) \\
\Leftrightarrow\ & (v,\pi) \in \mathcal{T}\,[\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\ \textbf{when}\ \tau' \Leftarrow \alpha\texttt{)}]\!]
\end{aligned}$$

□

Sin embargo, si queremos hundir la restricción de encaje en el tipo τ_1 que representa los elementos del cuerpo de la lista no vacía, tenemos que asegurarnos de que no hay colisión alguna entre las variables de tipo de τ' y aquellas que aparecen en τ_1.

Proposición 24. *Si tenemos* $(v,\pi) \in \mathcal{T}\,[\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}\ \textbf{when}\ \tau' \Leftarrow \alpha]\!]$, $ftv(\tau_1) \cap ftv\big(\tau'\big) = \emptyset$, *y* α *no aparece libre en* τ', *entonces* $(v,\pi) \in \mathcal{T}\,[\![\texttt{nelist(}\tau_1\ \textbf{when}\ \tau' \Leftarrow \alpha\texttt{,}\tau_2\texttt{)}]\!]$.

Demostración. Supongamos un par $(v,\pi) \in \mathcal{T}\,[\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}]\!]$ tal que $\pi \vDash \tau' \Leftarrow \alpha$. Tendremos entonces un valor $v = [v_1,\ldots,v_n|v']$ para algunos valores $v_1,\ldots,v_n,v'$, y una instanciación $\pi \in Dcp\big(\{\overline{\pi_i}^n\},\tau_1\big)$ para algunas $\pi_1,\ldots,\pi_n$, tales que $(v_i,\pi_i) \in \mathcal{T}\,[\![\tau_1]\!]$ y $(v',\pi) \in \mathcal{T}\,[\![\tau_2]\!]$.

Siguiendo la semántica de la restricción $\tau' \Leftarrow \alpha$ podremos encontrar otra descomposición de π en varias π_v, una por cada valor $v \in \pi(\alpha)$. Es decir, $\pi \in Dcp\big(\{\pi_v\},\tau'\big)$. Esto implica que $\pi = \bigcup_{v\in\pi(\alpha)} \pi_v$. Como α no aparece libre en τ', podemos suponer sin perder precisión que $\pi_v(\alpha) = \{v\}$.

Como sabemos que $\pi \in Dcp(\{\overline{\pi_i}^n\}, \tau_1)$ implica que $\pi(\alpha) = \pi_1(\alpha) \cup \cdots \cup \pi_n(\alpha)$, esto nos permitirá reescribir la descomposición—de la instanciación π anterior—de la siguiente manera:

$$\pi = \bigcup_{v \in \pi(\alpha)} \pi_v = \bigcup_{i=1}^{n} \bigcup_{v \in \pi_i(\alpha)} \pi_v$$

Vamos a denotar con π'_i la instanciación $\bigcup_{v \in \pi_i(\alpha)} \pi_v$, para que podamos decir que $\pi = \bigcup_{i=1}^{n} \pi'_i$. De ello también se deduce que $\pi'_i \in Dcp(\{\pi_v \mid v \in \pi_i(\alpha)\}, \tau_1)$ y por lo tanto $\pi'_i \vDash \tau' \Leftarrow \alpha$.

Ahora, por cada $i \in \{1..n\}$, vamos a definir una instanciación π''_i de la siguiente manera:

$$\text{Para cada } \beta \in \textbf{TypeVar} : \pi''_i(\beta) = \begin{cases} \pi'_i(\beta) & \text{si } \beta \in ftv(\tau') \\ \pi_i(\beta) & \text{e.o.c.} \end{cases}$$

Nótese que $\pi'_i(\alpha) = \bigcup_{v \in \pi(\alpha)} \pi_v(\alpha) = \bigcup_{v \in \pi_i(\alpha)} \{v\} = \pi_i(\alpha)$, por lo tanto podemos asegurar que $\pi''_i \equiv \pi'_i$ (módulo $ftv(\tau') \cup \{\alpha\}$) para cada $i \in \{1..n\}$. Por consiguiente, tomando $\pi'_i \vDash \tau' \Leftarrow \alpha$ obtenemos que $\pi''_i \vDash \tau' \Leftarrow \alpha$. Además de esto, como $\pi''_i \equiv \pi_i$ (módulo $\setminus ftv(\tau')$) y $ftv(\tau')$ es disjunto de $ftv(\tau_1)$, se cumple que $\pi''_i \equiv \pi_i$ (módulo $ftv(\tau_1)$) y por ello $(v_i, \pi''_i) \in \mathscr{T}[\![\tau_1]\!]$ para cada $i \in \{1..n\}$. Como consecuencia de esto, $(v_i, \pi''_i) \in \mathscr{T}[\![\tau_1 \textbf{ when } \tau' \Leftarrow \alpha]\!]$ para todo $i \in \{1..n\}$.

Ahora vamos a demostrar que $\pi = \bigcup_{i=1}^{n} \pi''_i$. Supongamos la existencia de una variable de tipo β. Si $\beta \in ftv(\tau')$ obtenemos que:

$$\bigcup_{i=1}^{n} \pi''_i(\beta) = \bigcup_{i=1}^{n} \pi'_i(\beta) = \pi(\beta)$$

Por el contrario, si $\beta \notin ftv(\tau')$ obtenemos que:

$$\bigcup_{i=1}^{n} \pi''_i(\beta) = \bigcup_{i=1}^{n} \pi_i(\beta) = \pi(\beta)$$

Lo siguiente que vamos a demostrar es que $\pi \equiv \pi''_i$ (módulo $\setminus itv(\tau_1 \textbf{ when } \tau' \Leftarrow \alpha)$) para cada $i \in \{1..n\}$. Suponiendo algunas variables de tipo $\beta \notin itv(\tau_1 \textbf{ when } \tau' \Leftarrow \alpha)$:

- Si $\beta \in ftv(\tau')$, entonces $\pi''_i(\beta) = \pi'_i(\beta) = \bigcup_{v \in \pi_i(\alpha)} \pi_v(\beta)$.

  Sabemos que $\pi \in Dcp(\{\pi_v \mid v \in \pi(\alpha)\}, \tau')$. Como $\beta \notin itv(\tau')$, deducimos que $\pi_v(\beta) = \pi(\beta)$, así que $\pi''_i(\beta) = \bigcup_{v \in \pi_i(\alpha)} \pi(\beta) = \pi(\beta)$.

- Si $\beta \notin ftv(\tau)$, entonces $\pi''_i(\beta) = \pi_i(\beta)$.

  En este caso, como $\pi \in Dcp(\{\overline{\pi_i}^n\}, \tau_1)$ y $\beta \notin itv(\tau_1)$, tendremos que $\pi_i(\beta) = \pi(\beta)$, así que $\pi''_i(\beta) = \pi(\beta)$.

Por lo tanto, hemos demostrado que:

$$\pi \in Dcp\big(\{\pi''_i \mid i \in \{1..n\}\}, \tau_1\big)$$
$$\big(v_i, \pi''_i\big) \in \mathcal{T}\,[\![\tau_1 \textbf{ when } \tau' \Leftarrow \alpha]\!] \text{ para cada } i \in \{1..n\}$$
$$\big(v', \pi\big) \in \mathcal{T}\,[\![\tau_2]\!]$$

Por lo tanto, $([v_1, \ldots, v_n | v'], \pi) \in \mathcal{T}\,[\![\texttt{nelist(}\tau_1 \textbf{ when } \tau' \Leftarrow \alpha\texttt{,}\tau_2\texttt{)}]\!]$. □

En la proposición anterior, si la condición $ftv\big(\tau'\big) \cap ftv(\tau_1)$ no se cumpliera, hundir la restricción dentro del tipo τ_1 podría alterar la semántica del tipo. Por ejemplo, supongamos los siguientes tipos:

$$\tau_1 = \texttt{nelist(\{}\alpha,\beta\texttt{\},[])} \textbf{ when } \beta \Leftarrow \alpha \qquad\qquad \tau_2 = \texttt{nelist(\{}\alpha,\beta\texttt{\}} \textbf{ when } \beta \Leftarrow \alpha\texttt{,[])}$$

La lista `[{1,2},{2,1}]` pertenece a la semántica de τ_1, pero no a la de τ_2.

En el caso $\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}$ **when** $\tau' \Leftarrow \alpha$, si τ' y τ_1 tienen variables de tipo compartidas, podemos renombrar aquellas que hay en τ_1 para evitar las colisiones. Además, deberemos añadir las correspondientes restricciones de igualdad entre las variables frescas y las antiguas.

Proposición 25. *Supongamos un renombramiento μ y un tipo τ tal que $dom\,\mu \subseteq ftv(\tau)$ y $rng\,\mu \cap ftv(\tau) = \emptyset$. Vamos a denotar con C el siguiente conjunto de restricciones:*

$$C = \big\{\beta \Leftarrow \mu(\beta) \mid \beta \in dom\,\mu \wedge \beta \in itv(\tau)\big\} \cup \big\{\beta = \mu(\beta) \mid \beta \in dom\,\mu \wedge \beta \notin itv(\tau)\big\}$$

Entonces:

1. *Si $(v,\pi) \in \mathcal{T}\,[\![\tau]\!]$, entonces $(v,\pi\mu) \in \mathcal{T}\,[\![\tau\mu \textbf{ when } C]\!]$.*
2. *Si $(v,\pi) \in \mathcal{T}\,[\![\tau\mu \textbf{ when } C]\!]$, entonces $(v,\pi) \in \mathcal{T}\,[\![\tau]\!]$.*
3. *$ftv\big(\tau\mu \textbf{ when } C\big) = ftv(\tau) \cup rng\,\mu$.*
4. *$itv\big(\tau\mu \textbf{ when } C\big) = itv(\tau) \cup \big\{\mu(\beta) \mid \beta \in dom\,\mu \wedge \beta \in itv(\tau)\big\}$.*

Demostración. Vamos a empezar con la propiedad (1). Supongamos que tenemos $(v,\pi) \in \mathcal{T}\,[\![\tau]\!]$. Obtendríamos $(v,\pi\mu) \in \mathcal{T}\,[\![\tau\mu]\!]$. Ahora tendremos que demostrar que $\pi\mu \vDash C$. Para cualquier $\beta \in dom\,\mu$, sabemos que $\mu(\beta) \in rng\,\mu$ y, por la definición de $\pi\mu$, se cumple que $(\pi\mu)(\mu(\beta)) = \pi(\mu^{-1}(\mu(\beta))) = \pi(\beta) = (\pi\mu)(\beta)$, ya que $\beta \notin rng\,\mu$ como consecuencia de que $rng\,\mu$ y $dom\,\mu$ son disjuntos. Por lo tanto, $\pi\mu \vDash \beta \Leftarrow \mu(\beta)$ y $\pi\mu \vDash \beta = \mu(\beta)$. Dado que se cumple para cualquier $\beta \in dom\,\mu$, entonces obtenemos que $\pi\mu \vDash C$ y por ello $(v,\pi\mu) \in \mathcal{T}\,[\![\tau\mu \textbf{ when } C]\!]$.

Continuaremos con la propiedad (2). Supongamos un par (v,π) tal que $(v,\pi) \in \mathcal{T}\,[\![\tau\mu]\!]$ y $\pi \vDash C$. Deducimos de ello que $(v,\pi\mu^{-1}) \in \mathcal{T}\,[\![\tau\mu\mu^{-1}]\!]$. Como $dom\,\mu$ es disjunto de $rng\,\mu$, obtenemos que $\tau\mu\mu^{-1} = \tau$, así que $(v,\pi\mu^{-1}) \in \mathcal{T}\,[\![\tau]\!]$. Ahora vamos a demostrar que $\pi\mu^{-1} = \pi$. Por la definición de $\pi\mu^{-1}$, sabemos

que $\pi\mu^{-1} \equiv \pi$ (módulo $\setminus rng\ \mu^{-1}$), por lo que nos vamos a centrar en las variables que hay en $rng\ \mu^{-1}$. Si $\beta \in rng\ \mu^{-1}$, entonces $\beta \in dom\ \mu$. Como $\pi \vDash C$, esto significará que $\pi(\beta) = \pi(\mu(\beta))$. Luego, por la definición de $\pi\mu^{-1}$, obtendremos que $(\pi\mu^{-1})(\beta) = \pi((\mu^{-1})^{-1}(\beta)) = \pi(\mu(\beta)) = \pi(\beta)$, por lo que habremos demostrado que $\pi\mu^{-1}$ también coincide con π in $rng\ \mu^{-1}$. Por lo tanto, con $(\upsilon, \pi\mu^{-1}) \in \mathscr{T}\ [\![\tau]\!]$ y $\pi\mu^{-1} = \pi$ se deduce que $(\upsilon, \pi) \in \mathscr{T}\ [\![\tau]\!]$.

Vamos a demostrar ahora la propiedad (3):

- $ftv\big(\tau\mu\ \mathbf{when}\ C\big) \subseteq ftv(\tau) \cup rng\ \mu$:

  Supongamos una variable de tipo $\gamma \in ftv\big(\tau\mu\ \mathbf{when}\ C\big)$. Si $\gamma \in rng\ \mu$ habremos terminado. Si no, podemos distinguir entre los siguientes casos:

  - Si $\gamma \in ftv\big(\tau\mu\big)$, se cumple que $\gamma\mu^{-1} \in ftv(\tau)$. En este caso, como $\gamma \notin rng\ \mu$ se deduce que $\gamma \notin dom\ \mu^{-1}$, así que $\gamma\mu^{-1} = \gamma$ y por lo tanto $\gamma \in ftv(\tau)$.
  - Si $\gamma \in ftv(C)$, sabemos que $ftv(C) = dom\ \mu \cup rng\ \mu$. Como $\gamma \notin rng\ \mu$ tendremos que $\gamma \in dom\ \mu$. Es más, dado que $dom\ \mu \subseteq ftv(\tau)$ obtenemos que $\gamma \in ftv(\tau)$.

- $ftv\big(\tau\mu\ \mathbf{when}\ C\big) \supseteq ftv(\tau) \cup rng\ \mu$:

  - Supongamos $\gamma \in ftv(\tau)$, de lo que se deduce que $\gamma\mu \in ftv\big(\tau\mu\big)$. Si $\gamma \in dom\ \mu$ entonces $\gamma \in ftv(C)$. Si $\gamma \notin dom\ \mu$ entonces $\gamma\mu = \gamma$ y obtenemos que $\gamma \in ftv\big(\tau\mu\big)$.
  - Supongamos $\gamma \in rng\ \mu$. Tendremos que existe $\beta \in dom\ \mu$ tal que $\mu(\beta) = \gamma$, pero $\mu(\beta)$ aparece libre en C, por lo tanto también lo hace γ.

Finalmente, vamos a demostrar (4):

- $itv\big(\tau\mu\ \mathbf{when}\ C\big) \subseteq itv(\tau) \cup \big\{\mu(\beta) \mid \beta \in dom\ \mu \wedge \beta \in itv(\tau)\big\}$:

  - Supongamos $\gamma itv\big(\tau\mu\big)$. Por consiguiente $\gamma\mu^{-1} \in itv\big(\tau\mu\mu^{-1}\big) = itv(\tau)$. Si $\gamma \notin dom\ \mu^{-1}$ entonces $\gamma\mu^{-1} = \gamma$ y por ello $\gamma \in itv(\tau)$. Si $\gamma \in dom\ \mu^{-1}$, entonces $\gamma\mu^{-1} \in dom\ \mu$, así que $\mu(\gamma\mu^{-1})$ pertenece al conjunto $\big\{\mu(\beta) \mid \beta \in dom\ \mu \wedge \beta \in itv(\tau)\big\}$, y también lo hace γ ya que $\mu(\gamma\mu^{-1}) = \mu(\mu^{-1}(\gamma)) = \gamma$.
  - Supongamos $\gamma \in itv(C)$. Las únicas variables en $itv(C)$ son aquellas que aparecen en el lado izquierdo de las restricción de encaje. Por lo tanto $\gamma \in itv(\tau)$.

- $itv\big(\tau\mu\ \mathbf{when}\ C\big) \supseteq itv(\tau) \cup \big\{\mu(\beta) \mid \beta \in dom\ \mu \wedge \beta \in itv(\tau)\big\}$:

  - Si $\gamma \in itv(\tau)$ se cumple que $\gamma\mu \in itv\big(\tau\mu\big)$. Si $\gamma \notin dom\ \mu$ entonces $\gamma = \gamma\mu \in itv\big(\tau\mu\big)$. Sin embargo, si $\gamma \in dom\ \mu$ entonces la restricción $\gamma \Leftarrow \mu(\gamma)$ aparece en C, y por lo tanto $\gamma \in itv(C)$.
  - Si $\gamma \in \big\{\mu(\beta) \mid \beta \in itv(\tau) \wedge \beta \in dom\ \mu\big\}$, entonces existe β tal que $\gamma = \mu(\beta)$ y $\beta \in itv(\tau)$. De esto último se deduce que $\mu(\beta) \in itv\big(\tau\mu\big)$, y por ello $\gamma \in itv\big(\tau\mu\big)$.

□

Esta última proposición nos permite generalizar la proposición 24 para poder también encargarnos de las colisiones entre variables de tipo en τ' y τ_1:

Colorario 3. *Si* $(v,\pi) \in \mathcal{T}\,[\![\texttt{nelist}(\tau_1,\tau_2)\ \textbf{when}\ \tau' \Leftarrow \alpha]\!]$ *y* α *no aparece libre en* τ'*, entonces* $(v,\pi\mu) \in \mathcal{T}\,[\![\texttt{nelist}(\tau_1\mu\ \textbf{when}\ \tau' \Leftarrow \alpha, \tau_2\mu)\ \textbf{when}\ C]\!]$ *para todo renombramiento* μ *y conjunto de restricciones* C *tales que:*

1. $dom\,\mu = ftv(\tau_1) \cap ftv(\tau')$.
2. *rng* μ *son variables frescas que no aparecen ni en* τ_1, τ_2 *o* $\tau' \Leftarrow \alpha$.
3. $C = \{\beta \Leftarrow \mu(\beta) \mid \beta \in dom\,\mu \wedge \beta \in itv(\tau_1)\} \cup \{\beta = \mu(\beta) \mid \beta \in dom\,\mu \wedge \beta \notin itv(\tau_1)\}$

Demostración. Dado un par $(v,\pi) \in \mathcal{T}\,[\![\texttt{nelist}(\tau_1,\tau_2)\ \textbf{when}\ \tau' \Leftarrow \alpha]\!]$ podemos aplicar la proposición 25 para poder obtener que $(v,\pi\mu) \in \mathcal{T}\,[\![\texttt{nelist}(\tau_1\mu,\tau_2\mu)\ \textbf{when}\ C \cup \{\tau' \Leftarrow \alpha\}]\!]$. Ahora las variables en $\tau_1\mu$ y τ' son disjuntas, así que podemos aplicar la proposición 24 para obtener que $(v,\pi\mu) \in \mathcal{T}\,[\![\texttt{nelist}(\tau_1\mu\ \textbf{when}\ \{\tau' \Leftarrow \alpha\}, \tau_2\mu)\ \textbf{when}\ C]\!]$. □

Tras estudiar cómo manejar las listas no vacías, pasamos a los tipos funcionales. De nuevo, vamos a suponer que la restricción de encaje $\tau \Leftarrow \alpha$ que queremos hundir no comparte variables de tipo con el tipo funcional en cuestión. Sin embargo, incluso si exigimos que las variables sean disjuntas, tenemos que ser cuidadosos con las variables instanciables de τ en el caso particular en el que obtenemos el grafo vacío de la semántica del tipo funcional. Por ejemplo, vamos a suponer el tipo $((\texttt{none}()) \to \texttt{none}())\ \textbf{when}\ \beta \Leftarrow \alpha$. Suponiendo la instanciación $\pi = [\beta \mapsto \{1\}, \alpha \mapsto \{1\}]$, el par $(\emptyset,\pi)$ pertenece a la semántica de este tipo restringido. Desafortunadamente, el par no pertenece a la semántica del tipo $(\texttt{none}()) \to (\texttt{none}()\ \textbf{when}\ \beta \Leftarrow \alpha)$, ya que cualquier instanciación π tal que $(\emptyset,\pi)$ pertenezca a $\mathcal{F}\,[\![(\texttt{none}()) \to (\texttt{none}()\ \textbf{when}\ \beta \Leftarrow \alpha)]\!]$ exige que $\pi(\beta)$ sea vacía, porque β se encuentra en una posición instanciable.

Sin embargo, podemos transformar de forma segura el tipo $((\texttt{none}()) \to \texttt{none}())\ \textbf{when}\ \beta \Leftarrow \alpha$ en $((\texttt{none}()) \to \texttt{none}()\ \textbf{when}\ \beta' \Leftarrow \alpha)\ \textbf{when}\ \beta' \subseteq \beta$, pero corriendo el riesgo de perder precisión con la transformación.

La siguiente proposición generaliza el hecho que acabamos de describir:

Proposición 26. *Supongamos un grafo de función* f *y una instanciación* π *tales que* $\pi \vDash \tau \Leftarrow \alpha$ *y que* $(f,\pi) \in \mathcal{F}\,[\![(\overline{\tau_i}^n) \to \tau']\!]$*. Si* α *no aparece libre en* τ *y los tipos* τ *y* $(\overline{\tau_i}^n) \to \tau'$ *no tienen variables libres en común, entonces existe un* π' *tal que:*

$$(f,\pi') \in \mathcal{T}\,[\![((\overline{\tau_i}^n) \to \tau'\ \textbf{when}\ \tau\left[\overline{\beta_i/\beta'_i}\right] \Leftarrow \alpha)\ \textbf{when}\ \overline{\beta'_i \subseteq \beta_i}]\!]$$

donde $\{\overline{\beta_i}\} = itv(\tau)$, $\overline{\beta'_i}$ *son variables de tipo frescas, y* $\pi' \equiv \pi$ *(módulo* $\setminus \{\overline{\beta'_i}\}$*).*

Demostración. Vamos a suponer un par $(f,\pi) \in \mathscr{F}[\![(\overline{\tau_i}^n) \to \tau']\!]$ y una instanciación $\pi \vDash \tau \Leftarrow \alpha$. En adelante, vamos a denotar con μ el renombramiento con variables frescas $\left[\overline{\beta_i / \beta'_i}\right]$.

Empezaremos con el caso en el que $f = \emptyset$. En este caso $\pi(\gamma) = \emptyset$ para cualquier variable de tipo γ en $itv\big((\overline{\tau_i}^n) \to \tau'\big)$. Vamos a definir la instanciación π' de la siguiente manera:

$$\text{Para todo } \gamma \in \mathbf{TypeVar} : \pi'(\gamma) = \begin{cases} \emptyset & \text{si } \gamma \in \left\{\overline{\beta'_i}\right\} \\ \pi(\gamma) & \text{e.o.c.} \end{cases}$$

Obviamente sabemos que $\pi' \equiv \pi$ (módulo $\setminus \left\{\overline{\beta'_i}\right\}$). Con lo que ahora vamos a demostrar que $\pi'(\gamma) = \emptyset$ para cualquier γ perteneciente a $itv\big((\overline{\tau_i}^n) \to \tau'$ **when** $\tau\mu \Leftarrow \alpha\big)$.

1. Si $\gamma \in itv\big((\overline{\tau_i}^n) \to \tau'\big)$, entonces $\pi'(\gamma) = \pi(\gamma) = \emptyset$.
2. Si $\gamma \in itv(\tau\mu)$, entonces γ deberá ser alguna de las β'_i, por lo que $\pi'(\gamma) = \emptyset$.

Como consecuencia de lo anterior, tenemos que $(\emptyset, \pi') \in \mathscr{F}[\![(\overline{\tau_i}^n) \to \tau'$ **when** $\tau\mu \Leftarrow \alpha]\!]$. Es más, sabemos que $\pi' \vDash \beta'_i \subseteq \beta_i$ para cualquier i, ya que $\pi'(\beta'_i) = \emptyset$. Por lo tanto, la proposición se cumple para el caso en el que $f = \emptyset$.

Ahora vamos a suponer que $f \neq \emptyset$. Sabemos que existe una familia de instanciaciones $\{\pi_w \mid w \in f\}$ tales que para cada $w \in f$, si $w = \big((\overline{v_i}^n), v'\big)$:

$$\begin{array}{l} (v_i, \pi_w) \in \mathscr{T}[\![\tau_i]\!] \text{ para cada } i \in \{1..n\} \\ \big(v', \pi_w\big) \in \mathscr{T}[\![\tau']\!] \\ \pi \in Dcp\big(\{\pi_w \mid w \in f\}, (\overline{\tau_i}^n) \to \tau'\big) \end{array}$$

Además, partiendo de que π cumple que $\pi \vDash \tau \Leftarrow \alpha$, se deduce la existencia de una familia de instanciaciones $\{\pi^\circ_v \mid v \in \pi(\alpha)\}$ tales que $\pi \in Dcp\big(\{\pi^\circ_v \mid v \in \pi(\alpha)\}, \tau\big)$ y $(v, \pi^\circ_v) \in \mathscr{T}[\![\tau]\!]$ para cada $v \in \pi(\alpha)$. Dado que α no pertenece a τ, suponemos sin perder precisión que $\pi_v(\alpha) = \{v\}$ para cualquier $v \in \pi(\alpha)$.

Para cualquier $w \in f$, vamos a definir $\pi^\bullet_w$ de la siguiente manera:

$$\pi^\bullet_w = \bigcup_{v \in \pi_w(\alpha)} \pi^\circ_v \tag{5.3}$$

Como $\pi(\alpha) = \bigcup_{w \in f} \pi_w(\alpha)$, podemos descomponer π del siguiente modo:

$$\pi = \bigcup_{v \in \pi(\alpha)} \pi^\circ_v = \bigcup_{w \in f} \bigcup_{v \in \pi_w(\alpha)} \pi^\circ_v = \bigcup_{w \in f} \pi^\bullet_w$$

Ahora, como sabemos que $\pi \equiv \pi^\circ_v$ (módulo $\setminus itv(\tau)$) para cualquier $v \in \pi_w(\alpha)$ por la definición de *Dcp*, se deduce que $\pi \equiv \bigcup_{v \in \pi_w(\alpha)} \pi^\circ_v$ (módulo $\setminus itv(\tau)$) y por ello $\pi \equiv \pi^\bullet_w$ (módulo $\setminus itv(\tau)$), que a su vez

implica que $\pi^{\circ}_{v} \equiv \pi^{\bullet}_{w}$ (módulo $\setminus itv(\tau)$). Por lo tanto obtenemos que:

$$\pi^{\bullet}_{w} \in Dcp\big(\{\pi^{\circ}_{v} \mid v \in \pi_{w}(\alpha)\}, \tau\big)$$

Es más, como hemos supuesto que $\pi_{v}(\alpha) = \{v\}$ para cualquier $v \in \pi(\alpha)$, se deduce que $\pi^{\bullet}_{w}(\alpha) = \bigcup_{v \in \pi_{w}(\alpha)} \pi^{\circ}_{v}(\alpha) = \bigcup_{v \in \pi_{w}(\alpha)} \{v\} = \pi_{w}(\alpha)$. Por consiguiente, podemos rescribir (5.3) y los previas hechos de la siguiente manera:

$$\pi^{\bullet}_{w} = \bigcup_{v \in \pi^{\bullet}_{w}(\alpha)} \pi^{\circ}_{v}$$

$$\pi^{\bullet}_{w} \in Dcp\big(\{\pi^{\circ}_{v} \mid v \in \pi^{\bullet}_{w}(\alpha)\}, \tau\big)$$

Por lo tanto, obtenemos que $\pi^{\bullet}_{w} \vDash \tau \Leftarrow \alpha$ para cualquier $w \in f$.

Vamos a definir, para cualquier $w \in f$ la instanciación π'_{w} del siguiente modo:

$$\text{Para cualquier } \gamma \in \mathbf{TypeVar} : \pi'_{w}(\gamma) = \begin{cases} \pi^{\bullet}_{w}(\gamma) & \text{si } \gamma \in ftv(\tau) \\ \pi_{w}(\gamma) & \text{e.o.c.} \end{cases}$$

Dado que $\pi'_{w} \equiv \pi^{\bullet}_{w}$ (módulo $ftv(\tau) \cup \{\alpha\}$), se deduce que $\pi'_{w} \vDash \tau \Leftarrow \alpha$, lo que implica que $\pi'_{w}\mu \vDash \tau\mu \Leftarrow \alpha$. Aparte de eso, obtenemos que $\pi'_{w} \equiv \pi_{w}$ (módulo $\setminus ftv(\tau)$), lo que implica que $\pi'_{w}\mu \equiv \pi_{w}$ (módulo $\setminus (ftv(\tau) \cup \{\overline{\beta'_{i}}\})$). Como las variables en τ son disjuntas de aquellas que hay en el tipo funcional, y también lo son las variables frescas $\overline{\beta'_{i}}$, tenemos que $\pi'_{w}\mu \equiv \pi_{w}$ (módulo $ftv\big((\overline{\tau_{i}}^{n}) \to \tau'\big)$). Por lo tanto, sabiendo que $(v_{i}, \pi_{w}) \in \mathscr{T}[\![\tau_{i}]\!]$ para cada $i \in \{1..n\}$ y que $(v', \pi_{w}) \in \mathscr{T}[\![\tau']\!]$, obtenemos que:

$$\begin{array}{l} (v_{i}, \pi'_{w}\mu) \in \mathscr{T}[\![\tau_{i}]\!] \text{ para cada } i \in \{1..n\} \\ (v', \pi'_{w}\mu) \in \mathscr{T}[\![\tau']\!] \end{array}$$

Si combinamos lo último con el hecho de que $\pi'_{w}\mu \vDash \tau\mu \Leftarrow \alpha$, tendremos que:

$$\big(v_{i}, \pi'_{w}\mu\big) \in \mathscr{T}[\![\tau_{i}]\!] \text{ para cada } i \in \{1..n\} \tag{5.4}$$

$$\big(v', \pi'_{w}\mu\big) \in \mathscr{T}[\![\tau' \textbf{ when } \tau\mu \Leftarrow \alpha]\!] \tag{5.5}$$

para toda tupla $w \in f$. Ahora vamos a demostrar que:

$$\pi\mu \in Dcp\big(\{\pi'_{w}\mu \mid w \in f\}, (\overline{\tau_{i}}^{n}) \to \tau' \textbf{ when } \tau\mu \Leftarrow \alpha\big) \tag{5.6}$$

De cara a demostrar esto último, necesitamos primero demostrar:

$$\pi\mu = \bigcup_{w \in f} \pi'_{w}\mu \tag{5.7}$$

$$\pi\mu \equiv \pi'_{w}\mu \ \big(\text{módulo } itv\big((\overline{\tau_{i}}^{n}) \to \tau' \textbf{ when } \tau\mu \Leftarrow \alpha\big)\big) \tag{5.8}$$

Vamos a empezar demostrando (5.7). Supongamos una $\gamma \in \mathbf{TypeVar}$.

1. Si $\gamma \in \left\{\overline{\beta'_i}\right\}$, recordemos que su contraparte $\beta_i = \mu^{-1}(\beta'_i)$ pertenece a $ftv(\tau)$, por lo que:

$$(\pi\mu)(\beta'_i) = \pi(\beta_i) = \bigcup_{w \in f} \pi^{\bullet}_w(\beta_i) = \bigcup_{w \in f} \pi'_w(\beta_i) = \bigcup_{w \in f} (\pi'_w\mu)(\beta'_i)$$

2. Si $\gamma \notin \left\{\overline{\beta'_i}\right\}$, vamos a distinguir entre casos:

   a) Si $\gamma \in ftv(\tau)$ tendremos que:

$$(\pi\mu)(\gamma) = \pi(\gamma) = \bigcup_{w \in f} \pi^{\bullet}_w(\gamma) = \bigcup_{w \in f} \pi'_w(\gamma) = \bigcup_{w \in f} (\pi'_w\mu)(\gamma)$$

   b) Si $\gamma \notin ftv(\tau)$ tendremos que:

$$(\pi\mu)(\gamma) = \pi(\gamma) = \bigcup_{w \in f} \pi_w(\gamma) = \bigcup_{w \in f} \pi'_w(\gamma) = \bigcup_{w \in f} (\pi'_w\mu)(\gamma)$$

Ahora vamos a demostrar (5.8). Supongamos una variable de tipo $\gamma \notin itv\left(\left(\overline{\tau_i}^n\right) \to \tau' \textbf{ when } \tau\mu \Leftarrow \alpha\right)$. Esto implica que γ no es ninguna de las variables $\left\{\overline{\beta'_i}\right\}$, porque estas últimas variables aparecen en posiciones instanciables de la restricción $\tau\mu \Leftarrow \alpha$.

1. Si $\gamma \in ftv(\tau)$ tendremos que:

$$(\pi'_w\mu)(\gamma) = \pi'_w(\gamma) = \pi^{\bullet}_w(\gamma) = \bigcup_{v \in \pi_w(\alpha)} \pi^{\circ}_v(\gamma)$$

   Pero, como $\pi \in Dcp\left(\left\{\pi^{\circ}_v \mid v \in \pi(\alpha)\right\}, \tau\right)$ y $\gamma \notin itv(\tau)$, obtenemos que:

$$\bigcup_{v \in \pi_w(\alpha)} \pi^{\circ}_v(\gamma) = \bigcup_{v \in \pi_w(\alpha)} \pi(\gamma) = \pi(\gamma) = (\pi\mu)(\gamma)$$

2. Si $\gamma \notin ftv(\tau)$ tendremos que:

$$(\pi'_w\mu)(\gamma) = \pi'_w(\gamma) = \pi_w(\gamma)$$

   y, dado que $\gamma \notin itv\left(\left(\overline{\tau_i}^n\right) \to \tau'\right)$ y $\pi \in Dcp\left(\left\{\pi_w \mid w \in f\right\}, \left(\overline{\tau_i}^n\right) \to \tau'\right)$, obtenemos que:

$$\pi_w(\gamma) = \pi(\gamma) = (\pi\mu)(\gamma)$$

Por lo tanto, hemos demostrado (5.8). Junto con (5.7) implica que (5.6). Teniendo en cuenta (5.4) y (5.5) obtenemos que:

$$(f, \pi\mu) \in \mathscr{F}\left[\!\left[\left(\overline{\tau_i}^n\right) \to \tau' \textbf{ when } \tau\mu \Leftarrow \alpha\right]\!\right]$$

Finalmente, como $(\pi\mu)(\beta'_i) = (\pi\mu)(\beta_i)$ para cada β_i, se cumple que $\pi\mu \vDash \left\{\overline{\beta'_i \subseteq \beta_i}\right\}$, de lo que se deduce

la proposición:

$$(f,\pi\mu) \in \mathscr{T}\left[\!\left[\left(\left(\overline{\tau_i}^n\right) \to \tau' \textbf{ when } \tau\mu \Leftarrow \alpha\right) \textbf{ when } \overline{\beta'_i \subseteq \beta_i}\right]\!\right]$$

□

En la proposición previa hemos hundido la restricción $\tau \Leftarrow \alpha$ en el tipo del resultado del tipo funcional, pero también podríamos haberlo hundido en cualquiera de los tipos que representan a los parámetros. Esto se debe a lo siguiente:

Proposición 27. *Para cualquier tipo funcional $(\tau_1,\ldots,\tau_n) \to \tau$ y el conjunto de restricciones C, se cumple lo siguiente:*

$$\mathscr{F}[\![(\tau_1,\ldots,\tau_n) \to \tau \textbf{ when } C]\!] = \mathscr{F}[\![(\tau_1,\ldots,\tau_i \textbf{ when } C,\ldots,\tau_n) \to \tau]\!]$$

para toda $i \in \{1..n\}$.

Demostración. Se deduce a partir de la definición de $\mathscr{F}[\![\_]\!]$. □

También podemos hundir restricciones dentro de esquemas de tipo polimórficos:

Proposición 28. *Dado un esquema de tipos polimórfico $\forall\overline{\alpha_i}.\tau$, para cada conjunto de restricciones C tal que $ftv(C) \cap \{\overline{\alpha_i}\} = \emptyset$, se cumple lo siguiente:*

$$\mathscr{T}\left[\!\left[(\forall\overline{\alpha_i}.\tau) \textbf{ when } C\right]\!\right] = \mathscr{S}\left[\!\left[\forall\overline{\alpha_i}.\tau \textbf{ when } C\right]\!\right]$$

Demostración. Supongamos un par (v,π) y una instanciación π' tal que $\pi \equiv \pi'$ (módulo $\{\overline{\alpha_i}\}$):

$$\begin{array}{lll}
 & (v,\pi) \in \mathscr{T}\left[\!\left[(\forall\{\overline{\alpha_i}\}.\tau) \textbf{ when } C\right]\!\right] & \\
\Longleftrightarrow & (v,\pi') \in \mathscr{T}[\![\tau]\!] \wedge \pi \vDash C & \\
\Longleftrightarrow & (v,\pi') \in \mathscr{T}[\![\tau]\!] \wedge \pi' \vDash C & \text{ya que } \pi \equiv \pi' \text{ (módulo } C\text{)} \\
\Longleftrightarrow & (v,\pi') \in \mathscr{T}[\![\tau \textbf{ when } C]\!] & \\
\Longleftrightarrow & (v,\pi) \in \mathscr{S}\left[\!\left[\forall\overline{\alpha_i}.\tau \textbf{ when } C\right]\!\right] &
\end{array}$$

□

Ahora abordaremos los tipos unión. Podemos hundir una restricción de encaje a través de un elemento de una unión de tipos $\cup$-normal, siempre que nos aseguremos de que el resultado sigue siendo $\cup$-normal.

Proposición 29. *Supongamos un tipo $\cup$-normal $\tau_1 \cup \tau_2 \cup \ldots \cup \tau_n$ y una restricción $\tau \Leftarrow \alpha$ tal que $\alpha \notin itv(\tau)$. Si $(v,\pi) \in \mathcal{T}[\![\tau_1 \cup \tau_2 \cup \ldots \cup \tau_n]\!]\, y\, \pi \vDash \tau \Leftarrow \alpha$, entonces:*

$$(v,\pi') \in \mathcal{T}\left[\!\!\left[\left(\tau_1 \textbf{ when } \tau\mu \Leftarrow \alpha\right) \cup \left(\tau_2 \textbf{ when } \overline{\text{none}() \Leftarrow \beta'_i}\right) \cup \ldots \left(\tau_n \textbf{ when } \overline{\text{none}() \Leftarrow \beta'_i}\right) \right]\!\!\right]$$

donde $\left\{\overline{\beta_i}\right\} = itv(\tau) \setminus (itv(\tau_2) \cup \ldots \cup itv(\tau_n))$, $\left\{\overline{\beta'_i}\right\}$ son variables frescas, $\mu = \left[\overline{\beta_i / \beta'_i}\right]$ y π' es una instanciación tal que $\pi \equiv \pi'$ (módulo $\setminus \left\{\overline{\beta'_i}\right\}$) y $\pi' \vDash \overline{\beta'_i \subseteq \beta_i}$.

Demostración. Primero vamos a provar que el tipo resultante,

$$\left(\tau_1 \textbf{ when } \tau\mu \Leftarrow \alpha\right) \cup \left(\tau_2 \textbf{ when } \overline{\text{none}() \Leftarrow \beta'_i}\right) \cup \ldots \left(\tau_n \textbf{ when } \overline{\text{none}() \Leftarrow \beta'_i}\right)$$

es $\cup$-normal. Vamos a definir el conjunto E_1 de la siguiente manera:

$$E_1 = \left(\bigcup_{k=2}^{n} itv\left(\tau_k \textbf{ when } \overline{\text{none}() \Leftarrow \beta'_i}\right)\right) \setminus ftv\left(\tau_1 \textbf{ when } \tau\mu \Leftarrow \alpha\right)$$

y, para todo $j \in \{2..n\}$, definimos E_j del siguiente modo:

$$E_j = \left(itv\left(\tau_1 \textbf{ when } \tau\mu \Leftarrow \alpha\right) \cup \bigcup_{\substack{k \in \{2..n\} \\ k \neq j}} itv\left(\tau_k \textbf{ when } \overline{\text{none}() \Leftarrow \beta'_i}\right) \right) \setminus ftv\left(\tau_j \textbf{ when } \overline{\text{none}() \Leftarrow \beta'_i}\right)$$

Tenemos que demostrar que $E_i = \emptyset$ para cada $i \in \{1..n\}$. Vamos a empezar con E_1. Sabemos que:

$$\begin{aligned} E_1 &= \left(\bigcup_{k=2}^{n} itv\left(\tau_k \textbf{ when } \overline{\text{none}() \Leftarrow \beta'_i}\right)\right) \setminus ftv\left(\tau_1 \textbf{ when } \tau\mu \Leftarrow \alpha\right) \\ &= \left(\bigcup_{k=2}^{n} itv(\tau_k)\right) \setminus ftv\left(\tau_1 \textbf{ when } \tau\mu \Leftarrow \alpha\right) \\ &\subseteq \left(\bigcup_{k=2}^{n} itv(\tau_k)\right) \setminus ftv(\tau_1) \end{aligned}$$

siendo vacío el último de ellos porque $\tau_1 \cup \tau_2 \cup \ldots \tau_n$ es $\cup$-normal. Ahora demostraremos lo mismo con el resto de E_j ($j \in \{2..n\}$), que se puede simplificar de la siguiente manera:

$$E_j = \left(itv\left(\tau_1 \textbf{ when } \tau\mu \Leftarrow \alpha\right) \cup \bigcup_{\substack{k \in \{2..n\} \\ k \neq j}} itv(\tau_k) \right) \setminus \left(ftv\left(\tau_j\right) \cup \left\{\overline{\beta'_i}\right\} \right)$$

Demostraremos por reducción al absurdo que E_j está vacío. Suponiendo una $\gamma \in E_j$. Sabemos que $\gamma \notin ftv\left(\tau_j\right)$ y que γ no es ninguna de las $\overline{\beta'_i}$.

- Si $\gamma \in itv(\tau_k)$ para alguna $k \in \{1..n\}$ y $k \neq j$. En este caso tendríamos que

$$\gamma \in \left(\bigcup_{\substack{k \in \{1..n\} \\ k \neq j}} itv(\tau_k) \right) \setminus ftv(\tau_j)$$

  pero, como $\tau_1 \cup \tau_2 \cup \ldots \cup \tau_n$ es $\cup$-normal, lo último se puede rescribir como $\gamma \in \emptyset$, que conduce a una contradicción.

- Si $\gamma \in itv(\tau\mu)$, como γ no es ninguna de las β'_i, sabemos que γ ha de ser alguna de las variables que hay en $itv(\tau)$ y que no fueron afectadas por el renombramiento μ. Es decir, γ no puede ser ninguna de las β_i. Con mirar a la definición de $\left\{\overline{\beta_i}\right\}$, γ ha de encontrarse en $itv(\tau_k)$ para algún $k \in \{2..n\}$. Supondremos que $\gamma \notin ftv(\tau_j)$, así que estaremos en el caso $\gamma \in itv(\tau_k)$ para algún $k \neq j$, que fue tratado previamente.

Habiendo demostrado que el tipo resultante es $\cup$-normal, vamos a demostrar la proposición. Suponiendo un par $(v,\pi) \in \mathcal{T}[\![\tau_1 \cup \ldots \cup \tau_n]\!]$. Vamos a distinguir entre casos:

- $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$. Dado que las variables β'_i son frescas, sabemos que $\pi\mu \equiv \pi$ (módulo $ftv(\tau_1)$). Por lo tanto, $(v,\pi\mu) \in \mathcal{T}[\![\tau_1]\!]$. Es más, con $\pi \vDash \tau \Leftarrow \alpha$ se deduce que $\pi\mu \vDash \tau\mu \Leftarrow \alpha\mu$, pero como $\alpha \notin itv(\tau)$, entonces α no es ninguna de las β_i, por ello $\pi\mu \vDash \tau\mu \Leftarrow \alpha$. Por consiguiente, obtenemos que $(v,\pi\mu) \in \mathcal{T}[\![\tau_1 \textbf{ when } \tau\mu \Leftarrow \alpha]\!]$, y por ello $(v,\pi\mu)$ pertenece al ($\cup$-normalizado) tipo unión resultante. Obviamente se cumple que $\pi\mu \equiv \pi$ (módulo $\setminus \{\beta'_i\}$) y $\pi\mu \vDash \overline{\beta'_i \subseteq \beta_i}$.

- $(v,\pi) \in \mathcal{T}[\![\tau_j]\!]$ para algún otro $j \neq 1$. Vamos a denotar con π' la instanciación $\pi \setminus \left\{\overline{\beta'_i}\right\}$. Se cumple que $\pi \equiv \pi'$ (módulo $\setminus \left\{\overline{\beta'_i}\right\}$). Dado que las variables β'_i son frescas, sabemos que $(v,\pi') \in \mathcal{T}[\![\tau_j]\!]$. Es más, como $\pi'(\beta'_i) = \emptyset$ para toda β'_i, se cumple que $\pi' \vDash \mathsf{none}() \Leftarrow \beta'_i$. Entonces, se deduce que (v,π') pertenece a $\mathcal{T}\left[\!\left[\tau_j \textbf{ when } \overline{\mathsf{none}() \Leftarrow \beta_i}\right]\!\right]$, y por ello a la semántica del tipo unión resultante. De nuevo, como $\pi'(\beta'_i) = \emptyset$ para cualquier β'_i, se cumple que $\pi' \vDash \overline{\beta'_i \subseteq \beta_i}$.

□

5.3.3. Ínfimo de tipos polimórfico

Para encontrar la mayor cota inferior entre tipos anotados utilizamos la operación del ínfimo, que vamos a detallar en esta sección para el algoritmo de inferencia. Dados dos pares de tipos $\langle \tau;\Gamma \rangle$ y $\langle \tau';\Gamma' \rangle$, usaremos la notación $\langle \tau;\Gamma \rangle \sqcap \langle \tau';\Gamma' \rangle$ para representar el ínfimo entre los dos pares, que definimos de la siguiente forma:

$$\langle \tau;\Gamma \rangle \sqcap \langle \tau';\Gamma' \rangle = \begin{cases} \langle \tau;\Gamma \sqcap \Gamma' \rangle & \text{si } \tau = \tau' \\ \langle \delta;\Gamma\,[\tau \Leftarrow \delta] \sqcap \Gamma'\,[\tau' \Leftarrow \delta] \rangle & \text{e.o.c.} \end{cases}$$

La operación de ínfimo $\Gamma \sqcap \Gamma'$ usada es la definida en la sección 4.4. Nótese que si bien podría reducirse la operación al segundo caso, tendremos en cuenta también el primer caso a la hora de implementar la codificación de esta operación, para simplificar la generación de restricciones en la medida de lo posible.

El operador ínfimo $\sqcap$ entre tipos recibe dos tipos τ_1 y τ_2 y devuelve un tipo τ que cumple $\mathscr{T}[\![\tau]\!]_1 = \mathscr{T}[\![\tau_1]\!]_1 \cap \mathscr{T}[\![\tau_2]\!]_1$. La definición de este operador requiere dos condiciones: que los tipos sean $\cup$-normalizados y que τ_1 y τ_2 no tengan variables de tipo en común. En principio no se supone que los tipos recibidos estén normalizados como vimos en la sección 5.1.1, por lo que los tipos condicionales con restricciones podrían estar en posiciones no contempladas en los tipos normalizados. Asimismo, el tipo τ resultante del operador de ínfimo podría no ser normalizado, por lo que podría tener restricciones en cualquier parte de su árbol sintáctico abstracto, y en ese caso habría que normalizarlo posteriormente. Entonces, dados dos tipos τ y τ' $\cup$-normalizados, tales que que $ftv(\tau) \cap ftv(\tau') = \emptyset$, usaremos la notación $\tau \sqcap \tau'$ para representar el ínfimo entre tipos, que definiremos, a continuación, mediante una serie de ecuaciones. Aunque algunas ecuaciones pueden solaparse, deben interpretarse en orden secuencial. Es decir, para calcular el ínfimo entre dos tipos τ y τ', debe aplicarse la primera ecuación cuyo lado izquierdo encaje con $\tau \sqcap \tau'$, o bien con $\tau' \sqcap \tau$, ya que la operación de ínfimo es conmutativa.

1. $\tau \sqcap \texttt{none()} = \texttt{none()}$
2. $\tau \sqcap \texttt{any()} = \tau$
3. $c \sqcap c = c$
4. $c \sqcap B = \begin{cases} c & c \in \mathscr{B}[\![B]\!] \\ \texttt{none()} & c \notin \mathscr{B}[\![B]\!] \end{cases}$
5. $B \sqcap B' = \begin{cases} B & \mathscr{B}[\![B]\!] \subseteq \mathscr{B}[\![B']\!] \\ B' & \mathscr{B}[\![B']\!] \subseteq \mathscr{B}[\![B]\!] \\ \texttt{none()} & \text{e.o.c.} \end{cases}$
6. $\tau \sqcap \alpha = \alpha$ **when** $\tau \Leftarrow \alpha$
7. $\tau \sqcap (\tau' \textbf{ when } C) = (\tau \sqcap \tau') \textbf{ when } C$
8. $\{\tau_1, \ldots, \tau_n\} \sqcap \{\tau'_1, \ldots \tau'_n\} = \{\tau_1 \sqcap \tau'_1, \ldots, \tau_n \sqcap \tau'_n\}$
9. $\texttt{nelist}(\tau_1, \tau'_1) \sqcap \texttt{nelist}(\tau_2, \tau'_2) = \texttt{nelist}(\tau_1 \sqcap \tau_2, \tau'_1 \sqcap \tau'_2)$
   $\cup\ \texttt{nelist}(\tau_1\mu_1 \sqcap \tau_2, \texttt{nelist}(\tau_1\mu_2, \tau'_1) \sqcap \tau'_2)$ **when** $\left\{\overline{\alpha_i \subseteq \delta_i^1 \cup \delta_i^2}, \overline{\delta_i^1 \subseteq \alpha_i}, \overline{\delta_i^2 \subseteq \alpha_i}\right\}$
   $\cup\ \texttt{nelist}(\tau_1 \sqcap \tau_2\mu_3, \tau'_1 \sqcap \texttt{nelist}(\tau_2\mu_4, \tau'_2))$ **when** $\left\{\overline{\beta_i \subseteq \delta_i^3 \cup \delta_i^4}, \overline{\delta_i^3 \subseteq \beta_i}, \overline{\delta_i^4 \subseteq \beta_i}\right\}$

   donde $\{\overline{\alpha_i}\} = ftv(\tau_1), \left\{\overline{\beta_i}\right\} = ftv(\tau_2), \mu_1 = \left[\overline{\alpha_i/\delta_i^1}\right], \mu_2 = \left[\overline{\alpha_i/\delta_i^2}\right], \mu_3 = \left[\overline{\beta_i/\delta_i^3}\right],$
   $\mu_4 = \left[\overline{\beta_i/\delta_i^4}\right], \left\{\overline{\delta_i^1}, \overline{\delta_i^2}, \overline{\delta_i^3}, \overline{\delta_i^4}\right\}$ son variables frescas

10. $\left(\left(\overline{\tau_i}\right) \to \tau\right) \sqcap \left(\left(\overline{\tau'_i}\right) \to \tau'\right) = \left(\left(\overline{\tau_i \sqcap \tau'_i}\right) \to \tau \sqcap \tau'\right)$
11. $\left(\forall \overline{\alpha_i}.\tau_1\right) \sqcap \left(\forall \overline{\beta_i}.\tau_2\right) = \forall \overline{\alpha_i}, \overline{\beta_i}.(\tau_1 \sqcap \tau_2)$
    donde $\{\overline{\alpha_i}\} \cap \left\{\overline{\beta_i}\right\} = \emptyset, \{\overline{\alpha_i}\} \cap ftv(\tau_2) = \emptyset, \left\{\overline{\beta_i}\right\} \cap ftv(\tau_1) = \emptyset$
12. $\bigsqcup_{i=1}^{n} \sigma_i \sqcap \bigsqcup_{j=1}^{m} \sigma'_j = \bigsqcup_{\substack{i \in \{1..n\} \\ j \in \{1..m\}}} \left(\sigma_i \sqcap \sigma'_j\right)$
13. $\tau \sqcap \left(\bigcup_{i=1}^{n} \tau_i\right) = \bigcup_{i=1}^{n} (\tau \sqcap \tau_i)$
14. En otro caso: $\tau \sqcap \tau' = \mathsf{none}()$

Teniendo en cuenta lo anterior, se cumple la siguiente proposición para el ínfimo entre tipos:

Proposición 30. *Supongamos los tipos τ_1 y τ_2, tales que $ftv(\tau_1) \cap ftv(\tau_1) = \emptyset$. Para todo $v \in$ **DVal**, sea π una instanciación tal que $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$ y $(v,\pi) \in \mathcal{T}[\![\tau_2]\!]$. Entonces existe una instanciación π' tal que $(v,\pi') \in \mathcal{T}[\![\tau_1 \sqcap \tau_2]\!]$ y $\pi' \equiv \pi$ (módulo $\setminus X$), donde X es el conjunto de variables de tipo frescas generadas.*

Demostración. Por inducción sobre la estructura de τ_1 y τ_2. Para demostrarlo vamos a distinguir cada caso de la operación:

- **Caso** $\tau_2 = \mathsf{none}()$

  No existe ningún par (v,π) tal que $(v,\pi) \in \mathcal{T}[\![\mathsf{none}()]\!]$. Por lo tanto se cumple trivialmente.

- **Caso** $\tau_2 = \mathsf{any}()$

  Suponemos un par $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$ y que $(v,\pi) \in \mathcal{T}[\![\mathsf{any}()]\!]$. Entonces $(v,\pi) \in \mathcal{T}[\![\tau_1 \sqcap \mathsf{any}()]\!] = \mathcal{T}[\![\tau_1]\!]$.

- **Caso** $\tau_1 = c$ **y** $\tau_2 = c$

  La demostración es trivial.

- **Caso** $\tau_1 = c$ **y** $\tau_2 = B_2$

  Tenemos dos casos posibles:

  - Si $c \in \mathcal{B}[\![B_2]\!]$, sabemos que $\mathcal{T}[\![c]\!]$ está contenido en $\mathcal{T}[\![B_2]\!]$, por lo que podemos suponer un par (c,π) tal que $(c,\pi) \in \mathcal{T}[\![c]\!]$ y $(c,\pi) \in \mathcal{T}[\![B_2]\!]$, para cualquier instanciación π. Por lo tanto, tendremos que $(c,\pi) \in \mathcal{T}[\![c \sqcap B_2]\!] = \mathcal{T}[\![c]\!]$.
  - Si $c \notin \mathcal{B}[\![B_2]\!]$, no existe ninguna $(c,\pi) \in \mathcal{T}[\![c]\!]$ que pueda existir en $\mathcal{T}[\![B_2]\!]$. Por lo tanto se devuelve el conjunto vacío, representado por $\mathsf{none}()$, y la propiedad se cumple trivialmente.

- **Caso** $\tau_1 = B_1$ **y** $\tau_2 = B_2$

  Teniendo en cuenta la propiedad 3.2, sabemos que:

- Si $\mathscr{B}[\![B_1]\!] \subseteq \mathscr{B}[\![B_2]\!]$, supongamos que tenemos un par $(v,\pi) \in \mathscr{T}[\![B_1]\!]$, entonces sabemos que $(v,\pi) \in \mathscr{T}[\![B_2]\!]$, y por consiguiente $(v,\pi) \in \mathscr{T}[\![B_1 \sqcap B_2]\!] = \mathscr{T}[\![B_1]\!]$.
- Si $\mathscr{B}[\![B_2]\!] \subseteq \mathscr{B}[\![B_1]\!]$, supongamos que tenemos un par $(v,\pi) \in \mathscr{T}[\![B_2]\!]$, entonces sabemos que $(v,\pi) \in \mathscr{T}[\![B_1]\!]$, y por consiguiente $(v,\pi) \in \mathscr{T}[\![B_1 \sqcap B_2]\!] = \mathscr{T}[\![B_2]\!]$.
- Si $\mathscr{B}[\![B_1]\!] \cap \mathscr{B}[\![B_2]\!] = \emptyset$, sabemos que no existe ninguna $(v,\pi) \in \mathscr{T}[\![B_1]\!]$ que pueda existir en $\mathscr{T}[\![B_2]\!]$. Por lo tanto se devuelve el conjunto vacío, representado por `none()`, y la propiedad se cumple trivialmente.

- **Caso** $\tau_2 = \alpha$

  Suponemos un par (v,π), tal que $(v,\pi) \in \mathscr{T}[\![\tau_1]\!]$ y $(v,\pi) \in \mathscr{T}[\![\alpha]\!]$, donde $\pi(\alpha) = \{v\}$. Por la definición semántica de las restricciones, como tenemos $\pi(\alpha) = \{v\}$ y $(v,\pi) \in \mathscr{T}[\![\tau_1]\!]$, obtendremos que se cumple $\pi \vDash \tau_1 \Leftarrow \alpha$. Con esta restricción que acabamos de obtener, tendremos que $(v,\pi) \in \mathscr{T}[\![\alpha \textbf{ when } \tau_1 \Leftarrow \alpha]\!] = \mathscr{T}[\![\tau_1 \sqcap \alpha]\!]$.

- **Caso** $\tau_2 = \tau$ **when** C

  Suponemos un par (v,π), tal que $(v,\pi) \in \mathscr{T}[\![\tau_1]\!]$ y $(v,\pi) \in \mathscr{T}[\![\tau \textbf{ when } C]\!]$. Por la semántica de τ **when** C obtendremos que $(v,\pi) \in \mathscr{T}[\![\tau]\!]$ y $\pi \vDash C$. Por hipótesis de inducción, tomando $(v,\pi) \in \mathscr{T}[\![\tau_1]\!]$ y $(v,\pi) \in \mathscr{T}[\![\tau]\!]$, tendremos que $(v,\pi') \in \mathscr{T}[\![\tau_1 \sqcap \tau]\!]$, para algún $\pi' \equiv \pi$ (módulo $\setminus X'$), donde X' es el conjunto de variables de tipo frescas generadas por $\tau_1 \sqcap \tau$. Definimos la instanciación π° del siguiente modo:

  $$\forall \alpha \in \textbf{TypeVar}.\, \pi^\circ(\alpha) = \begin{cases} \pi'(\alpha) & \text{si } \alpha \in X' \\ \pi(\alpha) & \text{e.o.c.} \end{cases}$$

  Sabemos que $ftv(C) \cap X' = \emptyset$, porque $\tau_1 \sqcap \tau$ solo genera variables frescas. Por lo tanto $\pi \equiv \pi^\circ$ (módulo $ftv(C)$) y, al cumplirse $\pi \vDash C$, se cumple entonces $\pi^\circ \vDash C$. Además, tenemos que $\pi^\circ \equiv \pi'$ (módulo X') y $\pi^\circ \equiv \pi \equiv \pi'$ (módulo $\setminus X'$), por lo tanto se cumple que $\pi^\circ \equiv \pi'$ (módulo $ftv(\tau_1 \sqcap \tau)$), obteniendo que si $(v,\pi') \in \mathscr{T}[\![\tau_1 \sqcap \tau]\!]$, entonces se cumple que $(v,\pi^\circ) \in \mathscr{T}[\![\tau_1 \sqcap \tau]\!]$. Finalmente, porque $X = X'$, obtendremos que $\pi^\circ \equiv \pi$ (módulo $\setminus X$) y $(v,\pi^\circ) \in \mathscr{T}[\![(\tau_1 \sqcap \tau) \textbf{ when } C]\!] = \mathscr{T}[\![\tau_1 \sqcap (\tau \textbf{ when } C)]\!]$.

- **Caso** $\tau_1 = \{\tau'_1, \ldots \tau'_n\}$ **y** $\tau_2 = \{\tau''_1, \ldots \tau''_n\}$

  Suponemos un par (v,π), tal que $(v,\pi) \in \mathscr{T}[\![\{\tau'_1, \ldots \tau'_n\}]\!]$ y $(v,\pi) \in \mathscr{T}[\![\{\tau''_1, \ldots \tau''_n\}]\!]$. Por la definición semántica sabemos que v tiene la forma $\left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right)$. Por lo tanto sabemos que $(v_i,\pi) \in \mathscr{T}[\![\tau'_i]\!]$ y $(v_i,\pi) \in \mathscr{T}[\![\tau''_i]\!]$, para cada $i \in \{1..n\}$, y aplicando la hipótesis de inducción—a cada pareja de componentes—obtendremos que existe una instanciación π'_i tal que $(v_i,\pi'_i) \in \mathscr{T}[\![\tau'_i \sqcap \tau''_i]\!]$ y $\pi'_i \equiv \pi$ (módulo $\setminus X_i$), donde X_i es el conjunto de variables frescas generadas por cada $\tau'_i \sqcap \tau''_i$, para cada $i \in \{1..n\}$. Sabemos que se cumple que $X_i \cap X_j = \emptyset$, para todo $i, j \in \{1..n\}$ tal que $i \neq j$, por el hecho de ser variables de tipo frescas. Ahora definiremos la instanciación π°

del siguiente modo:

$$\forall \alpha \in \mathbf{TypeVar}.\ \pi^\circ(\alpha) = \begin{cases} \pi'_1(\alpha) & \text{si } \alpha \in X_1 \\ \quad\vdots & \\ \pi'_n(\alpha) & \text{si } \alpha \in X_n \\ \pi(\alpha) & \text{e.o.c.} \end{cases}$$

Sea una variable de tipo $\alpha \in ftv\big(\tau'_i \sqcap \tau''_i\big)$, para algún $i \in \{1..n\}$:

- Si $\alpha \in X_i$, para algún $i \in \{1..n\}$, entonces $\pi^\circ(\alpha) = \pi'_i(\alpha)$.
- Si $\alpha \notin X_i$, para todo $i \in \{1..n\}$, entonces $\alpha \notin \bigcup_{i=1}^{n} X_i$ y por lo tanto $\pi^\circ(\alpha) = \pi(\alpha) = \pi'_i(\alpha)$.

Llegados a este punto, para todo $i \in \{1..n\}$, obtenemos que $\pi^\circ \equiv \pi'_i$ (módulo $ftv\big(\tau'_i \sqcap \tau''_i\big)$) y como tenemos que $(v_i, \pi_i) \in \mathcal{T}\,[\![\tau'_i \sqcap \tau''_i]\!]$, entonces se cumple que $(v_i, \pi^\circ) \in \mathcal{T}\,[\![\tau'_i \sqcap \tau''_i]\!]$ y por la definición semántica de los tipos tupla obtendremos $(v, \pi^\circ) \in \mathcal{T}\,[\![\{\tau'_1 \sqcap \tau''_1, \ldots, \tau'_n \sqcap \tau''_n\}]\!]$. Además, para toda variable de tipo $\alpha \notin X = X_1 \cup \cdots \cup X_n$, tenemos que $\pi^\circ(\alpha) = \pi(\alpha)$, por lo que $\pi^\circ \equiv \pi$ (módulo $\setminus X$).

- **Caso** $\tau_1 = \texttt{nelist}(\tau'_1, \tau''_1)$ **y** $\tau_2 = \texttt{nelist}(\tau'_2, \tau''_2)$

  Por un lado, suponemos $(v, \pi) \in \mathcal{T}\,[\![\texttt{nelist}(\tau'_1, \tau''_1)]\!]$, donde $v = [v_1, \ldots, v_n \mid v']$ y tenemos que $\pi \in Dcp\big(\{\overline{\pi_i}^{\,n}\}, \tau'_1\big)$ con $(v_i, \pi_i) \in \mathcal{T}\,[\![\tau'_1]\!]$, para todo $i \in \{1..n\}$, y $(v', \pi) \in \mathcal{T}\,[\![\tau''_1]\!]$. Por otro lado, tenemos $(v, \pi) \in \mathcal{T}\,[\![\texttt{nelist}(\tau'_2, \tau''_2)]\!]$, donde $v = [v'_1, \ldots, v'_m \mid v'']$ y tenemos que la instanciación $\pi \in Dcp\Big(\Big\{\overline{\pi'_j}^{\,m}\Big\}, \tau'_2\Big)$ con $(v'_j, \pi'_j) \in \mathcal{T}\,[\![\tau'_2]\!]$, para todo $j \in \{1..m\}$, y $(v'', \pi) \in \mathcal{T}\,[\![\tau''_2]\!]$. Tendremos que distinguir entre las diferentes combinaciones de tamaños entre n y m:

  - **Caso** $n = m$

    Cuando son iguales los tamaños, sabemos que $v_i = v'_i$, para todo $i \in \{1..n\}$, y que $v' = v''$. Entonces, para cada $i \in \{1..n\}$, definimos la instanciación π°_i del siguiente modo:

    $$\forall \alpha \in \mathbf{TypeVar}.\ \pi^\circ_i(\alpha) = \begin{cases} \pi_i(\alpha) & \text{si } \alpha \in ftv\big(\tau'_1\big) \\ \pi'_i(\alpha) & \text{e.o.c.} \end{cases}$$

    Como se cumple que $ftv\big(\tau'_1\big) \cap ftv\big(\tau'_2\big) = \emptyset$, tendremos para cada $i \in \{1..n\}$ que:

    $$\pi^\circ_i \equiv \pi_i \text{ (módulo } ftv\big(\tau'_1\big)) \qquad \wedge \qquad \pi^\circ_i \equiv \pi'_i \text{ (módulo } ftv\big(\tau'_2\big))$$

    por lo que $(v_i, \pi^\circ_i) \in \mathcal{T}\,[\![\tau'_1]\!]$ y $(v'_i, \pi^\circ_i) = (v_i, \pi^\circ_i) \in \mathcal{T}\,[\![\tau'_2]\!]$. Por hipótesis de inducción, con lo anterior, para cada $i \in \{1..n\}$, obtendremos que $(v_i, \pi^\bullet_i) \in \mathcal{T}\,[\![\tau'_1 \sqcap \tau'_2]\!]$ para algún $\pi^\bullet_i \equiv \pi^\circ_i$ (módulo $\setminus X'$), donde X' es el conjunto de variables frescas generadas por $\tau'_1 \sqcap \tau'_2$. Por otro lado, al tener que $(v', \pi) \in \mathcal{T}\,[\![\tau''_1]\!]$ y $(v'', \pi) = (v', \pi) \in \mathcal{T}\,[\![\tau''_2]\!]$, por hipótesis de inducción obtendremos que $(v', \pi') \in \mathcal{T}\,[\![\tau''_1 \sqcap \tau''_2]\!]$ para algún $\pi' \equiv \pi$ (módulo $\setminus X''$), donde X'' es el

conjunto de variables frescas generadas por $\tau_1'' \sqcap \tau_2''$. Ahora definimos la instanciación $\pi^\bullet = \bigcup_{i=1}^{n} \pi_i^\bullet$ y la instanciación $\pi^\star$ del siguiente modo:

$$\forall \alpha \in \textbf{TypeVar}.\ \pi^\star(\alpha) = \begin{cases} \pi'(\alpha) & \text{si } \alpha \in X'' \\ \pi^\bullet(\alpha) & \text{e.o.c.} \end{cases}$$

Sea una variable de tipo $\alpha \in ftv\left(\tau_1'' \sqcap \tau_2''\right)$:

- Si $\alpha \in X''$, entonces se cumple que $\pi^\star(\alpha) = \pi'(\alpha)$.
- Si $\alpha \notin X''$, se cumplirá entonces que $\pi^\star(\alpha) = \pi^\bullet(\alpha) = \bigcup_{i=1}^{n} \pi_i^\bullet(\alpha) = \bigcup_{i=1}^{n} \pi_i^\circ(\alpha)$ porque $ftv\left(\tau_1'' \sqcap \tau_2''\right) \cap X' = \emptyset$.
  - Si $\alpha \in ftv\left(\tau_1'\right)$, entonces $\pi^\star(\alpha) = \bigcup_{i=1}^{n} \pi_i^\circ(\alpha) = \bigcup_{i=1}^{n} \pi_i^\circ(\alpha) = \pi(\alpha) = \pi'(\alpha)$.
  - Si $\alpha \notin ftv\left(\tau_1'\right)$, entonces $\pi^\star(\alpha) = \bigcup_{i=1}^{n} \pi_i^\circ(\alpha) = \bigcup_{i=1}^{n} \pi_i'(\alpha) = \pi(\alpha) = \pi'(\alpha)$.

En los dos últimos casos tenemos que $\pi(\alpha) = \pi'(\alpha)$, porque $\pi \equiv \pi'$ (módulo $\setminus X''$). Por lo tanto, tenemos que $\pi^\star \equiv \pi'$ (módulo $ftv\left(\tau_1'' \sqcap \tau_2''\right)$) y que $(\nu', \pi^\star) \in \mathcal{T}\left[\!\left[\tau_1'' \sqcap \tau_2''\right]\!\right]$. Ahora definimos la instanciación $\pi_i^\star$, para todo $i \in \{1..n\}$, del siguiente modo:

$$\forall \alpha \in \textbf{TypeVar}.\ \pi_i^\star(\alpha) = \begin{cases} \pi_i^\bullet(\alpha) & \text{si } \alpha \notin X'' \\ \pi'(\alpha) & \text{si } \alpha \in X'' \end{cases}$$

Como $X'' \cap ftv\left(\tau_1' \sqcap \tau_2'\right) = \emptyset$, sabemos que se cumple $\pi_i^\star \equiv \pi_i^\bullet$ (módulo $ftv\left(\tau_1' \sqcap \tau_2'\right)$) y que el par $(\nu_i, \pi_i^\star) \in \mathcal{T}\left[\!\left[\tau_1' \sqcap \tau_2'\right]\!\right]$. Lo siguiente a demostrar es que $\pi^\star = Dcp\left(\left\{\overline{\pi_i^\star}^{\,n}\right\}, \tau_1' \sqcap \tau_2'\right)$ se cumple. Para ello, sea una variable de tipo $\alpha \in \textbf{TypeVar}$.

- Si $\alpha \in X''$, entonces $\pi^\star(\alpha) = \pi'(\alpha) = \bigcup_{i=1}^{n} \pi_i'(\alpha) = \bigcup_{i=1}^{n} \pi_i^\star(\alpha)$.
- Si $\alpha \notin X''$, entonces $\pi^\star(\alpha) = \pi^\bullet(\alpha) = \bigcup_{i=1}^{n} \pi_i^\bullet(\alpha) = \bigcup_{i=1}^{n} \pi_i^\star(\alpha)$.

Por lo tanto, para toda $\alpha \in \textbf{TypeVar}$, se cumple que $\pi^\star(\alpha) = \bigcup_{i=1}^{n} \pi_i^\star(\alpha)$. Además, sea $\alpha \notin itv\left(\tau_1' \sqcap \tau_2'\right)$, entonces $\alpha \notin itv\left(\tau_1'\right) \cup itv\left(\tau_2'\right)$, porque $itv\left(\tau_1'\right) \cup itv\left(\tau_2'\right) \subseteq itv\left(\tau_1' \sqcap \tau_2'\right)$. Por lo tanto:

- Si $\alpha \in X''$, entonces $\pi_i^\star(\alpha) = \pi'(\alpha) = \pi^\star(\alpha)$, para cada $i \in \{1..n\}$.
- Si $\alpha \notin X''$ y $\alpha \in ftv(\tau_1)$, para cada $i \in \{1..n\}$, obtendremos:
  1. $\pi_i^\star(\alpha) = \pi_i^\bullet(\alpha)$.
  2. $\pi_i^\bullet(\alpha) = \pi_i^\circ(\alpha)$, porque $ftv(\tau_1) \cap X' = \emptyset \Longrightarrow \alpha \notin X'$.
  3. $\pi_i^\circ(\alpha) = \pi_i(\alpha)$, porque $\alpha \in ftv(\tau_1)$.
  4. $\pi_i(\alpha) = \pi(\alpha)$, porque $\pi \in Dcp\left(\left\{\overline{\pi_i}^{\,n}\right\}, \tau_1\right)$ y $\alpha \notin itv(\tau_1)$.
  5. $\pi(\alpha) = \bigcup_{i=1}^{n} \pi_i(\alpha) = \bigcup_{i=1}^{n} \pi_i^\circ(\alpha)$, porque $\alpha \in itv(\tau_1)$.
  6. $\bigcup_{i=1}^{n} \pi_i^\circ(\alpha) = \bigcup_{i=1}^{n} \pi_i^\bullet(\alpha)$, porque $\alpha \notin X'$.
  7. $\bigcup_{i=1}^{n} \pi_i^\bullet(\alpha) = \pi^\bullet(\alpha) = \pi^\star(\alpha)$, porque $\alpha \notin X''$.
- Si $\alpha \notin X''$ y $\alpha \notin ftv(\tau_1)$, para cada $i \in \{1..n\}$, obtendremos:

1. $\pi_i^\star(\alpha) = \pi_i^\bullet(\alpha)$.
2. $\pi_i^\bullet(\alpha) = \pi_i^\circ(\alpha)$, porque $ftv(\tau_2) \cap X' = \emptyset \Longrightarrow \alpha \notin X'$.
3. $\pi_i^\circ(\alpha) = \pi_i'(\alpha)$, porque $\alpha \in ftv(\tau_2)$.
4. $\pi_i'(\alpha) = \pi(\alpha)$, porque $\pi \in Dcp\left(\left\{\overline{\pi_i'}^n\right\}, \tau_2\right)$ y $\alpha \notin itv(\tau_2)$.
5. $\pi(\alpha) = \bigcup_{i=1}^n \pi_i'(\alpha) = \bigcup_{i=1}^n \pi_i^\circ(\alpha)$, porque $\alpha \in itv(\tau_2)$.
6. $\bigcup_{i=1}^n \pi_i^\circ(\alpha) = \bigcup_{i=1}^n \pi_i^\bullet(\alpha)$, porque $\alpha \notin X'$.
7. $\bigcup_{i=1}^n \pi_i^\bullet(\alpha) = \pi^\bullet(\alpha) = \pi^\star(\alpha)$, porque $\alpha \notin X''$.

Por lo tanto $\pi^\star \equiv \pi_i^\star$ (módulo $itv\left(\tau_1' \sqcap \tau_2'\right)$), para todo $i \in \{1..n\}$, y por consiguiente $\pi^\star = Dcp\left(\left\{\overline{\pi_i^\star}^n\right\}, \tau_1' \sqcap \tau_2'\right)$, que junto con $(v_i, \pi_i^\star) \in \mathcal{T}\left[\!\left[\tau_1' \sqcap \tau_2'\right]\!\right]$ y $(v', \pi^\star) \in \mathcal{T}\left[\!\left[\tau_1'' \sqcap \tau_2''\right]\!\right]$, obtendremos que $(v, \pi^\star) \in \mathcal{T}\left[\!\left[\texttt{nelist}(\tau_1' \sqcap \tau_2', \tau_1'' \sqcap \tau_2'')\right]\!\right]$. Además, si $\alpha \notin X'$ y $\alpha \notin X''$, entonces $\pi^\star(\alpha) = \pi^\bullet(\alpha) = \bigcup_{i=1}^n \pi_i^\bullet(\alpha) = \bigcup_{i=1}^n \pi_i^\circ(\alpha)$:

- Si $\alpha \in ftv(\tau_1)$, entonces $\pi^\star(\alpha) = \bigcup_{i=1}^n \pi_i^\circ(\alpha) = \bigcup_{i=1}^n \pi_i(\alpha) = \pi(\alpha)$.
- Si $\alpha \notin ftv(\tau_1)$, entonces $\pi^\star(\alpha) = \bigcup_{i=1}^n \pi_i^\circ(\alpha) = \bigcup_{i=1}^n \pi_i'(\alpha) = \pi(\alpha)$.

Por lo tanto $\pi^\star \equiv \pi$ (módulo $\setminus \left(X' \cup X''\right)$).

- **Caso** $n > m$

  En este caso la forma de v es la siguiente $v = [v_1, \ldots, v_m, \ldots, v_n \mid v']$ y $v = [v_1', \ldots, v_m' \mid v'']$. Entonces $v_i' = v_i$, para cada $i \in \{1..m\}$, y $v'' = [v_{m+1}, \ldots, v_n \mid v']$. Suponemos una sustituciones de variables de tipo $\mu_1 = \left[\overline{\alpha_i / \delta_i^1}\right]$ y $\mu_2 = \left[\overline{\alpha_i / \delta_i^2}\right]$, donde $\{\overline{\alpha_i}\} = ftv(\tau_1)$ y $\overline{\delta_i^1}, \overline{\delta_i^2}$ son variables de tipo frescas. Con $(v_i, \pi_i) \in \mathcal{T}\left[\!\left[\tau_1'\right]\!\right]$ y $(v_i, \pi_i') \in \mathcal{T}\left[\!\left[\tau_2'\right]\!\right]$, para todo $i \in \{1..m\}$, aplicando la sustitución μ_1, tendremos que $(v_i, \pi_i\mu_1) \in \mathcal{T}\left[\!\left[\tau_1'\mu_1\right]\!\right]$. Definimos la instanciación π_i°, para todo $i \in \{1..m\}$, de la siguiente manera:

$$\forall \alpha \in \textbf{TypeVar}.\ \pi_i^\circ(\alpha) = \begin{cases} (\pi_i\mu_1)(\alpha) & \text{si } \alpha \notin ftv\left(\tau_1'\mu_1\right) \\ \pi_i'(\alpha) & \text{e.o.c.} \end{cases}$$

  Siendo obvio que $\pi_i^\circ \equiv \pi_i\mu_1$ (módulo $ftv\left(\tau_1'\mu_1\right)$), para cada $i \in \{1..m\}$, por lo tanto $(v_i, \pi_i^\circ) \in \mathcal{T}\left[\!\left[\tau_1'\mu_1\right]\!\right]$. Además se cumple que $\pi_i^\circ \equiv \pi_i'$ (módulo $ftv\left(\tau_2'\right)$), para cada $i \in \{1..m\}$, porque $ftv\left(\tau_1'\mu_1\right) \cap ftv\left(\tau_2'\right) = \emptyset$, ya que $ftv\left(\tau_1'\mu_1\right) \subseteq \left\{\overline{\delta_i^1}\right\}$, que no están en τ_2'. Entonces, para cada $i \in \{1..m\}$, tendremos que $(v_i, \pi_i^\circ) \in \mathcal{T}\left[\!\left[\tau_2'\right]\!\right]$, que junto a $(v_i, \pi_i^\circ) \in \mathcal{T}\left[\!\left[\tau_1'\mu_1\right]\!\right]$, por hipótesis de inducción obtendremos que $(v_i, \pi_i^\bullet) \in \mathcal{T}\left[\!\left[\tau_1'\mu_1 \sqcap \tau_2'\right]\!\right]$, para algún $\pi_i^\bullet \equiv \pi_i^\circ$ (módulo $\setminus X'$), donde X' son las variables frescas generadas por $\tau_1'\mu_1 \sqcap \tau_2'$.

  Por otro lado, para todo $i \in \{m+1..n\}$, con $(v_i, \pi_i) \in \mathcal{T}\left[\!\left[\tau_1'\right]\!\right]$, aplicando la sustitución μ_2, obtendremos que $(v_i, \pi_i\mu_2) \in \mathcal{T}\left[\!\left[\tau_1'\mu_2\right]\!\right]$. Por lo que definimos la instanciación π_i^+, para todo $i \in \{m+1..n\}$, de la siguiente manera:

$$\forall \alpha \in \textbf{TypeVar}.\ \pi_i^+(\alpha) = \begin{cases} (\pi_i\mu_2)(\alpha) & \text{si } \alpha \notin ftv\left(\tau_1'\mu_2\right) \\ \pi(\alpha) & \text{e.o.c.} \end{cases}$$

Entonces, por ser $\pi_i^+ \equiv \pi_i\mu_2$ (módulo $ftv(\tau_1'\mu_2)$), se tiene que $(v_i, \pi_i^+) \in \mathcal{T}[\![\tau_1'\mu_2]\!]$, para cada $i \in \{m+1..n\}$. Ahora definimos $\pi^+ = \bigcup_{i=m+1}^{n} \pi_i^+$. Sabemos que $(v', \pi) \in \mathcal{T}[\![\tau_1'']\!]$ y que para toda variable $\alpha \in ftv(\tau_1'')$ se cumple que $\alpha \notin ftv(\tau_1'\mu_2)$, ya que $ftv(\tau_1'\mu_2) \subseteq \left\{\overline{\delta_i^2}\right\}$, que son variables de tipo frescas. Por lo tanto, si $\alpha \in ftv(\tau_1'')$, entonces $\pi^+(\alpha) = \bigcup_{i=m+1}^{n} \pi_i^+(\alpha) = \bigcup_{i=m+1}^{n} \pi_i(\alpha) = \pi(\alpha)$, de lo que obtendremos que $\pi^+ \equiv \pi$ (módulo $ftv(\tau_1'')$) y por consiguiente $(v', \pi^+) \in \mathcal{T}[\![\tau_1'']\!]$. Además, suponemos que $\alpha \notin itv(\tau_1'\mu_2)$. Se cumple que $\mu_2^{-1} \notin ftv(\tau_1')$. Por lo tanto, para todo $i \in \{m+1..n\}$ y $\alpha \in ftv(\tau_1'\mu_2) \setminus itv(\tau_1'\mu_2)$, tenemos que $\pi_i^+(\alpha) = (\pi_i\mu_2)(\alpha) = \pi_i(\mu_2^{-1}(\alpha)) = \pi(\alpha)$, porque $\pi \in Dcp(\{\overline{\pi_i}^n\}, \tau_1)$, y si $\alpha \notin ftv(\tau_1'\mu_2)$, entonces $\pi_i^+(\alpha) = \pi(\alpha)$. Por lo tanto, para todo $i \in \{m+1..n\}$, si $\alpha \notin itv(\tau_1'\mu_2)$, entonces $\pi_i^+(\alpha) = \bigcup_{i=m+1}^{n} \pi_i^+(\alpha) = \bigcup_{i=m+1}^{n} \pi_i(\alpha) = \pi(\alpha) = \pi_i(\alpha)$, obteniendo así que $\pi^+ \in Dcp\left(\left\{\overline{\pi_i^+}^n\right\}, \tau_1'\mu_2\right)$, que junto con $(v', \pi^+) \in \mathcal{T}[\![\tau_1'']\!]$ y $(v_i, \pi_i^+) \in \mathcal{T}[\![\tau_1'\mu_2]\!]$ se tiene que $([v_{m+1}, \ldots, v_n \mid v'], \pi^+) \in \mathcal{T}[\![\texttt{nelist}(\tau_1'\mu_2, \tau_1'')]\!]$, y como $v'' = [v_{m+1}, \ldots, v_n \mid v']$, podremos transformar este último resultado para obtener que $(v'', \pi^+) \in \mathcal{T}[\![\texttt{nelist}(\tau_1'\mu_2, \tau_1'')]\!]$.

Por otro lado, sabemos que $(v'', \pi) \in \mathcal{T}[\![\tau_2'']\!]$, y como $ftv(\tau_1'\mu_2) \cap ftv(\tau_2'') = \emptyset$, se cumple que $\pi^+ \equiv \pi$ (módulo $ftv(\tau_2'')$) y por lo tanto $(v'', \pi^+) \in \mathcal{T}[\![\tau_2'']\!]$, que junto al resultado previo de $(v'', \pi^+) \in \mathcal{T}[\![\texttt{nelist}(\tau_1'\mu_2, \tau_1'')]\!]$, por hipótesis de inducción, obtendremos que $(v'', \pi') \in \mathcal{T}[\![\texttt{nelist}(\tau_1'\mu_2, \tau_1'') \sqcap \tau_2'']\!]$ para algún $\pi' \equiv \pi^+$ (módulo $\setminus X''$), donde X'' es el conjunto de variables frescas generadas por $\texttt{nelist}(\tau_1'\mu_2, \tau_1'') \sqcap \tau_2''$. En adelante usaremos τ' para referirnos al tipo obtenido por dicho ínfimo. Para cada $i \in \{1..m\}$ vamos a definir la instanciación $\pi_i^\star$ como:

$$\forall \alpha \in \textbf{TypeVar}.\ \pi_i^\star(\alpha) = \begin{cases} \pi_i^\bullet(\alpha) & \text{si } \alpha \notin ftv(\tau_1'\mu_1 \sqcap \tau_2') \\ \pi'(\alpha) & \text{e.o.c.} \end{cases}$$

Resulta obvio que $\pi_i^\star \equiv \pi_i^\bullet$ (módulo $ftv(\tau_1'\mu_1 \sqcap \tau_2')$), para cada $i \in \{1..m\}$, y por lo tanto $(v_i, \pi_i^\star) \in \mathcal{T}[\![\tau_1'\mu_1 \sqcap \tau_2']\!]$. Sea una instanciación $\pi^\star = \bigcup_{i=1}^{m} \pi_i^\star$, como tenemos que se cumple que $ftv(\tau_1'\mu_1 \sqcap \tau_2') \cap ftv(\tau') = \emptyset$, obtendremos entonces que $\pi^\star \equiv \pi'$ (módulo $ftv(\tau')$) y por consiguiente $(v'', \pi^\star) \in \mathcal{T}[\![\tau']\!]$. Por lo tanto, $\pi^\star \in Dcp\left(\left\{\overline{\pi_i^\star}^n\right\}, \tau_1'\mu_1 \sqcap \tau_2'\right)$ y $(v, \pi^\star) \in \mathcal{T}[\![\texttt{nelist}(\tau_1'\mu_1 \sqcap \tau_2', \tau')]\!]$. Además, si $\alpha \notin X'$, $\alpha \notin X''$ y $\alpha \notin \left\{\overline{\delta_i^1}\right\} \cup \left\{\overline{\delta_i^2}\right\}$, tendremos que:

- Si $\alpha \in ftv(\tau_1'\mu_1 \sqcap \tau_2')$, obtendremos:
  1. $\pi^\star(\alpha) = \bigcup_{i=1}^{m} \pi_i^\star(\alpha) = \bigcup_{i=1}^{m} \pi_i^\bullet(\alpha) = \bigcup_{i=1}^{m} \pi_i^\circ(\alpha)$, porque $\alpha \notin X'$.
  2. $\bigcup_{i=1}^{m} \pi_i^\circ(\alpha) = \bigcup_{i=1}^{m} \pi_i'(\alpha) = \pi(\alpha)$, porque $\alpha \notin ftv(\tau_1'\mu_1) \subseteq \left\{\overline{\delta_i^1}\right\}$.
- Si $\alpha \in ftv(\tau') = ftv(\texttt{nelist}(\tau_1'\mu_2, \tau_1'') \sqcap \tau_2'')$, obtendremos:
  1. $\pi^\star(\alpha) = \pi'(\alpha) = \pi^+(\alpha)$, porque $\alpha \notin X''$.
  2. $\pi^+(\alpha) = \bigcup_{i=1}^{m} \pi_i^+(\alpha) = \bigcup_{i=1}^{m} \pi_i(\alpha) = \pi(\alpha)$, porque $\alpha \notin ftv(\tau_1'\mu_2) \subseteq \left\{\overline{\delta_i^2}\right\}$

Por lo tanto, $\pi^\star(\alpha) = \pi(\alpha)$ para las variables no frescas. Ahora tendremos que demostrar que:

$$\pi^\star \vDash \{\alpha \subseteq \mu_1(\alpha) \cup \mu_2(\alpha), \mu_1(\alpha) \subseteq \alpha, \mu_2(\alpha) \subseteq \alpha \mid \alpha \in ftv(\tau_1')\}$$

Sea una variable de tipo $\alpha \in ftv(\tau'_1)$, tendremos entonces que $\mu_1(\alpha) \in ftv(\tau'_1\mu_1)$ y $\mu_2(\alpha) \in ftv(\tau'_1\mu_2)$, por lo tanto $\pi^\star(\alpha) = \pi(\alpha)$, porque α no es fresca. Entonces:

$$
\begin{array}{ll}
 & \pi^\star(\mu_1(\alpha)) \cup \pi^\star(\mu_2(\alpha)) \\
= & \bigcup_{i=1}^{m} \pi_i^\star(\mu_1(\alpha)) \cup \bigcup_{i=1}^{m} \pi_i^\star(\mu_2(\alpha)) \\
= & \quad \{ \text{porque } \mu_1(\alpha) \in ftv(\tau'_1\mu_1 \sqcap \tau'_2) \text{ y } \mu_2(\alpha) \notin ftv(\tau'_1\mu_1 \sqcap \tau'_2) \} \\
 & \bigcup_{i=1}^{m} \pi_i^\bullet(\mu_1(\alpha)) \cup \bigcup_{i=1}^{m} \pi'_i(\mu_2(\alpha)) \\
= & \bigcup_{i=1}^{m} \pi_i^\bullet(\mu_1(\alpha)) \cup \pi'(\mu_2(\alpha)) \\
= & \quad \{ \text{porque } \mu_2(\alpha) \notin X'' \} \\
 & \bigcup_{i=1}^{m} \pi_i^\bullet(\mu_1(\alpha)) \cup \pi^+(\mu_2(\alpha)) \\
= & \bigcup_{i=1}^{m} \pi_i^\bullet(\mu_1(\alpha)) \cup \bigcup_{i=m+1}^{n} \pi_i^+(\mu_2(\alpha)) \\
= & \quad \{ \text{porque } \mu_2(\alpha) \in ftv(\tau'_1\mu_2) \} \\
 & \bigcup_{i=1}^{m} \pi_i^\bullet(\mu_1(\alpha)) \cup \bigcup_{i=m+1}^{n} (\pi_i\mu_2)(\mu_2(\alpha)) \\
= & \quad \{ \text{porque } \forall i \in \{m+1..n\} : (\pi_i\mu_2)(\mu_2(\alpha)) = \pi_1(\mu_2^{-1}(\mu_2(\alpha))) = \pi_i(\alpha) \} \\
 & \bigcup_{i=1}^{m} \pi_i^\bullet(\mu_1(\alpha)) \cup \bigcup_{i=m+1}^{n} \pi_i(\alpha) \\
= & \quad \{ \text{porque } \mu_1(\alpha) \notin X' \} \\
 & \bigcup_{i=1}^{m} \pi_i^\circ(\mu_1(\alpha)) \cup \bigcup_{i=m+1}^{n} \pi_i(\alpha) \\
= & \quad \{ \text{porque } \mu_1(\alpha) \in ftv(\tau'_1\mu_1) \} \\
 & \bigcup_{i=1}^{m} (\pi_i\mu_1)(\mu_1(\alpha)) \cup \bigcup_{i=m+1}^{n} \pi_i(\alpha) \\
= & \quad \{ \text{porque } \forall i \in \{1..m\} : (\pi_i\mu_1)(\mu_1(\alpha)) = \pi_i(\mu_1^{-1}(\mu_1(\alpha))) = \pi_i(\alpha) \} \\
 & \bigcup_{i=1}^{m} \pi_i(\alpha) \cup \bigcup_{i=m+1}^{n} \pi_i(\alpha) \\
= & \bigcup_{i=1}^{n} \pi_i(\alpha) = \pi(\alpha)
\end{array}
$$

Por lo tanto, se cumple que $\pi^\star(\alpha) = \pi(\alpha) = \pi^\star(\mu_1(\alpha)) \cup \pi^\star(\mu_2(\alpha))$ y por consiguiente $\pi^\star \vDash \alpha \subseteq \mu_1(\alpha) \cup \mu_2(\alpha)$, $\pi^\star \vDash \mu_1(\alpha) \subseteq \alpha$ y $\pi^\star \vDash \mu_2(\alpha) \subseteq \alpha$, para cada $\alpha \in ftv(\tau'_1)$, obteniendo finalmente que

$$
\begin{array}{c}
(v, \pi^\star) \in \mathscr{T}[\![\texttt{nelist}(\tau'_1\mu_1 \sqcap \tau'_2, \texttt{nelist}(\tau'_1\mu_2, \tau''_1) \sqcap \tau''_2) \textbf{ when } C]\!] \\
\text{donde } C = \{\alpha \subseteq \mu_1(\alpha) \cup \mu_2(\alpha), \mu_1(\alpha) \subseteq \alpha, \mu_2(\alpha) \subseteq \alpha \mid \alpha \in ftv(\tau'_1)\}
\end{array}
$$

y también que $\pi^\star \equiv \pi$ (módulo $\setminus (X' \cup X'')$).

- **Caso** $n < m$

En este caso la forma de v es la siguiente $v = [v_1, \ldots, v_n \mid v']$ y $v = [v'_1, \ldots, v'_n, \ldots, v'_m \mid v'']$. Entonces $v_i = v'_i$, para cada $i \in \{1..n\}$, y $v' = [v'_{n+1}, \ldots, v'_m \mid v'']$. Suponemos una sustituciones de variables de tipo $\mu_3 = \left[\overline{\alpha_i / \delta_i^3}\right]$ y $\mu_4 = \left[\overline{\alpha_i / \delta_i^4}\right]$, donde $\{\overline{\alpha_i}\} = ftv(\tau'_2)$ y $\overline{\delta_i^3}, \overline{\delta_i^4}$ son variables de tipo frescas. El resto de la demostración es análoga al caso donde $n > m$, intercambiando el orden de los tipos y ocupando μ_3 y μ_4 el lugar de μ_1 y μ_2 respectivamente.

Para obtener finalmente que

$$(\nu,\pi^\star) \in \mathcal{T}\,[\![\texttt{nelist}(\tau'_1 \sqcap \tau'_2\mu_3\,,\tau''_1 \sqcap \texttt{nelist}(\tau'_2\mu_4\,,\tau''_2))\ \textbf{when}\ C]\!]$$
$$\text{donde } C = \{\alpha \subseteq \mu_3(\alpha) \cup \mu_4(\alpha), \mu_3(\alpha) \subseteq \alpha, \mu_4(\alpha) \subseteq \alpha \mid \alpha \in ftv(\tau'_2)\}$$

y también que $\pi^\star \equiv \pi$ (módulo $\setminus (X' \cup X'')$), donde X' es el conjunto de variables frescas generadas por $\tau'_1 \sqcap \tau'_2\mu_3$ y X'' el de las generadas por $\tau''_1 \sqcap \texttt{nelist}(\tau'_2\mu_4\,,\tau''_2)$.

Finalmente se demuestra la propiedad para el caso de las listas no vacías al unir los tres casos anteriores.

- **Caso** $\tau_1 = \left(\overline{\tau'_i}\right) \to \tau'$ **y** $\tau_2 = \left(\overline{\tau''_i}\right) \to \tau''$

  Vamos a demostrar el caso más sencillo, en el que $\tau_1 = (\tau'_1) \to \tau'_2$ y $\tau_2 = (\tau''_1) \to \tau''_2$, porque su demostración se puede generalizar a n parámetros sobre los resultados obtenidos con los tipos τ'_1 y τ''_1. Entonces, suponemos un par (ν,π), tal que $(\nu,\pi) \in \mathcal{T}\,[\![(\tau'_1) \to \tau'_2]\!]$ y $(\nu,\pi) \in \mathcal{T}\,[\![(\tau''_1) \to \tau''_2]\!]$. Denominamos X' y X'' los conjuntos de variables frescas generadas por $\tau'_1 \sqcap \tau''_1$ y $\tau'_2 \sqcap \tau''_2$ respectivamente.

  - **Caso** $\nu = \emptyset$

    Si estamos ante la función de grafo vacío, entonces tenemos que $\pi(\alpha) = \emptyset$, para toda variable de tipo $\alpha \in itv((\tau'_1) \to \tau'_2) \cup itv((\tau''_1) \to \tau''_2)$. Definimos la instanciación π^+ del siguiente modo:

    $$\forall \alpha \in \textbf{TypeVar}.\ \pi^+(\alpha) = \begin{cases} \emptyset & \text{si } \alpha \in X' \cup X'' \\ \pi(\alpha) & \text{e.o.c.} \end{cases}$$

    Sea una variable de tipo $\alpha \in itv((\tau'_1 \sqcap \tau''_1) \to \tau'_2 \sqcap \tau''_2)$, tenemos que:

    - Si $\alpha \in itv(\tau'_1 \sqcap \tau''_1)$, entonces $\alpha \in itv(\tau'_1) \cup itv(\tau''_1) \cup X'$, por lo que:
      - Si $\alpha \in itv(\tau'_1) \cup itv(\tau''_1)$, entonces $\pi^+(\alpha) = \pi(\alpha) = \emptyset$.
      - Si $\alpha \in X'$, entonces $\pi^+(\alpha) = \emptyset$.
    - Si $\alpha \in itv(\tau'_2 \sqcap \tau''_2)$, entonces $\alpha \in itv(\tau'_2) \cup itv(\tau''_2) \cup X''$, por lo que:
      - Si $\alpha \in itv(\tau'_2) \cup itv(\tau''_2)$, entonces $\pi^+(\alpha) = \pi(\alpha) = \emptyset$.
      - Si $\alpha \in X''$, entonces $\pi^+(\alpha) = \emptyset$.

    Por lo tanto, se cumple que $\pi^+(\alpha) = \emptyset$, para todo $\alpha \in itv((\tau'_1 \sqcap \tau''_1) \to \tau'_2 \sqcap \tau''_2)$, obteniendo así que $(\emptyset,\pi^+) \in \mathcal{T}\,[\![(\tau'_1 \sqcap \tau''_1) \to \tau'_2 \sqcap \tau''_2]\!]$ y además $\pi^+ \equiv \pi$ (módulo $\setminus (X' \cup X'')$), por definición de π^+.

  - **Caso** $\nu \neq \emptyset$

    Suponemos que existen los conjuntos de instanciaciones $\{\pi'_w \mid w \in \nu\}$ y $\{\pi''_w \mid w \in \nu\}$, tales que $\pi \in Dcp(\{\pi'_w \mid w \in \nu\}, (\tau'_1) \to \tau'_2)$ y $\pi \in Dcp(\{\pi''_w \mid w \in \nu\}, (\tau''_1) \to \tau''_2)$. Además, para cada $w \in \nu$, si w tiene la forma $((\nu'), \nu'')$, tenemos que:

    $$(\nu',\pi'_w) \in \mathcal{T}\,[\![\tau'_1]\!] \quad \wedge \quad (\nu'',\pi'_w) \in \mathcal{T}\,[\![\tau'_2]\!] \quad \wedge \quad (\nu',\pi''_w) \in \mathcal{T}\,[\![\tau''_1]\!] \quad \wedge \quad (\nu'',\pi''_w) \in \mathcal{T}\,[\![\tau''_2]\!]$$

Entonces definimos, para todo $w \in v$, la instanciación π°_w del siguiente modo:

$$\forall \alpha \in \mathbf{TypeVar}.\, \pi^{\circ}_w(\alpha) = \begin{cases} \pi'_w(\alpha) & \text{si } \alpha \in ftv\big((\tau'_1) \to \tau'_2\big) \\ \pi''_w(\alpha) & \text{e.o.c.} \end{cases}$$

Resulta obvio que $\pi^{\circ}_w \equiv \pi'_w(\alpha)$ (módulo $ftv\big((\tau'_1) \to \tau'_2\big)$). Además, por ser $ftv\big((\tau'_1) \to \tau'_2\big) \cap ftv\big((\tau''_1) \to \tau''_2\big) = \emptyset$, obtendremos que $\pi^{\circ}_w \equiv \pi''_w(\alpha)$ (módulo $ftv\big((\tau''_1) \to \tau''_2\big)$). Por lo tanto, $(v', \pi^{\circ}_w) \in \mathscr{T}\,[\![\tau'_1]\!]$ y $(v', \pi^{\circ}_w) \in \mathscr{T}\,[\![\tau''_1]\!]$, y por hipótesis de inducción tenemos que $(v', \pi^{\bullet}_w) \in \mathscr{T}\,[\![\tau'_1 \sqcap \tau''_1]\!]$, para algún $\pi^{\bullet}_w$ tal que $\pi^{\bullet}_w \equiv \pi^{\circ}_w$ (módulo $\setminus X'$). Luego con $(v'', \pi^{\circ}_w) \in \mathscr{T}\,[\![\tau'_2]\!]$ y $(v'', \pi^{\circ}_w) \in \mathscr{T}\,[\![\tau''_2]\!]$, por hipótesis de inducción tenemos que $(v'', \pi^{\bullet\bullet}_w) \in \mathscr{T}\,[\![\tau'_2 \sqcap \tau''_2]\!]$, para algún $\pi^{\bullet\bullet}_w$ tal que $\pi^{\bullet\bullet}_w \equiv \pi^{\circ}_w$ (módulo $\setminus X''$). Así, para todo $w \in v$, definimos la instanciación π^{+}_w del siguiente modo:

$$\forall \alpha \in \mathbf{TypeVar}.\, \pi^{+}_w(\alpha) = \begin{cases} \pi^{\bullet}_w(\alpha) & \text{si } \alpha \in X' \\ \pi^{\bullet\bullet}_w(\alpha) & \text{si } \alpha \in X'' \\ \pi^{\circ}_w(\alpha) & \text{e.o.c.} \end{cases}$$

Esta instanciación está bien definida porque $X' \cap X'' = \emptyset$. Ahora, sea una variable de tipo $\alpha \in ftv\big(\tau'_1 \sqcap \tau''_1\big)$, entonces $\alpha \in ftv\big(\tau'_1\big) \cup ftv\big(\tau''_1\big) \cup X'$. Por lo tanto:

- Si $\alpha \in X'$, entonces $\pi^{+}_w(\alpha) = \pi^{\bullet}_w(\alpha)$.
- Si $\alpha \notin X'$, entonces $\pi^{+}_w(\alpha) = \pi^{\circ}_w(\alpha) = \pi^{\bullet}_w(\alpha)$, porque $\pi^{\circ}_w \equiv \pi^{\bullet}_w$ (módulo $\setminus X'$).

Por lo tanto $\pi^{+}_w \equiv \pi^{\bullet}_w$ (módulo $ftv\big(\tau'_1 \sqcap \tau''_1\big)$). Ahora, sea una variable de tipo $\alpha \in ftv\big(\tau'_2 \sqcap \tau''_2\big)$, entonces $\alpha \in ftv\big(\tau'_2\big) \cup ftv\big(\tau''_2\big) \cup X''$. Por lo tanto:

- Si $\alpha \in X''$, entonces $\pi^{+}_w(\alpha) = \pi^{\bullet\bullet}_w(\alpha)$.
- Si $\alpha \notin X''$, entonces $\pi^{+}_w(\alpha) = \pi^{\circ}_w(\alpha) = \pi^{\bullet\bullet}_w(\alpha)$, porque $\pi^{\circ}_w \equiv \pi^{\bullet\bullet}_w$ (módulo $\setminus X''$).

Por lo tanto $\pi^{+}_w \equiv \pi^{\bullet\bullet}_w$ (módulo $ftv\big(\tau'_2 \sqcap \tau''_2\big)$). Con esto obtenemos que $(v', \pi^{+}_w) \in \mathscr{T}\,[\![\tau'_1 \sqcap \tau''_1]\!]$ y $(v'', \pi^{+}_w) \in \mathscr{T}\,[\![\tau'_2 \sqcap \tau''_2]\!]$. Todo lo anterior se cumple para todo $w \in v$. A continuación definimos la instanciación $\pi^{+} = \bigcup_{w \in v} \pi^{+}_w$. Ahora vamos a demostrar que se cumple que:

$$\pi^{+} \in Dcp\big(\{\pi^{+}_w \mid w \in v\}, \big(\tau'_1 \sqcap \tau''_1\big) \to \tau'_2 \sqcap \tau''_2\big)$$

Sea una variable de tipo $\alpha \notin itv\big((\tau'_1 \sqcap \tau''_1) \to \tau'_2 \sqcap \tau''_2\big)$, pero $\alpha \in ftv\big((\tau'_1 \sqcap \tau''_1) \to \tau'_2 \sqcap \tau''_2\big)$. Para toda $w \in v$, tenemos que:

- Si $\alpha \in ftv\big(\tau'_1\big)$, entonces se cumple que $\pi^{+}_w(\alpha) = \pi^{\bullet}_w(\alpha) = \pi^{\circ}_w(\alpha) = \pi'_w(\alpha) = \pi(\alpha)$ y $\pi^{+}(\alpha) = \bigcup_{w \in v} \pi^{+}_w(\alpha) = \bigcup_{w \in v} \pi(\alpha) = \pi(\alpha)$.
- Si $\alpha \in ftv\big(\tau''_1\big)$, entonces se cumple que $\pi^{+}_w(\alpha) = \pi^{\bullet}_w(\alpha) = \pi^{\circ}_w(\alpha) = \pi''_w(\alpha) = \pi(\alpha)$ y $\pi^{+}(\alpha) = \bigcup_{w \in v} \pi^{+}_w(\alpha) = \bigcup_{w \in v} \pi(\alpha) = \pi(\alpha)$.
- Si $\alpha \in X'$, entonces existe una variable de tipo $\beta \in ftv\big(\tau'_1\big) \cup ftv\big(\tau''_1\big)$ con $\pi^{+}_w(\beta) = \pi^{+}_w(\alpha)$

y $\beta \notin itv(\tau_1) \cup itv(\tau_2)$, por lo que se cumple que $\pi^+_w(\alpha) = \pi^+_w(\beta) = \pi(\beta) = \bigcup_{w \in v} \pi(\beta) = \bigcup_{w \in v} \pi^+_w(\beta) = \pi^+(\alpha)$.

- Si $\alpha \in ftv(\tau'_2)$, entonces se cumple que $\pi^+_w(\alpha) = \pi^{\bullet\bullet}_w(\alpha) = \pi^\circ_w(\alpha) = \pi'_w(\alpha) = \pi(\alpha)$ y $\pi^+(\alpha) = \bigcup_{w \in v} \pi^+_w(\alpha) = \bigcup_{w \in v} \pi(\alpha) = \pi(\alpha)$.
- Si $\alpha \in ftv(\tau''_2)$, entonces se cumple que $\pi^+_w(\alpha) = \pi^{\bullet\bullet}_w(\alpha) = \pi^\circ_w(\alpha) = \pi''_w(\alpha) = \pi(\alpha)$ y $\pi^+(\alpha) = \bigcup_{w \in v} \pi^+_w(\alpha) = \bigcup_{w \in v} \pi(\alpha) = \pi(\alpha)$.
- Si $\alpha \in X''$, entonces existe una variable de tipo $\beta \in ftv(\tau'_2) \cup ftv(\tau''_2)$ con $\pi^+_w(\beta) = \pi^+_w(\alpha)$ y $\beta \notin itv(\tau_1) \cup itv(\tau_2)$, por lo que se cumple que $\pi^+_w(\alpha) = \pi^+_w(\beta) = \pi(\beta) = \bigcup_{w \in v} \pi(\beta) = \bigcup_{w \in v} \pi^+_w(\beta) = \pi^+(\alpha)$.

Por lo tanto $\pi^+_w(\alpha) = \pi^+(\alpha)$, cumpliéndose que $\pi^+ \in Dcp(\{\pi^+_w \mid w \in v\}, (\tau'_1 \sqcap \tau''_1) \to \tau'_2 \sqcap \tau''_2)$ y $(v, \pi^+) \in \mathcal{T}[\![(\tau'_1 \sqcap \tau''_1) \to \tau'_2 \sqcap \tau''_2]\!]$. Ahora solo queda demostrar que se cumple que $\pi^+ \equiv \pi$ (módulo $\setminus (X' \cup X'')$). Sea una variable de tipo $\alpha \in X' \cup X''$. Tendremos que se cumple que $\pi^+(\alpha) = \bigcup_{w \in v} \pi^+_w(\alpha) = \bigcup_{w \in v} \pi^\circ_w(\alpha)$. Entonces:

- Si $\alpha \in ftv((\tau'_1) \to \tau'_2)$, entonces $\bigcup_{w \in v} \pi^\circ_w(\alpha) = \bigcup_{w \in v} \pi'_w(\alpha) = \pi(\alpha)$.
- Si $\alpha \notin ftv((\tau'_1) \to \tau'_2)$, entonces $\bigcup_{w \in v} \pi^\circ_w(\alpha) = \bigcup_{w \in v} \pi''_w(\alpha) = \pi(\alpha)$.

Demostrando así la propiedad al unir los resultados obtenidos para ambos casos.

- **Caso** $\tau_1 = \forall \overline{\alpha_i}.\tau'_1$ **y** $\tau_2 = \forall \overline{\beta_j}.\tau'_2$

Suponemos un par (v, π), tal que $(v, \pi) \in \mathcal{T}[\![\forall \overline{\alpha_i}.\tau'_1]\!]$ y $(v, \pi) \in \mathcal{T}[\![\forall \overline{\beta_j}.\tau'_2]\!]$, tales que $\{\overline{\alpha_i}\} \cap \{\overline{\beta_j}\} = \emptyset$, $\{\overline{\alpha_i}\} \cap ftv(\tau_2) = \emptyset$ y $\{\overline{\beta_j}\} \cap ftv(\tau_1) = \emptyset$. Existe una instanciación π', tal que $(v, \pi') \in \mathcal{T}[\![\tau'_1]\!]$ y $\pi' \equiv \pi$ (módulo $\setminus \{\overline{\alpha_i}\}$). Existe una instanciación π'', tal que $(v, \pi'') \in \mathcal{T}[\![\tau'_2]\!]$ y $\pi'' \equiv \pi$ (módulo $\setminus \{\overline{\beta_j}\}$). Definimos la instanciación π° del siguiente modo:

$$\forall \alpha \in \textbf{TypeVar}.\ \pi^\circ(\alpha) = \begin{cases} \pi'(\alpha) & \text{si } \alpha \in \{\overline{\alpha_i}\} \\ \pi''(\alpha) & \text{si } \alpha \in \{\overline{\beta_j}\} \\ \pi(\alpha) & \text{e.o.c.} \end{cases}$$

Sea una variable de tipo $\alpha \in ftv(\tau'_1)$, entonces $\alpha \notin \{\overline{\beta_j}\}$, por lo que:

- Si $\alpha \in \{\overline{\alpha_i}\}$, entonces $\pi^\circ(\alpha) = \pi'(\alpha)$.
- Si $\alpha \notin \{\overline{\alpha_i}\}$, entonces $\pi^\circ(\alpha) = \pi(\alpha) = \pi'(\alpha)$.

Por lo tanto $\pi^\circ \equiv \pi'$ (módulo $ftv(\tau'_1)$). Ahora sea una variable de tipo $\alpha \in ftv(\tau'_2)$, entonces $\alpha \notin \{\overline{\alpha_i}\}$, por lo que:

- Si $\alpha \in \{\overline{\beta_j}\}$, entonces $\pi^\circ(\alpha) = \pi''(\alpha)$.
- Si $\alpha \notin \{\overline{\beta_j}\}$, entonces $\pi^\circ(\alpha) = \pi(\alpha) = \pi''(\alpha)$.

Por lo tanto $\pi^\circ \equiv \pi''$ (módulo $ftv(\tau'_2)$). Como consecuencia de esto último, tenemos que $(v,\pi^\circ) \in \mathcal{T}[\![\tau'_1]\!]$ y $(v,\pi^\circ) \in \mathcal{T}[\![\tau'_2]\!]$, y por hipótesis de inducción obtendremos que $(v,\pi^\bullet) \in \mathcal{T}[\![\tau'_1 \sqcap \tau'_2]\!]$ y $\pi^\bullet \equiv \pi^\circ$ (módulo $\setminus X'$), donde X' es el conjunto de variables frescas generadas por $\tau'_1 \sqcap \tau'_2$. Ahora definimos la instanciación $\pi^\star$ del siguiente modo:

$$\forall \alpha \in \mathbf{TypeVar}.\ \pi^\star(\alpha) = \begin{cases} \pi(\alpha) & \text{si } \alpha \in \{\overline{\alpha_i}\} \cup \left\{\overline{\beta_j}\right\} \\ \pi^\bullet(\alpha) & \text{e.o.c.} \end{cases}$$

Se cumple de un modo obvio que $\pi^\star \equiv \pi^\bullet$ (módulo $\setminus \left(\{\overline{\alpha_i}\} \cup \left\{\overline{\beta_j}\right\}\right)$). Por lo tanto, tendremos que $(v,\pi^\star) \in \mathcal{T}\left[\!\left[\forall \overline{\alpha_i}, \overline{\beta_j}.\tau'_1 \sqcap \tau'_2\right]\!\right]$. Ahora sea una variable de tipo $\alpha \notin X'$:

- Si $\alpha \in \left(\{\overline{\alpha_i}\} \cup \left\{\overline{\beta_j}\right\}\right)$, entonces $\pi^\star(\alpha) = \pi(\alpha)$.
- Si $\alpha \notin \left(\{\overline{\alpha_i}\} \cup \left\{\overline{\beta_j}\right\}\right)$, entonces $\pi^\star(\alpha) = \pi^\bullet(\alpha) = \pi^\circ(\alpha) = \pi(\alpha)$.

Por lo tanto, $\pi^\star \equiv \pi$ (módulo $\setminus X'$).

- **Caso** $\tau_1 = \bigsqcup_{i=1}^n \sigma_i$ **y** $\tau_2 = \bigsqcup_{j=1}^m \sigma'_j$

  Llamamos $X_{i,j}$ al conjunto de variables frescas generadas por $\sigma_i \sqcap \sigma'_j$. Suponemos un par (v,π), tal que $(v,\pi) \in \mathcal{T}[\![\bigsqcup_{i=1}^n \sigma_i]\!]$ y $(v,\pi) \in \mathcal{T}\left[\!\left[\bigsqcup_{j=1}^m \sigma'_j\right]\!\right]$. Suponemos la existencia de un $i \in \{1..n\}$ y un $j \in \{1..m\}$, tales que $(v,\pi) \in \mathcal{T}[\![\sigma_i]\!]$ y $(v,\pi) \in \mathcal{T}\left[\!\left[\sigma'_j\right]\!\right]$, y por hipótesis de inducción obtendremos que $(v,\pi') \in \mathcal{T}\left[\!\left[\sigma_i \sqcap \sigma'_j\right]\!\right]$, para algún $\pi' \equiv \pi$ (módulo $\setminus X_{i,j}$). Por lo tanto $(v,\pi') \in \mathcal{T}\left[\!\left[\bigsqcup_{\substack{i\in\{1..n\}\\ j\in\{1..m\}}} \left(\sigma_i \sqcap \sigma'_j\right)\right]\!\right]$. Además, para toda variable de tipo $\alpha \notin \bigcup_{i=1}^n \bigcup_{j=1}^m X_{i,j}$, tenemos que $\pi'(\alpha) = \pi(\alpha)$ porque $\alpha \notin X_{i,j}$. Por lo tanto $\pi' \equiv \pi$ (módulo $\setminus \bigcup_{i=1}^n \bigcup_{j=1}^m X_{i,j}$).

- **Caso** $\tau_2 = \bigcup_{i=1}^n \tau'_i$

  Suponemos un tipo $\bigcup_{i=1}^n \tau'_i$ que está $\cup$-normalizado. Llamamos X'_i al conjunto de variables frescas generadas por $\tau_1 \sqcap \tau'_i$. Suponemos un par (v,π), tal que $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$ y $(v,\pi) \in \mathcal{T}[\![\bigcup_{i=1}^n \tau'_i]\!]$. Suponemos la existencia de un $i \in \{1..n\}$, tal que $(v,\pi) \in \mathcal{T}[\![\tau'_i]\!]$, que junto a $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$, por hipótesis de inducción obtendremos que $(v,\pi') \in \mathcal{T}[\![\tau_1 \sqcap \tau'_i]\!]$, para algún $\pi' \equiv \pi$ (módulo $\setminus X'_i$). Por lo tanto $(v,\pi') \in \mathcal{T}[\![\bigcup_{i=1}^n (\tau_1 \sqcap \tau'_i)]\!]$ y $\pi' \equiv \pi$ (módulo $\setminus \bigcup_{i=1}^n X'_i$).

- **Caso restante**

  Para cualquier τ_2 hay una regla cuyo lado derecho del ínfimo encaja con τ_2:

  - Si τ_2 es `none()`, `any()`, α, τ **when** C o $\bigcup_{i=1}^n \tau_i$, usaremos la regla correspondiente para cualquier τ_1.
  - Si $\tau_2 = c$, entonces $(c,\pi) \in \mathcal{T}[\![c]\!]$. Suponemos que $(c,\pi) \in \mathcal{T}[\![\tau_1]\!]$, entonces:
    - Si $\tau_1 =$ `any()`, se aplica $c \sqcap$ `any()`.
    - Si $\tau_1 = \alpha$, se aplica $c \sqcap \alpha$.

  - Si $\tau_1 = \tau$ **when** C, se aplica $c \sqcap \tau$ **when** C.
  - Si $\tau_1 = \bigcup_{i=1}^{n} \tau'_i$, se aplica $c \sqcap \bigcup_{i=1}^{n} \tau'_i$.
  - Si $\tau_1 = c$, se aplica $c \sqcap c$.
  - Si $\tau_1 = B$, se aplica $c \sqcap B$.
- Si $\tau_2 = B$, entonces $(v,\pi) \in \mathcal{T}[\![B]\!]$. Suponemos que $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$, entonces:
  - Si $\tau_1 = \texttt{any()}$, se aplica $B \sqcap \texttt{any()}$.
  - Si $\tau_1 = \alpha$, se aplica $B \sqcap \alpha$.
  - Si $\tau_1 = \tau$ **when** C, se aplica $B \sqcap \tau$ **when** C.
  - Si $\tau_1 = \bigcup_{i=1}^{n} \tau'_i$, se aplica $B \sqcap \bigcup_{i=1}^{n} \tau'_i$.
  - Si $\tau_1 = B'$, se aplica $B' \sqcap B$.
- Si $\tau_2 = \{\overline{\tau'_i}^{\,n}\}$, entonces $(v,\pi) \in \mathcal{T}\left[\!\left[\{\overline{\tau'_i}^{\,n}\}\right]\!\right]$, por lo que $v = \left(\{\overset{n}{\cdots}\}, v_1, \ldots, v_n\right)$, para algunos $v_1, \ldots, v_n$. Suponemos que $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$, entonces:
  - Si $\tau_1 = \texttt{any()}$, se aplica $\{\tau'_1, \ldots, \tau'_n\} \sqcap \texttt{any()}$.
  - Si $\tau_1 = \alpha$, se aplica $\{\tau'_1, \ldots, \tau'_n\} \sqcap \alpha$.
  - Si $\tau_1 = \tau$ **when** C, se aplica $\{\tau'_1, \ldots, \tau'_n\} \sqcap \tau$ **when** C.
  - Si $\tau_1 = \bigcup_{i=1}^{n} \tau''_i$, se aplica $\{\tau'_1, \ldots, \tau'_n\} \sqcap \bigcup_{i=1}^{n} \tau''_i$.
  - Si $\tau_1 = \{\tau''_1, \ldots, \tau''_n\}$, se aplica $\{\tau''_1, \ldots, \tau''_n\} \sqcap \{\tau'_1, \ldots, \tau'_n\}$.
- Si $\tau_2 = \texttt{nelist(}\tau'_1\texttt{,}\tau'_2\texttt{)}$, entonces $(v,\pi) \in \mathcal{T}[\![\texttt{nelist(}\tau'_1\texttt{,}\tau'_2\texttt{)}]\!]$, por lo que, para algunos $v_1, \ldots, v_n, v'$, tenemos que $v = [v_1, \ldots, v_n \mid v']$. Suponemos que $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$, entonces:
  - Si $\tau_1 = \texttt{any()}$, se aplica $\texttt{nelist(}\tau'_1\texttt{,}\tau'_2\texttt{)} \sqcap \texttt{any()}$.
  - Si $\tau_1 = \alpha$, se aplica $\texttt{nelist(}\tau'_1\texttt{,}\tau'_2\texttt{)} \sqcap \alpha$.
  - Si $\tau_1 = \tau$ **when** C, se aplica $\texttt{nelist(}\tau'_1\texttt{,}\tau'_2\texttt{)} \sqcap \tau$ **when** C.
  - Si $\tau_1 = \bigcup_{i=1}^{n} \tau''_i$, se aplica $\texttt{nelist(}\tau'_1\texttt{,}\tau'_2\texttt{)} \sqcap \bigcup_{i=1}^{n} \tau''_i$.
  - Si $\tau_1 = \texttt{nelist(}\tau''_1\texttt{,}\tau''_2\texttt{)}$, se aplica $\texttt{nelist(}\tau''_1\texttt{,}\tau''_2\texttt{)} \sqcap \texttt{nelist(}\tau'_1\texttt{,}\tau'_2\texttt{)}$.
- Si $\tau_2 = \left(\overline{\tau'_i}\right) \to \tau'$, entonces $(v,\pi) \in \mathcal{T}\left[\!\left[\left(\overline{\tau'_i}\right) \to \tau'\right]\!\right]$. Suponemos que $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$, entonces:
  - Si $\tau_1 = \texttt{any()}$, se aplica $\left(\overline{\tau'_i}\right) \to \tau' \sqcap \texttt{any()}$.
  - Si $\tau_1 = \alpha$, se aplica $\left(\overline{\tau'_i}\right) \to \tau' \sqcap \alpha$.
  - Si $\tau_1 = \tau$ **when** C, se aplica $\left(\overline{\tau'_i}\right) \to \tau' \sqcap \tau$ **when** C.
  - Si $\tau_1 = \bigcup_{i=1}^{n} \tau'_i$, se aplica $\left(\overline{\tau'_i}\right) \to \tau' \sqcap \bigcup_{i=1}^{n} \tau'_i$.
  - Si $\tau_1 = \left(\overline{\tau''_i}\right) \to \tau''$, se aplica $\left(\overline{\tau''_i}\right) \to \tau'' \sqcap \left(\overline{\tau'_i}\right) \to \tau'$.
- Si $\tau_2 = \forall \overline{\beta_i}.\tau'_2$, entonces $(v,\pi) \in \mathcal{T}\left[\!\left[\forall \overline{\beta_i}.\tau'_2\right]\!\right]$. Suponemos que $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$, entonces:
  - Si $\tau_1 = \texttt{any()}$, se aplica $\forall \overline{\beta_i}.\tau'_2 \sqcap \texttt{any()}$.
  - Si $\tau_1 = \alpha$, se aplica $\forall \overline{\beta_i}.\tau'_2 \sqcap \alpha$.

- Si $\tau_1 = \tau$ **when** C, se aplica $\forall\overline{\beta_i}.\tau_2' \sqcap \tau$ **when** C.
- Si $\tau_1 = \bigcup_{i=1}^{n} \tau_i'$, se aplica $\forall\overline{\beta_i}.\tau_2' \sqcap \bigcup_{i=1}^{n} \tau_i'$.
- Si $\tau_1 = \forall\overline{\alpha_i}.\tau_1'$, se aplica $\forall\overline{\alpha_i}.\tau_1' \sqcap \forall\overline{\beta_i}.\tau_2'$.

- Si $\tau_2 = \bigsqcup_{i=1}^{n} \sigma_i$, entonces $(v,\pi) \in \mathcal{T}\left[\!\left[\bigsqcup_{i=1}^{n} \sigma_i\right]\!\right]$. Suponemos que $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$, entonces:
  - Si $\tau_1 = \mathsf{any}()$, se aplica $\bigsqcup_{i=1}^{n} \sigma_i \sqcap \mathsf{any}()$.
  - Si $\tau_1 = \alpha$, se aplica $\bigsqcup_{i=1}^{n} \sigma_i \sqcap \alpha$.
  - Si $\tau_1 = \tau$ **when** C, se aplica $\bigsqcup_{i=1}^{n} \sigma_i \sqcap \tau$ **when** C.
  - Si $\tau_1 = \bigcup_{i=1}^{n} \tau_i'$, se aplica $\bigsqcup_{i=1}^{n} \sigma_i \sqcap \bigcup_{i=1}^{n} \tau_i'$.
  - Si $\tau_1 = \bigsqcup_{j=1}^{m} \sigma_j'$, se aplica $\bigsqcup_{j=1}^{m} \sigma_j' \sqcap \bigsqcup_{i=1}^{n} \sigma_i$.

  Cualquiera de estos casos contradice al hecho de que estemos en el caso final del ínfimo. Por lo tanto, en el caso final del ínfimo, suponer que tenemos $(v,\pi) \in \mathcal{T}[\![\tau_1]\!]$ y $(v,\pi) \in \mathcal{T}[\![\tau_2]\!]$ lleva a una contradicción. Por lo que el teorema se cumple trivialmente para este caso particular.

□

5.3.4. Relación de subtipado entre tipos polimórficos

Durante el proceso de inferencia de tipos en funciones recursivas (que se describirá en la sección 5.6), es necesario realizar un proceso iterativo en el que se va obteniendo una cadena ascendente de tipos candidato, en la que cada tipo es un supertipo del anterior en la cadena. El proceso termina en el momento en el que un elemento de la cadena sea también subtipo del anterior (y, por tanto, sea semánticamente equivalente). Por tanto, necesitamos saber de manera efectiva si, dados dos tipos, uno es subtipo de otro.

La relación de subtipado que hemos introducido en capítulos anteriores es una noción puramente semántica, y no es fácil extraer, a partir de ella, un algoritmo que determine si un tipo es subtipo de otro. No obstante, para nuestros propósitos, no es necesario caracterizar con precisión absoluta la relación de subtipado, sino que basta una aproximación inferior de dicha relación. Para ello, dados dos esquemas de tipos funcionales τ y τ', tales que $vars(\tau) \cap vars(\tau') = \emptyset$, usaremos la notación $\tau \overset{\circ}{\subseteq} \tau'$ para representar una aproximación a la relación de subconjunto entre tipos. En caso de haber colisión entre las variables de τ y las de τ', se puede renombrar uno de los dos tipos para hacer la operación.

Nuestro algoritmo de inferencia ha sido diseñado para ser paramétrico en relación a diferentes operaciones entre tipos, entre ellos la relación de subtipado entre tipos polimórficos. De este modo podemos tener la corrección del algoritmo en sí por un lado y la de las operaciones por otro. En esta tesis supondremos que disponemos de un operador computable $\overset{\circ}{\subseteq}$ que cumple la siguiente propiedad:

Proposición 31. *Suponiendo un par de tipos polimórficos τ_1 y τ_2, tales que $ftv(\tau_1) \cap ftv(\tau_2) = \emptyset$. Si $\tau_1 \overset{\circ}{\subseteq} \tau_2$, entonces $\mathcal{T}[\![\tau_1]\!] \subseteq \mathcal{T}[\![\tau_2]\!]$.*

En el apéndice B.2 se describe una propuesta de algoritmo que permite aproximar la relación de subtipado entre tipos polimórficos, planteándose su demostración de corrección como trabajo futuro.

5.4. Reglas de normalización y simplificación de tipos

En la figura 5.1 presentamos la sintaxis normalizada que usaríamos para inferir tipos. Como no siempre, tras aplicar una operación de transformación, se obtiene un tipo normalizado, necesitamos un algoritmo que normalice de nuevo cualquier tipo recibido. Definimos el proceso de normalización de tipos mediante un conjunto de reglas que, aplicadas sucesivamente, permiten transformar un tipo en su forma normalizada. Para simplificar el manejo de restricciones en esas reglas usaremos estas funciones:

$$fst(\varphi) = \begin{cases} \alpha & \text{si } \varphi = \tau \Leftarrow \alpha \\ \alpha & \text{si } \varphi = \alpha \subseteq \tau \\ c & \text{si } \varphi = c \subseteq \tau \end{cases} \qquad snd(\varphi) = \begin{cases} \tau & \text{si } \varphi = \tau \Leftarrow \alpha \\ \tau & \text{si } \varphi = \alpha \subseteq \tau \\ \tau & \text{si } \varphi = c \subseteq \tau \end{cases}$$

Las reglas que introducimos en esta sección definen una relación entre tipos que suponemos cerrada bajo contextos. Esto viene expresado por las siguientes reglas:

$$\text{[NORM]}\ \frac{\rho \Longrightarrow \rho'}{\mathscr{C}[\rho] \Longrightarrow \mathscr{C}[\rho']} \qquad \text{[NORM-T]}\ \frac{\tau \Longrightarrow \tau'}{\mathscr{C}[\tau] \Longrightarrow \mathscr{C}[\tau']} \qquad \text{[NORM-C]}\ \frac{C \Longrightarrow C'}{\mathscr{C}[C] \Longrightarrow \mathscr{C}[C']}$$

que indica que podemos aplicar una de las transformaciones definidas por estas reglas no solo en el nivel superior del árbol abstracto del tipo, sino también en cualquiera de las subexpresiones de tipo o conjunto de restricciones. La notación $\mathscr{C}[\_]$, usada por las reglas hace referencia a una subexpresión dentro de un tipo anotado, ya sea un tipo anotado, un tipo, un esquema de tipo, un conjunto de restricciones o una restricción.

La primera transformación que vamos a describir tiene como finalidad eliminar información superflua. Por ello quitaremos todas aquellas restricciones que no aporten nada al conjunto de restricciones, es decir:

$$\text{[NOISE-A]}\ \frac{C = C' \uplus \{\varphi\} \qquad snd(\varphi) = \mathsf{any}()}{C \Longrightarrow C'}$$

En este caso, la regla [NOISE-A], elimina todas las restricciones que tienen la forma $c \subseteq \mathsf{any}()$, $\alpha \subseteq \mathsf{any}()$ o $\mathsf{any}() \Leftarrow \alpha$. Este tipo de restricciones son siempre satisfechas por cualquier instanciación y no añaden información alguna a las variables de tipo, ya que el tipo $\mathsf{any}()$ engloba a todos los valores del lenguaje.

Cuando tenemos tipos de semántica vacía, dentro de un tipo, podemos simplificar muchas veces el

resultado final normalizando al tipo `none()`. Las reglas para manejar esta situación son:

$$\text{[NONE-1]}\ \frac{}{\rho;\langle \texttt{none()};\bot\rangle \Longrightarrow \rho} \qquad \text{[NONE-2]}\ \frac{}{\tau \cup \texttt{none()} \Longrightarrow \tau} \qquad \text{[NONE-3]}\ \frac{}{\sigma \sqcup \texttt{none()} \Longrightarrow \sigma}$$

$$\text{[NONE-T]}\ \frac{\exists i \in 1..n : \tau_i = \texttt{none()}}{\{\overline{\tau_i}^n\} \Longrightarrow \texttt{none()}} \qquad \text{[NONE-L]}\ \frac{\tau = \texttt{none()} \vee \tau' = \texttt{none()}}{\texttt{nelist}(\tau,\tau') \Longrightarrow \texttt{none()}}$$

$$\text{[NONE-F]}\ \frac{(\exists i \in 1..n : \tau_i = \texttt{none()}) \vee \tau = \texttt{none()}}{\forall \overline{\alpha_i}.\left(\overline{\tau_i}^n\right) \to \tau \Longrightarrow \left(\overline{\texttt{none()}}^n\right) \to \texttt{none()}} \qquad \text{[NONE-C]}\ \frac{\tau = \texttt{none()} \vee C = \bot}{\tau\ \textbf{when}\ C \Longrightarrow \texttt{none()}}$$

$$\text{[NONE-P]}\ \frac{\tau = \texttt{none()} \vee C = \bot}{\langle \tau; C\rangle \Longrightarrow \langle \texttt{none()};\bot\rangle}$$

Las reglas [NONE-1], [NONE-2] y [NONE-3], toman una unión de tipos, esquemas de tipo o tipos anotados, y si encuentran un tipo de semántica vacía lo eliminan, dejando el resto de ramas. Aunque las reglas [NONE-1], [NONE-2] y [NONE-3] muestran el caso en el que el lado derecho tiene semántica vacía, también se podría aplicar la regla para el lado izquierdo. Las reglas [NONE-T] y [NONE-L] toman una tupla y una lista no vacía respectivamente, y si alguna de sus componentes es `none()` el resultado será entonces `none()`. La regla [NONE-F] toma un esquema de tipo funcional y si alguno de los argumentos o el resultado es de tipo `none()`, entonces el resultado será la función vacía. La regla [NONE-P] trata el caso en el que tenemos un tipo anotado que es un par, donde o τ es `none()` o el conjunto de restricciones no se no se satisface para ninguna instanciación π. Hay que señalar que, el hecho de que existan restricciones de la forma $\alpha \subseteq$ `none()` en un conjunto, no implica que el conjunto sea insatisfactible, ya que cualquier instanciación π en la que $\pi(\alpha) = \emptyset$ satisface la restricción anterior. La regla [NONE-C] es análoga a [NONE-P] pero aplicada a los tipos condicionales.

El punto más crítico para la normalización es la reubicación de las condiciones en los tipos, porque hay casos en los que no podremos elevar un **when** al tipo que contiene el fragmento donde se encuentra. El principal problema que encontraremos es que una variable de tipo que tenga semántica singular, al moverla de posición termine teniendo semántica plural. Para comprenderlo mejor, veamos el siguiente ejemplo: `nelist({`α,β`}` **when** $\beta \Leftarrow \alpha$`,[])`. En el cuerpo de la lista no vacía tenemos dos variables de tipo, en una tupla, que son α y β. Dentro del cuerpo de la lista no vacía, estas variables de tipo tienen semántica singular (es decir, se instancian a un único valor), por lo que también serán singulares en la restricción $\beta \Leftarrow \alpha$. Dada la naturaleza de esta situación, los valores que hay dentro de este tipo son listas con tuplas cuyas dos componentes contienen el mismo valor. Un ejemplo de esto último es el caso de lista `[{1,1},{2,2}]`, cuya instanciación para la tupla `{1,1}` es $\pi_1 = [\alpha \mapsto \{1\}, \beta \mapsto \{1\}]$ y para la tupla `{2,2}` es $\pi_2 = [\alpha \mapsto \{2\}, \beta \mapsto \{2\}]$, mientras que para la lista entera la instanciación final es $\pi = [\alpha \mapsto \{1,2\}, \beta \mapsto \{1,2\}]$. Como vemos, las variables α y β, fuera de la lista pasan a tener semántica plural. Entonces, si sacamos la restricción del cuerpo de la lista y pasamos a tener el tipo `nelist({`α,β`},[])` **when** $\beta \Leftarrow \alpha$, nos daremos cuenta que si bien seguimos pudiendo representar los valores contenidos en el tipo anterior, tenemos también que la semántica de este tipo es mayor que el tipo de origen. Ahora podemos tener como valor la lista `[{1,2},{2,1}]`, ya que la restricción de encaje se satisface para cualquier π tal que $\pi(\alpha) = \pi(\beta)$. En este caso, la igualdad en los

valores instanciados para α y β se comprueba globalmente para todos los elementos de la lista, en lugar de exigir la igualdad para cada una de las tuplas contenidas en ella.

Antes de mostrar las reglas para normalizar los tipos condicionales, tenemos que definir dos funciones que vamos a necesitar para extraer la información de los tipos condicionales:

$$get\_type(\tau) = \begin{cases} \tau' & \text{si } \tau = \tau' \textbf{ when } C' \\ \tau & \text{e.o.c.} \end{cases} \qquad get\_constraints(\tau) = \begin{cases} C' & \text{si } \tau = \tau' \textbf{ when } C' \\ \emptyset & \text{e.o.c.} \end{cases}$$

La función *get_type* nos permite obtener el tipo de un **when** y *get_constraints* obtener las restricciones del tipo condicional. Ahora podremos definir las siguientes reglas de transformación:

$$\text{[MOVE-W]}\ \frac{}{\tau \textbf{ when } C \textbf{ when } C' \Longrightarrow \tau \textbf{ when } C \cup C'}$$

$$\text{[MOVE-N]}\ \frac{}{\texttt{nelist(}\tau\texttt{,}\tau' \textbf{ when } C\texttt{)} \Longrightarrow \texttt{nelist(}\tau\texttt{,}\tau'\texttt{)} \textbf{ when } C}$$

$$\text{[MOVE-T]}\ \frac{C = \bigcup_{i=1}^{n} get\_constraints(\tau_i) \qquad C \neq \emptyset}{\{\overline{\tau_i}^n\} \Longrightarrow \{\overline{get\_type(\tau_i)}^n\} \textbf{ when } C}$$

$$\text{[MOVE-F]}\ \frac{C = \bigcup_{i=1}^{n} get\_constraints(\tau_i) \cup get\_constraints(\tau) \qquad C \neq \emptyset}{\left(\overline{\tau_i}^n\right) \to \tau \Longrightarrow \left(\overline{get\_type(\tau_i)}^n\right) \to get\_type(\tau) \textbf{ when } C}$$

Ninguna de estas reglas corre el peligro de mover una restricción con variables singulares a una posición plural. En el caso de las listas no vacías, los tipos con semántica singular en el tipo τ' seguirán teniendo semántica singular en el tipo $\texttt{nelist(}\tau\texttt{,}\tau'\texttt{)}$, por lo que no corremos peligro al elevar las restricciones a un contexto superior. En el caso de los tipos funcionales, estamos moviendo las restricciones de los parámetros al tipo que representa el resultado, con lo que no estamos saliendo del contexto del tipo funcional. Volviendo a las listas no vacías, existen una regla que, bajo ciertas condiciones, nos permite elevar restricciones sueltas que pertenecen al tipo del cuerpo, fuera del tipo lista no vacía:

$$\text{[MOVE-B]}\ \frac{ftv(\varphi) \cap itv(\tau \textbf{ when } C) = \emptyset}{\texttt{nelist(}\tau \textbf{ when } C \cup \{\varphi\}\texttt{,}\tau'\texttt{)} \Longrightarrow \texttt{nelist(}\tau \textbf{ when } C\texttt{,}\tau'\texttt{)} \textbf{ when } \{\varphi\}}$$

La condición es que ninguna de las variables de tipo de la restricción debe existir en el tipo del cuerpo de la lista no vacía. Si una restricción no menciona variables que se encuentran dentro del tipo que contiene los valores de una lista no vacía, no existe el riesgo de alterar la condición de singularidad al elevar dicha restricción al nivel del tipo lista.

Otra situación delicada es qué hacer con las restricciones que están dentro de una restricción. En principio la sintaxis de tipos normalizados no permite tener una restricción como la siguiente $\alpha \subseteq (\beta \textbf{ when } \beta \subseteq \texttt{atom()})$, por lo que tendríamos que extraer las restricciones fuera de la restricción. Sin embargo no siempre es correcto extraer una restricción que esté contenida dentro de otra restricción, como vimos en el ejemplo $\pi \vDash \{\alpha \subseteq \beta \textbf{ when } \beta \subseteq \texttt{integer()}\}$ de la sección 5.1.1, donde sacando la res-

tricción $\beta \subseteq \texttt{integer()}$ ganábamos precisión involuntariamente, dando como resultado situaciones en las que no podemos normalizar los tipos recibidos. Para solucionar esto, la solución drástica que tomamos es eliminar las restricciones internas, perdiendo precisión al hacerlo. Asumimos esta pérdida de precisión porque este tipo de restricciones como la anterior no aparecen de manera natural durante el proceso de inferencia, por los siguientes motivos. Primero, al generar los tipos, los únicos **when** que se generan son para los esquemas de tipo funcionales, respetando la sintaxis normalizada. Segundo, con la operación del ínfimo se generan **when** en dos casos: variables de tipo y listas no vacías. Con las listas no vacías, al generar restricciones en los componentes de un tipo unión estaría normalizado el resultado. Tercero, las reglas de transformación que definen el proceso de simplificación de tipos (que se describirá en las secciones siguientes) no generan restricciones anidadas dentro de una restricción de subconjunto.

Entonces, para extraer restricciones contenidas dentro de otras restricciones, tendremos las siguientes reglas:

$$\text{[MOVE-1]}\ \frac{}{\{\tau \textbf{ when } C \cup \{\tau' \Leftarrow \beta\} \Leftarrow \alpha\} \cup C' \Longrightarrow \{\tau \textbf{ when } C \Leftarrow \alpha, \tau' \Leftarrow \beta\} \cup C'}$$

$$\text{[MOVE-2]}\ \frac{}{\{\tau \textbf{ when } C \cup \{\tau' \subseteq \tau''\} \Leftarrow \alpha\} \cup C' \Longrightarrow \{\tau \textbf{ when } C \Leftarrow \alpha, \tau' \subseteq \tau''\} \cup C'}$$

En los dos casos estamos tratando restricciones de encaje $\tau \Leftarrow \alpha$, ya que son los únicos casos donde a lo largo del proceso de inferencia podría aparecer un **when** en el tipo τ que restringe a α. Para normalizar las restricciones de subconjunto, pasaríamos de $\alpha \subseteq \tau$ **when** C a la restricción $\alpha \subseteq \tau$, perdiendo precisión por el camino. Con la regla [MOVE-1] podemos elevar una restricción de encaje, sin exigir ninguna condición de por medio. En la regla [MOVE-2] tenemos un caso análogo al de la regla [MOVE-1], pero la restricción a elevar es de subconjunto en lugar de encaje.

Cuando tenemos esquemas de tipo funcionales, después de hacer una transformación del tipo funcional, podemos querer modificar las variables de tipo que son cerradas para añadir las variables frescas recién creadas o para quitar variables en desuso. Para realizar estas tareas tenemos las siguientes reglas:

$$\text{[SCHEME-1]}\ \frac{\alpha \notin ftv(\tau)}{\forall \alpha.\tau \Longrightarrow \tau}$$

Con la regla [SCHEME-1], si una variable de tipo no aparece libre en τ, nos limitamos a eliminarla del esquema, ya que no hace falta cerrarla porque no aparece.

En la sección 5.1.4 hemos hablado de los tipos $\cup$-normalizados, que son necesarios para poder realizar la operación del ínfimo con los tipos unión. Para normalizar un tipo no hace falta que sea $\cup$-normal, pero para aplicar el operador de ínfimo sí hace falta. Por ello tenemos una regla para convertir un tipo a $\cup$-normal y otra regla para el paso inverso:

$$\text{[UNION-1]}\ \frac{C = \{\texttt{none()} \Leftarrow \alpha \mid \alpha \in itv(\tau) \setminus ftv(\tau')\}}{\tau \cup \tau' \Longrightarrow \tau \cup \tau' \textbf{ when } C}$$

$$\text{[UNION-2]}\ \frac{\alpha \notin ftv(\overline{\varphi_i}^n) \cup ftv(\tau')}{\tau \cup \tau' \textbf{ when } \{\overline{\varphi_i}^n, \texttt{none()} \Leftarrow \alpha\} \Longrightarrow \tau \cup \tau' \textbf{ when } \{\overline{\varphi_i}^n\}}$$

Con la regla [UNION-1] añadimos restricciones $\texttt{none()} \Leftarrow \alpha$ para todas aquellas variables que estén instanciadas en el resto de la unión, excluyendo las que son libres en la rama actual de la unión. Por el contrario, [UNION-2] elimina las restricciones $\texttt{none()} \Leftarrow \alpha$ siempre que α no aparezca en el resto del tipo de la rama actual de la unión.

5.4.1. Propiedades de la normalización

Antes de pasar a la propiedad principal que cumplen las reglas de normalización de tipos, vamos a necesitar algunas propiedades auxiliares. El siguiente lema establece que si una restricción se cumple para un conjunto de instanciaciones, se cumplirá también para la unión de todas ellas.

Lema 13. *Dada una restricción φ y un conjunto de instanciaciones $\{\pi_i \mid i \in I\}$. Si $\pi_i \vDash \varphi$, para todo $i \in I$, entonces $\bigcup_{i\in I} \pi_i \vDash \varphi$.*

Demostración. Distinguimos casos según φ:

- **Caso** $\alpha \subseteq \tau$

  Como $\pi \vDash \alpha \subseteq \tau$ se tiene que:

  $$\pi_i(\alpha) \subseteq \left\{v \mid (v,\pi') \in \mathscr{T}[\![\tau]\!], \pi' \subseteq \pi_i\right\}$$

  para todo $i \in I$. Además, como $\pi_i \subseteq \bigcup_{i\in I} \pi_i$, para todo $i \in I$:

  $$\begin{array}{rcl} \pi_i(\alpha) & \subseteq & \left\{v \mid (v,\pi') \in \mathscr{T}[\![\tau]\!], \pi' \subseteq \pi_i\right\} \\ & \subseteq & \left\{v \mid (v,\pi') \in \mathscr{T}[\![\tau]\!], \pi' \subseteq \bigcup_{i\in I} \pi_i\right\} \end{array}$$

  para todo $i \in I$. Por lo tanto:

  $$\bigcup_{i\in I} \pi_i(\alpha) \subseteq \left\{ v \mid (v,\pi') \in \mathscr{T}[\![\tau]\!], \pi' \subseteq \bigcup_{i\in I} \pi_i \right\}$$

  que equivale a $\bigcup_{i\in I} \pi_i \vDash \alpha \subseteq \tau$.

- **Caso** $c \subseteq \tau$

  La demostración es similar al caso anterior.

- **Caso** $\tau \Leftarrow \alpha$

  Cada π_i puede descomponerse en $\pi_i = \bigcup_{v\in\pi_i(\alpha)} \pi_{i,v}$, tales que $(v,\pi_{i,v}) \in \mathscr{T}[\![\tau]\!]$, para todo $v \in \pi_i(\alpha)$ e $i \in I$. Por lo tanto, si definimos la instanciación $\pi = \bigcup_{i\in I} \pi_i$, podremos entonces descomponer π del siguiente modo:

  $$\pi = \bigcup_{i\in I} \pi_i = \bigcup_{i\in I} \bigcup_{v\in\pi_i(\alpha)} \pi_{i,v} = \bigcup_{\substack{i\in I \\ v\in\pi_i(\alpha)}} \pi_{i,v}$$

y como se cumple que $(v, \pi_{i,v}) \in \mathcal{T}[\![\tau]\!]$, para cada $i \in I$ y $v \in \pi_i(\alpha)$, entonces $\pi \vDash \tau \Leftarrow \alpha$.

□

El siguiente lema es como el anterior, pero aplicado a conjuntos de restricciones.

Lema 14. *Dado un conjunto de restricciones C y un conjunto de instanciaciones $\{\pi_i \mid i \in I\}$. Si $\pi_i \vDash C$, para todo $i \in I$, entonces $\bigcup_{i \in I} \pi_i \vDash C$.*

Demostración. Suponemos un $C = \{\varphi_1, \ldots, \varphi_m\}$. Para cada $i \in I$, tomando $\pi_i \vDash C$ obtenemos por la semántica de los conjuntos de restricciones que $\pi_i \vDash \varphi_j$, para cada $j \in \{1..m\}$. Aplicando el lema 13, obtendremos $\bigcup_{i \in I} \pi_i \vDash \varphi_j$, para cada $j \in \{1..m\}$, que siguiendo la semántica de los conjuntos de restricciones tendremos que $\bigcup_{i \in I} \pi_i \vDash C$. □

La siguiente propiedad demuestra cómo podemos extraer un conjunto de restricciones del interior de una restricción de encaje, propiedad que vamos a necesitar para demostrar la corrección de las reglas [MOVE-1] y [MOVE-2] de normalización.

Proposición 32. *Dados un tipo τ, una variable de tipo α, un conjunto de restricciones C y una instanciación π, si $\pi \vDash \tau$* **when** *$C \Leftarrow \alpha$, entonces $\pi \vDash \{\tau \Leftarrow \alpha\} \cup C$.*

Demostración. Suponemos $\pi \vDash (\tau$ **when** $C \Leftarrow \alpha)$. Para toda $v \in \pi(\alpha)$, existe una instanciación π_v tal que $(v, \pi_v) \in \mathcal{T}[\![\tau$ **when** $C]\!]$, que implica $(v, \pi_v) \in \mathcal{T}[\![\tau]\!]$ y $\pi_v \vDash C$. Además, sabemos que $\pi = \bigcup_{v \in \pi(\alpha)} \pi_v$, que junto a $(v, \pi_v) \in \mathcal{T}[\![\tau]\!]$, obtendremos que $\pi \vDash \tau \Leftarrow \alpha$. Y, además con $\pi_v \vDash C$ y $\pi = \bigcup_{v \in \pi(\alpha)} \pi_v$, aplicando el lema 14, tendremos $\pi \vDash C$. Por lo tanto, obtenemos $\pi \vDash \{\tau \Leftarrow \alpha\} \cup C$. □

A continuación veremos la propiedad que cumplen las reglas de normalización de tipos:

Proposición 33. *Suponemos unos tipos anotados ρ y ρ', unos tipos τ y τ', unos esquemas funcionales σ y σ', unos conjuntos de restricciones C y C', unas restricciones φ y φ'.*

- *Si $\rho \Longrightarrow \rho'$, entonces $\mathcal{T}[\![\rho]\!] \subseteq \mathcal{T}[\![\rho']\!]$.*
- *Si $\tau \Longrightarrow \tau'$, entonces $\mathcal{T}[\![\tau]\!] \subseteq \mathcal{T}[\![\tau']\!]$.*
- *Si $\sigma \Longrightarrow \sigma'$, entonces $\mathcal{T}[\![\sigma]\!] \subseteq \mathcal{T}[\![\sigma']\!]$.*
- *Si $C \Longrightarrow C'$, entonces $\forall \pi \in$* **TypeInst** *: $\pi \vDash C \Longrightarrow \pi \vDash C'$.*
- *Si $\varphi \Longrightarrow \varphi'$, entonces $\forall \pi \in$* **TypeInst** *: $\pi \vDash \varphi \Longrightarrow \pi \vDash \varphi'$.*

Demostración. Por inducción sobre la estructura del tipo anotado, tipo, esquema de tipo funcional, conjunto de restricciones y restricción, vamos a demostrar caso por caso cada regla de normalización y simplificación de tipos:

- **Caso [NOISE-A]**

  Sea un conjunto de restricciones C con la forma $C' \uplus \{\varphi\}$, donde el lado derecho de la restricción φ es de tipo $\texttt{any()}$. Suponemos una instanciación π tal que $\pi \vDash C$, por lo tanto tenemos $\pi \vDash C'$ y $\pi \vDash \varphi$, demostrando que se cumple la propiedad.

- **Caso [NONE-1]**

  Sea el juicio $\rho; \langle \texttt{none()}; \bot \rangle \Longrightarrow \rho$. Como $\mathscr{T}[\![\langle \texttt{none()}; \bot \rangle]\!] = \emptyset$, se descarta en la unión de conjuntos quedando $[\![\rho]\!]$ como resultado, demostrando así la propiedad.

- **Caso [NONE-2]**

  Sea el juicio $\tau \cup \texttt{none()} \Longrightarrow \tau$. Como $\mathscr{T}[\![\texttt{none()}]\!] = \emptyset$, se descarta en la unión de conjuntos quedando $[\![\tau]\!]$ como resultado, demostrando así la propiedad.

- **Caso [NONE-3]**

  Sea el juicio $\sigma \sqcup \texttt{none()} \Longrightarrow \sigma$. Como $\mathscr{T}[\![\texttt{none()}]\!] = \emptyset$, se descarta en la unión de conjuntos quedando $[\![\sigma]\!]$ como resultado, demostrando así la propiedad.

- **Caso [NONE-T]**

  Sea el juicio $\{\overline{\tau_i}^n\} \Longrightarrow \texttt{none()}$, donde existe para un $i \in \{1..n\}$ un tipo $\tau_i = \texttt{none()}$. Como la semántica del tipo $\texttt{none()}$ es vacía, entonces la semántica de la tupla también será vacía al no tener valores para alguna de sus componentes, demostrando así la propiedad.

- **Caso [NONE-L]**

  Sea el juicio $\texttt{nelist}(\tau, \tau') \Longrightarrow \texttt{none()}$, donde el tipo $\tau = \texttt{none()}$ o el tipo $\tau' = \texttt{none()}$. Como la semántica del tipo $\texttt{none()}$ es vacía, entonces la semántica de la lista no vacía también será vacía al no tener valores para alguna de sus componentes, demostrando así la propiedad.

- **Caso [NONE-F]**

  Sea el juicio $\forall \overline{\alpha_i}.(\overline{\tau_i}^n) \to \tau \Longrightarrow \left(\overline{\texttt{none()}}^n\right) \to \texttt{none()}$, donde o bien existe para un $i \in \{1..n\}$ un tipo $\tau_i = \texttt{none()}$, o bien el tipo $\tau = \texttt{none()}$. Como la semántica del tipo $\texttt{none()}$ es vacía, entonces la semántica de la función será la función vacía al no tener valores para alguna de sus componentes, demostrando así la propiedad.

- **Caso [NONE-C]**

  Sea el juicio τ **when** $C \Longrightarrow \texttt{none()}$, donde el tipo $\tau = \texttt{none()}$ o el conjunto de restricciones $C = \bot$. Si $\tau = \texttt{none()}$, como la semántica del tipo $\texttt{none()}$ es vacía, el resultado de la semántica del tipo **when** también será vacía. Si $C = \bot$, por la semántica del **when**, como no existe ninguna π que pueda satisfacer $\pi \vDash \bot$, la semántica del resultado será vacía. Quedando así demostrada la propiedad.

- **Caso [NONE-P]**

  La demostración de la regla [NONE-P] es análoga a la demostración de la regla [NONE-C].

- **Caso [MOVE-W]**

  La demostración es trivial al ser consecuencia directa de la definición semántica de los tipos **when**.

- **Caso [MOVE-N]**

  La demostración es trivial al ser consecuencia directa de la definición semántica de los tipos lista no vacía y los tipos **when**.

- **Caso [MOVE-T]**

  La demostración es trivial al ser consecuencia directa de la definición semántica de los tipos tupla y los tipos **when**.

- **Caso [MOVE-F]**

  La demostración es trivial al ser consecuencia directa de la definición semántica de los tipos función y los tipos **when**.

- **Caso [MOVE-B]**

  Sea el juicio $\texttt{nelist(}\tau$ **when** $C \cup \{\varphi\}\texttt{,}\tau'\texttt{)} \Longrightarrow \texttt{nelist(}\tau$ **when** $C\texttt{,}\tau'\texttt{)}$ **when** $\{\varphi\}$ tal que $ftv(\varphi) \cap itv(\tau$ **when** $C) = \emptyset$. Como φ no tiene variables en común con $itv(\tau$ **when** $C)$, las instanciaciones individuales π_i para cada elemento coinciden con π en el conjunto $ftv(\varphi)$. Por lo tanto, $\pi \vDash \varphi$, demostrando así la propiedad.

- **Caso [MOVE-1]**

  La demostración es consecuencia de aplicar la proposición 32 sobre la restricción de encaje de α.

- **Caso [MOVE-2]**

  La demostración es consecuencia de aplicar la proposición 32 sobre la restricción de encaje de α.

- **Caso [SCHEME-1]**

  La demostración es trivial al ser consecuencia directa de la definición semántica de los esquemas de tipo.

- **Caso [UNION-1]**

  La demostración es consecuencia de aplicar la proposición 14 sobre cualquier tipo de la unión.

- **Caso [UNION-2]**

  La demostración es consecuencia de aplicar la proposición 14 sobre cualquier tipo de la unión.

El resto de reglas de transformación que quedan se demuestran de forma trivial aplicando la hipótesis por inducción y como consecuencia de que la semántica de un tipo es monótona con respecto a las semánticas de sus subcomponentes. □

5.5. Transformación de funciones no recursivas

Con las reglas de generación de restricciones, que vimos en la sección 5.2, si bien obtenemos un *success type* para representar la expresión analizada, enfrentamos dos problemas con este tipo resultante. Primero, el tipo obtenido contiene aplicaciones simbólicas. Aunque este tipo de restricciones tiene una semántica definida, queremos eliminarlas con el fin de ofrecer al usuario un esquema de tipo sin este tipo de restricciones. Segundo, los tipos obtenidos por la generación no son especialmente legibles y en esta sección vamos a mostrar métodos para simplificarlos.

Como los programas de *Erlang* son conjuntos de funciones, la transformación estará orientada a tratar con expresiones que definen funciones. En la sección actual vamos a suponer que estamos transformando funciones que no son recursivas. La transformación de las funciones recursivas se verá en la siguiente sección. El algoritmo en cuestión tiene que recorrer el tipo generado normalizado para buscar los conjuntos de restricciones y transformarlo para intentar compactar la información. Las reglas de transformación que se presentan en esta sección pueden aplicarse a un tipo anotado normalizado, o a cualquier restricción o subexpresión de tipo contenido en este. Para poder aplicar una regla de transformación a una subexpresión τ contenida dentro de un tipo anotado $\langle \tau' ; C' \rangle$ (que forma parte de un ρ), debemos conocer el conjunto V_0 de variables de tipo libres hay en el resto de $\langle \tau' ; C' \rangle$, y el conjunto de restricciones vigentes en el contexto en el que se encuentra τ. Por este motivo, los juicios de transformación de reglas tienen la forma $V_0 ; C_0 \vdash \tau \Longrightarrow \tau_1, \ldots, \tau_n$, recibiendo como información de entrada un conjunto de variables de tipo V_0 y un conjunto de restricciones C_0, para tomar un tipo τ (aunque también podría ser un conjunto de restricciones C o una restricción φ) y obtener una secuencia de tipos con la transformación. El motivo por el que obtenemos una secuencia es porque al sustituir las aplicaciones simbólicas de una función, usando el esquema de tipo sobrecargado que da tipo a la misma, vamos a generar varias ramas de ejecución por cada elemento de la sobrecarga. El motivo por el que para $\langle \tau' ; C' \rangle$ no se tienen en cuenta las variables libres que hay en el resto de pares contenidos por ρ, es que las instanciaciones empleadas en cada par son independientes de las del resto, porque la semántica de los tipos anotados devuelve una (θ, υ) en lugar de (υ, π), por lo que sustituir una variable en un par no afecta al resultado de otro par distinto dentro de ρ.

A continuación se muestran unas reglas que indican cómo se propaga la transformación de una determinada subexpresión de tipo, tipo anotado, o conjunto de restricciones al resto del contexto superior en el que está contenido:

$$\text{[PAIRS]} \ \frac{\forall i \in \{1..n\} : V_0 ; C_0 \vdash \langle \tau_i ; C_i \rangle \Longrightarrow \rho'_i}{V_0 ; C_0 \vdash \langle \tau_1 ; C_1 \rangle ; \cdots ; \langle \tau_n ; C_n \rangle \Longrightarrow \rho'_1 ; \cdots ; \rho'_n}$$

$$\text{[PAIR-C]} \ \frac{V_0 \cup ftv(\tau) ; C_0 \vdash C \Longrightarrow C'_1, \ldots, C'_m}{V_0 ; C_0 \vdash \langle \tau ; C \rangle \Longrightarrow \langle \tau ; C'_1 \rangle ; \ldots ; \langle \tau ; C'_m \rangle}$$

$$\text{[PAIR-T]} \ \frac{V_0 \cup ftv(C) ; C_0 \cup C \vdash \tau \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0 ; C_0 \vdash \langle \tau ; C \rangle \Longrightarrow \langle \tau'_1 ; C \rangle ; \ldots ; \langle \tau'_m ; C \rangle}$$

$$\text{[WHEN-C]}\ \frac{V_0 \cup ftv(\tau); C_0 \vdash C \Longrightarrow C'_1, \ldots, C'_m}{V_0; C_0 \vdash \tau \textbf{ when } C \Longrightarrow \tau \textbf{ when } C'_1, \ldots, \tau \textbf{ when } C'_m}$$

$$\text{[WHEN-T]}\ \frac{V_0 \cup ftv(C); C_0 \cup C \vdash \tau \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0; C_0 \vdash \tau \textbf{ when } C \Longrightarrow \tau'_1 \textbf{ when } C, \ldots, \tau'_m \textbf{ when } C}$$

$$\text{[CONS]}\ \frac{V_0 \cup ftv(C); C_0 \cup C \vdash \varphi \Longrightarrow \varphi'_1, \ldots, \varphi'_m}{V_0; C_0 \vdash C \cup \{\varphi\} \Longrightarrow C \cup \{\varphi'_1\}, \ldots, C \cup \{\varphi'_m\}}$$

$$\text{[SUB-A]}\ \frac{V_0 \cup \{\alpha\}; C_0 \vdash \tau \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0; C_0 \vdash \alpha \subseteq \tau \Longrightarrow \alpha \subseteq \tau'_1, \ldots, \alpha \subseteq \tau'_m}$$

$$\text{[SUB-C]}\ \frac{V_0; C_0 \vdash \tau \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0; C_0 \vdash c \subseteq \tau \Longrightarrow c \subseteq \tau'_1, \ldots, c \subseteq \tau'_m}$$

$$\text{[MATCH]}\ \frac{V_0 \cup \{\alpha\}; C_0 \vdash \tau \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0; C_0 \vdash \tau \Leftarrow \alpha \Longrightarrow \tau'_1 \Leftarrow \alpha, \ldots, \tau'_m \Leftarrow \alpha}$$

$$\text{[TUPLE]}\ \frac{\forall i \in \{1..n\} : V_0 \cup ftv(\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n); C_0 \vdash \tau_i \Longrightarrow \tau'_1, \ldots, \tau'_m}{\begin{array}{r} V_0; C_0 \vdash \{\tau_1, \ldots, \tau_n\} \Longrightarrow \{\tau_1, \ldots, \tau_{i-1}, \tau'_1, \tau_{i+1}, \ldots, \tau_n\}, \\ \ldots, \{\tau_1, \ldots, \tau_{i-1}, \tau'_m, \tau_{i+1}, \ldots, \tau_n\} \end{array}}$$

$$\text{[NELIST-B]}\ \frac{V_0 \cup ftv(\tau_2); C_0 \vdash \tau_1 \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0; C_0 \vdash \texttt{nelist}(\tau_1, \tau_2) \Longrightarrow \texttt{nelist}(\tau'_1, \tau_2), \ldots, \texttt{nelist}(\tau'_m, \tau_2)}$$

$$\text{[NELIST-T]}\ \frac{V_0 \cup ftv(\tau_1); C_0 \vdash \tau_2 \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0; C_0 \vdash \texttt{nelist}(\tau_1, \tau_2) \Longrightarrow \texttt{nelist}(\tau_1, \tau'_1), \ldots, \texttt{nelist}(\tau_1, \tau'_m)}$$

$$\text{[UNION]}\ \frac{V_0 \cup ftv(\tau_1); C_0 \vdash \tau_2 \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0; C_0 \vdash \tau_1 \cup \tau_2 \Longrightarrow \tau_1 \cup \tau'_1, \ldots, \tau_1 \cup \tau'_m}$$

$$\text{[SEQ]}\ \frac{\exists i \in \{1..n\} : V_0 \cup ftv(\sigma_1, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n); C_0 \vdash \sigma_i \Longrightarrow \sigma'_1, \ldots, \sigma'_m}{\begin{array}{r} V_0; C_0 \vdash \sigma_1 \sqcup \cdots \sqcup \sigma_n \Longrightarrow \sigma_1 \sqcup \cdots \sqcup \sigma_{i-1} \sqcup \sigma'_1 \sqcup \sigma_{i+1} \sqcup \cdots \sqcup \sigma_n, \\ \ldots, \sigma_1 \sqcup \cdots \sqcup \sigma_{i-1} \sqcup \sigma'_m \sqcup \sigma_{i+1} \sqcup \cdots \sqcup \sigma_n \end{array}}$$

$$\text{[SCHEME]}\ \frac{\{\overline{\alpha_i}\} \not\subseteq V_0 \qquad V_0; C_0 \vdash \tau \Longrightarrow \tau'_1, \ldots, \tau'_m \qquad \left\{\overline{\beta_{k,j}}\right\} = \{\overline{\alpha_i}\} \cup \left(ftv\left(\tau'_k\right) \setminus ftv(\tau)\right)}{V_0; C_0 \vdash \forall \overline{\alpha_i}.\tau \Longrightarrow \forall \overline{\beta_{1,j}}.\tau'_1, \ldots, \forall \overline{\beta_{m,j}}.\tau'_m}$$

$$\text{[FUN-C]}\ \frac{V_0 \cup ftv(\tau_1, \ldots, \tau_n) \cup ftv(\tau); C_0 \vdash C \Longrightarrow C'_1, \ldots, C'_m}{V_0; C_0 \vdash \left(\overline{\tau_i}^n\right) \to \tau \textbf{ when } C \Longrightarrow \left(\overline{\tau_i}^n\right) \to \tau \textbf{ when } C_1, \ldots, \left(\overline{\tau_i}^n\right) \to \tau \textbf{ when } C_m}$$

$$\text{[FUN-R]}\ \frac{V_0 \cup ftv(\tau_1, \ldots, \tau_n) \cup ftv(C); C_0 \cup C \vdash \tau \Longrightarrow \tau'_1, \ldots, \tau'_m}{V_0; C_0 \vdash \left(\overline{\tau_i}^n\right) \to \tau \textbf{ when } C \Longrightarrow \left(\overline{\tau_i}^n\right) \to \tau'_1 \textbf{ when } C, \ldots, \left(\overline{\tau_i}^n\right) \to \tau'_m \textbf{ when } C}$$

$$\text{[FUN-P]}\ \frac{\exists i \in \{1..n\} : V_0 \cup ftv(\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n) \cup ftv(\tau) \cup ftv(C); C_0 \cup C \vdash \tau_i \Longrightarrow \tau'_1, \ldots, \tau'_m}{\begin{array}{r} V_0; C_0 \vdash (\tau_1, \ldots, \tau_n) \to \tau \textbf{ when } C \Longrightarrow \left(\tau_1, \ldots, \tau_{i-1}, \tau'_1, \tau_{i+1}, \ldots, \tau_n\right) \to \tau \textbf{ when } C, \\ \ldots, \left(\tau_1, \ldots, \tau_{i-1}, \tau'_m, \tau_{i+1}, \ldots, \tau_n\right) \to \tau \textbf{ when } C \end{array}}$$

Como detalle a destacar de la aplicación de las reglas, en la regla [SCHEME] las variables cerradas

del esquema no pueden colisionar con las que tenemos en V_0. En caso de que hubiera colisión, basta con renombrar las variables cerradas con variables frescas. Luego, en la misma regla, después de la transformación tomamos las variables cerradas, más las variables libres que no estuvieran libres antes, y hacemos un nuevo cierre de variables. En caso de que sobre alguna variable de tipo en el cierre del esquema de tipo, podremos eliminarla mediante las reglas de normalización.

En las secciones siguientes vamos a ver más reglas para transformar restricciones y tipos, explicando su funcionamiento. Después detallaremos cuál es la estrategia para transformar los tipos obtenidos a partir de funciones no recursivas. Finalmente, detallaremos algunas de las propiedades de la transformación.

5.5.1. Transformación de restricciones

La primera regla que vamos a ver nos permite eliminar restricciones de forma arbitraria:

$$\text{[DELETE-C]}\ \frac{}{V_0; C_0 \vdash C \cup \{\varphi\} \Longrightarrow C}$$

Esta regla no se aplica arbitrariamente durante el proceso de simplificación. La aplicación de reglas está sujeta a una estrategia que se explicará posteriormente. En efecto, si aplicamos esta regla de manera indiscriminada, podemos perder información relevante para el tipo, con la consiguiente pérdida de precisión en el análisis. En el peor de los casos, si nos cargamos una restricción al azar, perderíamos precisión y el tipo resultante representaría más valores. Una de las condiciones que se usan para aplicar esta regla es que en restricciones como $\tau \Leftarrow \alpha$ o $\alpha \subseteq \tau$, si $\alpha \notin V_0 \cup ftv(C)$, entonces podremos borrar la restricción con seguridad, porque se tratarían de restricciones que no están aportando información al tipo final.

Durante la transformación hay varias condiciones que hay que comprobar para validar que un conjunto de restricciones no tenga semántica vacía. La siguiente condición tiene que ver con los argumentos encontrados en una restricción con una aplicación simbólica:

$$\text{[CHECK-P]}\ \frac{gsol\ [\![C_0 \cup C]\!]\ [\,] = \phi \qquad \exists i \in \{1..n\} :\ lug\ [\![\alpha_{x_i}]\!]\ \phi = \mathsf{none}()}{V_0; C_0 \vdash C \cup \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{n}\right) \Leftarrow \alpha \right\} \Longrightarrow \bot}$$

Con esta regla comprobamos que la semántica de los tipos de las variables aplicadas a una función no tengan semántica vacía, porque si lo fueran sería del todo imposible aplicar la función y por consiguiente la ejecución del programa fallaría.

Dentro de las restricciones que tenemos en un tipo, aquellas que tienen la forma $c \subseteq \tau$ especifican condiciones necesarias que han de cumplirse. Para manejar este tipo de restricciones disponemos de

dos reglas:

$$\text{[CHECK-C]}\ \frac{\begin{array}{c} gsol\ [\![C_0 \cup C]\!]\ [\,] = \phi \\ lug\ [\![\tau]\!]\ \phi = \tau' \qquad c \notin \mathscr{T}\ [\![\tau']\!]\big|_1 \end{array}}{V_0; C_0 \vdash C \cup \{c \subseteq \tau\} \Longrightarrow \bot} \qquad \text{[CHECK-D]}\ \frac{vars(\tau) = \emptyset \qquad c \in \mathscr{T}\ [\![\tau]\!]|_1}{V_0; C_0 \vdash C \cup \{c \subseteq \tau\} \Longrightarrow C}$$

Con la regla [CHECK-C] estamos suponiendo que, con la información recogida en las restricciones que se aplican al contexto actual, la restricción $c \subseteq \tau$ no puede satisfacerse. Si τ es un tipo *ground*, no es necesario utilizar las funciones *gsol* y *lug*, por lo que para comprobar si un valor pertenece a un tipo *ground* puede utilizarse la relación descrita en la sección B.1, que proporciona una manera operativa de comprobar si un tipo es subconjunto de otro. La relación descrita en dicha sección es una aproximación inferior de la relación semántica de subtipado entre tipos ground. No obstante, en el caso particular en el que se quiere comprobar si un tipo constante es subtipo de otro, esta relación es una aproximación exacta de la relación semántica de subtipado. Cuando τ es un tipo polimórfico, calculamos un tipo *ground* τ' que lo sobreaproxime. Si c no es un valor que pertenezca a τ', el conjunto de restricciones no se cumple y devolvemos el valor *bottom* para los conjuntos de restricciones. La regla [CHECK-D] solo funciona con tipos que son ya *ground*, con lo que podemos aplicar la relación de subconjunto para comprobar que c pertenece a τ, y si lo confirmamos, eliminamos la restricción porque ya no aporta ninguna información adicional. Si la restricción se encuentra en un tipo polimórfico, no nos interesa eliminar la restricción, porque sus variables de tipo podrían conectarla a otras partes del tipo final y esta conexión sería información útil a la hora de ser aplicado el tipo funcional en el que se encuentra la restricción.

Cuando tenemos restricciones que contienen una aplicación simbólica, tenemos que buscar la definición del tipo funcional que representa la función aplicada, para sustituir la aplicación simbólica por las restricciones que ligan los argumentos y el resultado de la aplicación. Estas últimas son las que hubiéramos generado con la aplicación de la regla [APP1] del sistema de derivación de tipos polimórficos del capítulo anterior. Tenemos las siguientes reglas para reemplazar aplicaciones simbólicas:

$$\text{[CALL-F]}\ \frac{\begin{array}{c} C \supseteq \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{\,n}\right) \Leftarrow \alpha \right\} \\ C_0 \cup C \supseteq \left\{ \bigsqcup_{j=1}^{m} \left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right) \Leftarrow \alpha_f \right\} \\ \forall\, j \in \{1..m\} : \mu_j = freshRenaming\left(itv\left(\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right) \cup \left\{\overline{\alpha_{j,k}}\right\}\right) \\ \forall\, j \in \{1..m\} : C_j = \left\{\overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^{\,n}\right\} \cup \left\{\beta\mu_j \subseteq \beta \mid \beta \in itv\left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau_{j,i}}^{\,n}\right) \to \tau_j\right)\right\} \end{array}}{V_0; C_0 \vdash C \Longrightarrow \overline{C \cup C_j \cup \left\{\tau_j\mu_j \Leftarrow \alpha\right\}}^{\,m}}$$

$$\text{[CALL-A]}\ \frac{C \supseteq \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{\,n}\right) \Leftarrow \alpha \right\} \qquad \tau_f = \left(\overline{\delta_i}^{\,n}\right) \to \delta \qquad C' = \left\{\overline{\alpha_{x_i} \subseteq \delta_i}^{\,n}\right\}}{V_0; C_0 \vdash C \Longrightarrow C \cup C' \cup \left\{\alpha \subseteq \delta, \tau_f \Leftarrow \alpha_f\right\}}$$

En la regla [CALL-F] estamos en la situación donde tenemos una restricción que contiene un tipo para f, que suponemos que es un esquema de tipo sobrecargado. Por cada esquema de tipo funcional en la sobrecarga, crearemos una sustitución μ_j, que sustituirá todas las variables de tipo en posiciones

instanciables y las ligadas por el esquema, por variables frescas. Luego generaremos un conjunto de restricciones C_j por cada rama, recreando las restricciones que se añadían aplicando la regla [APP] de la sección 4.5, para finalmente añadirlas a las restricciones de C junto a una nueva restricción para encajar α con el tipo del resultado del tipo funcional, aplicando a todo lo anterior sus respectivos renombramientos para evitar colisiones de las variables de tipo generadas con otras variables de tipo que procedan de otra aplicación de la misma función. Cuando no disponemos de un tipo para f, podemos usar la regla [CALL-A], para generar un tipo funcional con variables frescas y así sustituir la aplicación simbólica con un resultado, ya que al aplicar la función f es un requisito imprescindible que f tenga un tipo funcional. Las restricciones que se generan son la versión simplificada de lo que habríamos hecho siguiendo la regla [CALL-F], añadiendo además el nuevo tipo funcional τ_f que hemos generado para la regla. Tras aplicar alguna de estas dos reglas, tendríamos que usar una estrategia para eliminar más tarde con [DELETE-C] la restricción que contiene la aplicación simbólica. Esta misma estrategia es la que nos permitirá decidir si aplicamos [CALL-F] o [CALL-A] a una restricción de aplicación simbólica dada.

La sintaxis del lenguaje *Mini Erlang* no permite valores literales en la aplicación de funciones, porque ello simplificaba las reglas para la derivación de tipos. Sin embargo, a la hora de implementar el algoritmo de transformación de tipos, podemos hacer algunos cambios para admitir literales como argumentos en una aplicación a función, evitando tener que escribir expresiones como **let** K = 1 **in** '+'(X,K). Al aceptar aplicaciones de función con literales, como '+'(X, 1), tenemos que tener en cuenta que las restricciones generadas por las reglas [CALL-F] y [CALL-A]—al sustituir las aplicaciones simbólicas—van a cambiar ligeramente si se encuentran que la aplicación simbólica tiene un literal c_i en uno de sus argumentos en lugar de tener una variable α_{x_i}. Supongamos que tenemos la restricción $Z_{\alpha_f}\left(\overline{\tau_i}^n\right) \Leftarrow \alpha$:

- Cuando el argumento τ_i es una variable de tipo α_{x_i}, el resultado será la restricción $\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}$ para [CALL-F] y $\alpha_{x_i} \subseteq \delta_i$ para [CALL-A], tal como hemos visto en sus respectivas reglas, quedándose igual el resultado final para este argumento.
- Cuando el argumento τ_i es un literal c_i y el tipo del parámetro en la función es una variable de tipo, es decir que $\tau_{j,i} = \beta_{j,i}$, el resultado será la restricción $c_i \Leftarrow \beta_{j,i}\mu_j$ para [CALL-F] y $c_i \subseteq \delta_i$ para [CALL-A].
- Cuando el argumento τ_i es un literal c_i y el tipo del parámetro en la función no es una variable de tipo, el resultado será la restricción $c_i \subseteq \tau_{j,i}\mu_j$ para [CALL-F]. Este caso no se aplica a [CALL-A] porque siempre manejamos un tipo funcional con variables de tipo en los tipos de sus parámetros.

Desde el punto de vista del sistema de tipos y del algoritmo de inferencia, las restricciones de encaje $\tau \Leftarrow \alpha$ identifican el tipo α con el tipo τ. Es habitual, al generar primero las restricciones y luego sustituir las aplicaciones simbólicas, tener varias restricciones de encaje con α en el lado derecho. Esto hace necesario tener que unirlas mediante la operación del ínfimo, que vimos en la sección 5.3.3,

para encontrar la mayor cota inferior que pueda identificarse con α. La regla que usamos para este propósito es la siguiente:

$$\text{[INFIMUM]}\ \frac{\tau \sqcap \tau' = \tau'' \textbf{ when } C' \qquad ftv(\tau) \cap ftv(\tau') = \emptyset}{V_0; C_0 \vdash C \cup \{\tau \Leftarrow \alpha, \tau' \Leftarrow \alpha\} \Longrightarrow C \cup \{\tau'' \Leftarrow \alpha\} \cup C'}$$

La regla toma los tipos de dos restricciones de encaje de α existentes y aplica el ínfimo entre los lados izquierdos, para crear una nueva restricción con el resultado que sustituya a las dos anteriores, siempre y cuando los tipos en los que encaja α no tengan variables de tipo en común. Cuando decimos que el resultado del ínfimo es τ'' **when** C', en caso de que no sea un tipo condicional el resultado tendríamos de forma equivalente el tipo τ'' **when** $\emptyset$, para evitar así tener que definir dos reglas que dependieran del resultado del ínfimo. Es importante que comprobemos si el resultado es un tipo condicional, ya que la sintaxis normalizada no permite que tengamos un tipo condicional en el lado izquierdo de una restricción de encaje, con lo que tendríamos que tomar las restricciones y extraerlas para mantener el resultado normalizado.

En cuanto a la estrategia de uso de la regla [INFIMUM], existe un caso en el que no aplicaríamos esta regla. Si τ y τ' son variables de tipo, no ganaríamos mucho aplicando la regla en cuestión, porque en un hipotético caso en el que tenemos las restricciones $\{\beta \Leftarrow \alpha, \beta' \Leftarrow \alpha\}$, el resultado de $\beta \sqcap \beta'$ es el tipo β **when** $\beta' \Leftarrow \beta$, con lo que las restricciones finales que tendríamos tras aplicar la regla son $\{\beta \Leftarrow \alpha, \beta' \Leftarrow \beta\}$. Se hace difícil intuir una utilidad para este cambio. Por ello, la estrategia de aplicación de reglas se abstiene de aplicar [INFIMUM] en este caso.

Un caso particular que necesita ser procesado aparte para el ínfimo de dos tipos es el siguiente:

$$\text{[INFIMUM-X]}\ \frac{\begin{array}{c}\tau_f = \bigsqcup_{j=1}^{m} \left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau_{j,i}}^{n}\right) \to \tau_j\right) \qquad ftv(\tau) \cap ftv(\tau') = \emptyset \\ \forall\, j \in \{1..m\} : \mu_j = \left[\overline{\alpha_{j,k}/\delta_k}\right] \qquad C'_j = \left\{\overline{\tau_{j,i}\mu_j \Leftarrow \beta_i}^{n}, \tau_j\mu_j \Leftarrow \beta\right\}\end{array}}{V_0; C_0 \vdash C \cup \left\{\tau_f \Leftarrow \alpha, \left(\overline{\beta_i}^{n}\right) \to \beta \Leftarrow \alpha\right\} \Longrightarrow C \cup \{\tau_f \Leftarrow \alpha\} \cup C'_1, \ldots, C \cup \{\tau_f \Leftarrow \alpha\} \cup C'_m}$$

La regla se aplica en casos en que una misma variable encaja con un esquema de tipo funcional sobrecargado y un tipo funcional compuesto por variables de tipo, para hacer el ínfimo entre ellos. Para lograrlo hay que tomar cada rama del esquema de tipo sobrecargado, crear un renombramiento para las variables de tipo cerradas y crear las restricciones de encaje que conecten las variable de tipo β con los tipos τ del esquema de tipo funcional, una vez se les ha aplicado el renombramiento. Con cada conjunto de restricciones C'_j obtenido se construirán los m conjuntos de restricciones finales que se van a devolver con la regla, eliminando la restricción $\left(\overline{\beta_i}^{n}\right) \to \beta \Leftarrow \alpha$, que ya no es necesaria.

En general, si tenemos dos restricciones de subconjunto $\alpha \subseteq \tau_1$ y $\alpha \subseteq \tau_2$, no sería correcto reemplazarlas por $\alpha \subseteq \tau_1 \sqcap \tau_2$, ya que la semántica de la restricción resultante podría ser un subconjunto a la conjunción de las restricciones de partida. Por ejemplo, si suponemos $\alpha \subseteq \beta_1$ y $\alpha \subseteq \beta_2$, el ínfimo $\beta_1 \sqcap \beta_2$ da como resultado β_1 **when** $\beta_2 \Leftarrow \beta_1$, introduciendo una ganancia de precisión, lo cual no es correcto. No obstante, es posible realizar este tipo de transformaciones cuando los tipos situados en el

lado derecho de la restricción de subconjunto son *ground*. En este caso, podemos aplicar la siguiente regla, que transforma una restricción de subconjunto de un tipo *ground* en una de encaje, para que luego pueda ser combinada con otra mediante la regla [INFIMUM].

$$\text{[UPGRADE]}\ \frac{vars(\tau) = \emptyset}{V_0; C_0 \vdash C \cup \{\alpha \subseteq \tau\} \Longrightarrow C \cup \{\tau \Leftarrow \alpha\}}$$

Otra regla que puede resultar útil en ocasiones, es aquella que cambie la orientación de una restricción de encaje de la forma $\beta \Leftarrow \alpha$, para obtener en su lugar $\alpha \Leftarrow \beta$. Necesitamos hacer eso, porque una de las operaciones de transformación que tenemos es la sustitución de una variable de tipo por el tipo en el que encaja, y como estrategia para la sustitución es mejor elegir aquella variable de tipo que tenga el menor número de apariciones. La regla en cuestión es la siguiente:

$$\text{[SWAP]}\ \frac{}{V_0; C_0 \vdash C \cup \{\beta \Leftarrow \alpha\} \Longrightarrow C \cup \{\alpha \Leftarrow \beta\}}$$

La siguiente regla nos permite añadir una restricción trivial sobre una variable de tipo cualquiera. Este tipo de restricciones se eliminan durante la normalización con la regla [NOISE-A], pero mientras estemos transformando un tipo nos puede ser útil añadir temporalmente esta información de cara a la sustitución de variables de tipo, que veremos en la siguiente sección. La regla queda definida de la siguiente manera:

$$\text{[ADD-ANY]}\ \frac{\alpha \in \textbf{TypeVar}}{V_0; C_0 \vdash C \Longrightarrow C \cup \{\mathsf{any}() \Leftarrow \alpha\}}$$

5.5.2. Transformación de tipos

En esta sección vamos a hablar de reglas de transformación que se aplican directamente sobre tipos, en vez de sobre conjuntos de restricciones como en la sección anterior. La primera regla que vamos a ver se utiliza para dividir una unión de tipos, con el fin de generar tantas ramas de ejecución como elementos tenga la unión:

$$\text{[SPLIT]}\ \frac{}{V_0; C_0 \vdash \tau \cup \tau' \Longrightarrow \tau, \tau'}$$

Aunque tengamos definida la regla [SPLIT], no es recomendable diseñar una estrategia donde se use sin limitaciones, porque la explosión exponencial de casos puede llegar a generar tipos excesivamente grandes. Sin embargo, ya que puede ser de utilidad en alguna circunstancia concreta, la añadimos como parte de nuestro sistema de inferencia de tipos.

Una de las operaciones más importantes que hay que realizar durante la transformación es la sustitución de una variable de tipo por el tipo en el que encaja. Para poder realizar la sustitución tenemos que recurrir al operador $\tau\left[\alpha/\tau'\right]$ descrito en la sección 5.3.2. La regla en cuestión es la siguiente:

$$\text{[REPLACE]}\ \frac{C_0 = C \cup \{\tau' \Leftarrow \alpha\} \qquad \alpha \notin V_0 \cup ftv(C)}{V_0; C_0 \vdash \tau \Longrightarrow \tau\,[\alpha/\tau']}$$

En la regla tenemos un tipo τ sobre el que se va a aplicar la sustitución, siempre que se cumpla dos condiciones: que tengamos en el contexto superior una restricción de encaje de la forma $\tau' \Leftarrow \alpha$ y que α no aparezca como variable en el resto de restricciones, ni esté libre en el contexto en el que se encuentra el tipo que estamos transformando. Adicionalmente, como parte de la estrategia de aplicación de esta regla, solo la utilizaremos si α aparece en τ una única vez, o si α aparece más de una vez en τ, pero τ' es una variable de tipo. En efecto, si hubiera varias apariciones de α en τ y reemplazásemos todas por τ, se perdería la conexión entre todas las apariciones. Por ejemplo, si C_0 contiene la restricción `integer()` $\Leftarrow \alpha$, y τ fuese $\{\alpha, \alpha\}$, tras la sustitución se obtendría `{integer(),integer()}`, perdiéndose la propiedad de que el tipo $\{\alpha, \alpha\}$ representa tuplas con componentes idénticas. Sin embargo, cuando τ' es una variable de tipo, supongamos β, la restricción $\beta \Leftarrow \alpha$ indica que α se instancia a los mismos valores que β, y en este caso no se pierde precisión al realizar la sustitución, ya que estaríamos realizando algo similar a un renombramiento, salvo las excepciones descritas en la operación de sustitución. Otro caso excepcional donde podríamos hacer la sustitución, a pesar de aparecer α múltiples veces, es cuando τ' es un valor literal c, ya que α encaja con un único valor posible, y no se perdería precisión al realizar la sustitución. Por ejemplo, en el ejemplo anterior, si tuviésemos la restricción $c \Leftarrow \alpha$, tras realizar la sustitución en $\{\alpha, \alpha\}$ tendríamos $\{c, c\}$. El tipo resultante sigue imponiendo que las componentes de la tupla sean iguales.

5.5.3. Estrategia de la transformación

En las secciones anteriores hemos presentado un conjunto de reglas que permiten transformar tipos anotados, conjuntos de restricciones y tipos. Dado uno de estos últimos, podría haber más de una regla aplicable. Por este motivo, y para que nuestro algoritmo de inferencia sea determinista, hay que fijar una estrategia que permita decidir, dado un tipo anotado, restricción, o tipo, qué regla aplicar. Para ello vamos a suponer que recibimos un tipo anotado ρ de entrada, que sería el tipo anotado obtenido mediante el proceso de generación de restricciones que hemos visto en la sección 5.2. También recibiremos un conjunto de restricciones iniciales C_0, que contendrá los tipos que las funciones de la biblioteca estándar (por ejemplo, el tipo para el operador suma y demás operadores). El algoritmo visitará cada fragmento contenido en ρ, buscando esquemas de tipos funcionales para transformarlos, tomando primero aquellos esquemas que se encuentren a mayor profundidad en el árbol sintáctico de ρ. Para realizar este recorrido vamos a usar las reglas que que describimos al principio de esta sección. En todo momento vamos a suponer que estaremos trabajando con tipos normalizados.

Esquemas de tipo funcional

Cuando nos encontramos un esquema de tipo funcional $\forall \overline{\alpha_k}.\left(\overline{\tau_i}^n\right) \to \tau$ **when** C, sin importar si está en un esquema sobrecargado o no, tenemos que transformar primero los tipos $\overline{\tau_i}^n$ y τ para simplificarlos. Si este esquema es un tipo obtenido a partir del proceso de generación de restricciones, los tipos de los parámetros serán variables de tipo, con lo que no habrá nada que simplificar.

Tras simplificar los tipos, vamos a separar el conjunto de restricciones C del tipo funcional, quedando este como $(\overline{\tau_i}^n) \to \tau$. Tomamos C y lo transformamos con la estrategia que vamos a describir en el siguiente apartado. Esto nos devolverá una secuencia de conjuntos de restricciones $C_1, \ldots, C_m$, que usaremos para componer m esquemas de tipo funcionales diferentes. Para cada C_j hay que hacer las siguientes tareas:

- Comprobar con la regla [CHECK-P], para cada aplicación simbólica que tenemos en C_j, que las variables de tipo usadas—como parámetros de la aplicación—no tienen `none()` como tipo *least upper ground*. En caso de que alguna variable de tipo esté contenida por la semántica de `none()`, el tipo final que usaríamos como resultado del tipo funcional sería `none()`.[1]
- En caso de no haber obtenido `none()` en el paso anterior, eliminar con la regla [DELETE-C] todas las restricciones con aplicaciones simbólicas del resultado final, porque no nos harán falta para construir el tipo que vamos a devolver al usuario.
- Recorrer todas las restricciones $\beta \Leftarrow \alpha$ que haya, para sustituir una variable de tipo por la otra usando la regla [REPLACE], preferiblemente la que tenga el menor número de apariciones, aplicando la regla [SWAP] si es necesario. Para ello usaremos la definición de sustitución definida en la sección 5.3.2. Supongamos que estamos sustituyendo α por β. Después de aplicar la sustitución con [REPLACE] pueden pasar dos cosas:
  - α no aparece ya más en C_j ni en $(\overline{\tau_i}^n) \to \tau$. Si no aparece libre en V_0, podremos eliminar la restricción $\beta \Leftarrow \alpha$ de C_j de forma segura. Si α aparece como una variable libre en V_0, no podremos eliminar la restricción de C_j.
  - α sigue apareciendo en C_j o $(\overline{\tau_i}^n) \to \tau$. Esto es posible, porque la definición de sustitución indicada en la sección 5.3.2 puede no reemplazar algunas de las apariciones de α en un tipo, ya que el hacerlo alteraría la semántica de dicho tipo obteniendo uno más preciso, lo cual sería incorrecto.
- Cuando tenemos restricciones $\tau \Leftarrow \alpha$, donde τ no es una variable de tipo, usaremos la regla [REPLACE] si las condiciones que dan para poder hacerlo. En caso de tener éxito con la sustitución, eliminaremos la restricción del conjunto C_j.
- Después de realizar las sustituciones, tomaremos C_j y trataremos de volver a transformarlo para ver si se produce alguna nueva simplificación. Si obtenemos el mismo conjunto sin modificar, habremos terminado con el proceso. Si no, tendremos con la estrategia mostrada más adelante que volver a repetir el paso anterior, donde se aplica la regla [REPLACE].

Una vez hemos terminado de modificar el conjunto de restricciones, tendremos un conjunto C'_j de restricciones transformadas y un tipo funcional $\left(\overline{\tau'_i}^n\right) \to \tau'$ transformado. Con esta información obtendremos el tipo funcional $\left(\overline{\tau'_i}^n\right) \to \tau'$ **when** C'_j, con las restricciones insertadas en el resultado del

[1] Podemos también sustituir los tipos de los parámetros del tipo funcional por `none()`, para unificar todos los tipos que hay en el tipo funcional resultante.

tipo funcional. Con este resultado podremos volver a hacer el cierre de variables de tipo, añadiendo aquellas que son libres y no eran libres en el esquema de tipo funcional inicial. Mediante el proceso de normalización del tipo se eliminarán aquellas variables cerradas que no aparecen en el tipo final.

Todo este proceso aplicado a los esquemas de tipo funcionales, excluyendo el proceso final de cierre, se puede aplicar a pares $\langle \tau; C \rangle$ o a tipos condicionales τ **when** C sin ningún cambio en la estrategia a seguir.

Conjuntos de restricciones

La estrategia de transformación de un conjunto de restricciones C que recibimos de entrada, comienza aplicando la regla [CHECK-D] todas las veces que se pueda, para comprobar que se cumplen las restricciones de la forma $c \subseteq \tau$, siendo c un literal y τ un tipo *ground*. También usaremos [CHECK-C] para detectar si queda alguna condición que no se cumpla, en cuyo caso termina la transformación devolviendo el valor *bottom*.

El siguiente paso es calcular el grafo de dependencias que existe entre las variables de tipo y las aplicaciones simbólicas. Es decir, supongamos que f es una función que es aplicada en un programa. Por un lado tendremos α_f, que estará asociada a una serie de restricciones de encaje que delimitarán el tipo funcional que α_f representa, y por otro lado tendremos la restricción con la aplicación simbólica $Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^n\right) \Leftarrow \alpha$. Antes de aplicar la regla [CALL-F] sobre la aplicación simbólica, tendremos que haber transformado las restricciones asociadas a α_f, para obtener el tipo funcional transformado asociado, ya que de hacerlo en el orden incorrecto perderíamos precisión. Con este grafo podremos determinar el orden para transformar las restricciones de encaje asociadas a cada variable de tipo, además de saber qué tipos funcionales son recursivos. En esta sección no vamos a describir cómo tratamos el procesamiento de las funciones recursivas. Eso lo haremos en la siguiente sección. En cuanto a la dependencia existente entre unas variables de tipos con otras variables de tipos, no existe el mismo problema de pérdida de precisión que hay con las aplicaciones simbólicas, porque el proceso itera varias veces entre la transformación de restricciones y la sustitución de variables de tipo. Además se tiene cuidado de no sustituir una variable α con [REPLACE] si quedan información pendiente de obtener, como por ejemplo que α sea uno de los argumentos en una restricción de aplicación simbólica, que con la regla [CALL-F] daría como resultado que se añaden nuevas restricciones de encaje con α, aportando nueva información.

Como ya dijimos previamente, una variable de tipo puede aparecer en el lado derecho de restricciones que tienen la forma $\tau \Leftarrow \alpha$. Es posible aplicar la regla [REPLACE] utilizando este tipo de restricciones de encaje, pero no utilizando restricciones de subconjunto. Sin embargo, en una restricción de subconjunto, si el tipo τ en la derecha es *ground*, podemos usar la regla [UPGRADE] para convertirla a una restricción de encaje. Evitaremos transformar restricciones de subconjunto en las que el tipo τ en la derecha sea también una variable de tipo, porque estas restricciones son útiles para relacionar los tipos que hay en un tipo funcional con los tipos que hay en el exterior.

Siguiendo el grafo de dependencias tomaremos una variable de tipo α, para transformar las restricciones de encaje asociadas a esta variable. El primer paso es comprobar si existen aplicaciones simbólicas que procesar mediante las reglas [CALL-F] y [CALL-A]. Como estamos realizando las transformaciones en orden de dependencias, tenemos la garantía de que solo se aplicará [CALL-A] en aquellos casos en los que no hay ninguna restricción asociada al tipo de la función aplicada. Un detalle, al hacer la sustitución de la aplicación simbólica, es que podemos estar añadiendo nuevas restricciones que afectan a variables que hemos procesado previamente, con lo que hay que volver a procesarlas para terminar su simplificación. A continuación, usando la regla [INFIMUM], tomaremos las restricciones de encaje que contienen α en el lado derecho para aplicar el ínfimo e ir reduciendo el número de estas restricciones. Como ya mencionamos antes, no aplicaremos el ínfimo si los tipos son dos variables de tipo. Para el resto de casos en los que se aplica el ínfimo, en caso de obtener restricciones nuevas—porque el resultado sea un tipo condicional—estas podrían estar aportando información nueva a variables de tipo procesadas previamente, por lo que de nuevo tendremos que reprocesarlas, para intentar simplificar el tipo final obtenido por el algoritmo.

5.5.4. Propiedades de la transformación

A continuación expondremos un teorema que indica que, al aplicar las reglas de transformación vistas en las secciones anteriores, la semántica de un tipo anotado transformado mediante estas reglas es mayor o igual que la semántica del tipo anotado inicial. El mismo resultado se aplica a tipos, esquemas de tipos sobrecargados, restricciones, y conjunto de restricciones. Antes de ello, necesitamos definir la operación de restricción de un tipo anotado ρ a un conjunto de restricciones C. Esta definición descarta todos los pares de la semántica de un tipo anotado que no puedan obtenerse a partir de una instanciación que satisfaga C. En particular:

$$\mathscr{T}[\![\rho_1;\rho_2]\!]\restriction_{[\![C]\!]} = \mathscr{T}[\![\rho_1]\!]\restriction_{[\![C]\!]} \cup \mathscr{T}[\![\rho_2]\!]\restriction_{[\![C]\!]}$$
$$\mathscr{T}[\![\langle\tau;\Gamma\rangle]\!]\restriction_{[\![C]\!]} = \left\{(\theta,\upsilon) \mid \theta \in \mathscr{T}^{\pi}_{Env}[\![\Gamma]\!], (\upsilon,\pi) \in \mathscr{T}[\![\tau]\!], \pi \vDash C\right\}$$

También tendremos de forma análoga para los tipos τ la notación $\mathscr{T}[\![\tau]\!]\restriction_{[\![C]\!]}$, que restringe un tipo bajo un conjunto de restricciones y se define así:

$$\mathscr{T}[\![\tau]\!]\restriction_{[\![C]\!]} = \{(\upsilon,\pi) \mid (\upsilon,\pi) \in \mathscr{T}[\![\tau_i]\!], \pi \vDash C\}$$

A continuación veremos la propiedad que cumplen las reglas de transformación de tipos:

Teorema 4. *Suponemos un conjunto de variables de tipo V_0, un conjunto de restricciones C_0, unos tipos anotados ρ_1 y ρ_2, unos tipos τ_0 y $\tau_1,\ldots,\tau_n$, unos esquemas funcionales σ_0 y $\sigma_1,\ldots,\sigma_n$, unos conjuntos de restricciones C'_0 y $C_1,\ldots,C_n$, unas restricciones φ_0 y $\varphi_1,\ldots,\varphi_n$, tales que $ftv(C_0) \subseteq V_0$.*

- *Si $V_0; C_0 \vdash \rho_1 \Longrightarrow \rho_2$ y $(\theta,\upsilon) \in \mathscr{T}[\![\rho_1]\!]\restriction_{[\![C_0]\!]}$, entonces $(\theta,\upsilon) \in \mathscr{T}[\![\rho_2]\!]\restriction_{[\![C_0]\!]}$.*

- *Si* $V_0; C_0 \vdash \tau_0 \Longrightarrow \tau_1, \ldots, \tau_n$ *y* $(v, \pi) \in \mathcal{T}[\![\tau_0]\!] \restriction_{[\![C_0]\!]}$, *entonces:*

$$\exists \pi' \in \textbf{TypeInst}: \quad (v, \pi') \in \bigcup_{i=1}^{n} \mathcal{T}[\![\tau_i]\!] \restriction_{[\![C_0]\!]} \quad \wedge \quad \pi' \equiv \pi \text{ (módulo } \setminus X)$$

  donde X *es el conjunto de variables de tipo libres frescas que aparecen en* $\tau_1, \ldots, \tau_n$.

- *Si* $V_0; C_0 \vdash \sigma_0 \Longrightarrow \sigma_1, \ldots, \sigma_n$ *y* $(v, \pi) \mathcal{S}[\![\sigma_0]\!] \restriction_{[\![C_0]\!]}$, *entonces:*

$$\exists \pi' \in \textbf{TypeInst}: \quad (v, \pi') \in \bigcup_{i=1}^{n} \mathcal{S}[\![\sigma_i]\!] \restriction_{[\![C_0]\!]} \quad \wedge \quad \pi' \equiv \pi \text{ (módulo } \setminus X)$$

  donde X *es el conjunto de variables de tipo libres frescas que aparecen en* $\sigma_1, \ldots, \sigma_n$.

- *Si* $V_0; C_0 \vdash C_0' \Longrightarrow C_1, \ldots, C_n$ *y* $\forall \pi \in \textbf{TypeInst}: \pi \vDash C_0 \wedge \pi \vDash C_0'$, *entonces:*

$$\exists \pi' \in \textbf{TypeInst}: \quad \bigvee_{i=1}^{n} \pi' \vDash C_i \quad \wedge \quad \pi' \equiv \pi \text{ (módulo } \setminus X)$$

  donde X *es el conjunto de variables de tipo libres frescas que aparecen en* $C_1, \ldots, C_n$.

- *Si* $V_0; C_0 \vdash \varphi_0 \Longrightarrow \varphi_1, \ldots, \varphi_n$ *y* $\forall \pi \in \textbf{TypeInst}: \pi \vDash C_0 \wedge \pi \vDash \varphi_0$, *entonces:*

$$\exists \pi' \in \textbf{TypeInst}: \quad \bigvee_{i=1}^{n} \pi' \vDash \varphi_i \quad \wedge \quad \pi' \equiv \pi \text{ (módulo } \setminus X)$$

  donde X *es el conjunto de variables de tipo libres frescas que aparecen en* $\varphi_1, \ldots, \varphi_n$.

Demostración. Vamos a demostrar caso por caso cada regla de transformación:

- **Caso [DELETE-C]**

  Suponemos un juicio $V_0; C_0 \vdash C \cup \{\varphi\} \Longrightarrow C$ y una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash (C \cup \{\varphi\})$. Por la definición semántica de las restricciones obtenemos que $\pi \vDash C$ y $\pi \vDash \varphi$. Como podemos prescindir de $\pi \vDash \varphi$, al quedarnos con $\pi \vDash C$, y como $\pi' = \pi$, porque $X = \emptyset$, el teorema se cumple para esta regla.

- **Caso [CHECK-P]**

  Suponemos un juicio $V_0; C_0 \vdash C \cup \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{n}\right) \Leftarrow \alpha \right\} \Longrightarrow \bot$ y una instanciación π. Vamos a demostrar este caso por reducción al absurdo. Suponemos π' tal que $\pi' \equiv \pi$ (módulo $\setminus X$). Sabemos que $\pi \vDash C_0$ y $\pi \vDash C \cup \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{n}\right) \Leftarrow \alpha \right\}$, que por la definición semántica de las restricciones obtenemos que $\pi \vDash C$ y $\pi \vDash \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^{n}\right) \Leftarrow \alpha \right\}$. Con la proposición 19, tomando $\pi \vDash C_0$, $\pi \vDash C$ y *gsol* $[\![C_0 \cup C]\!]\ [\,] = \phi$, obtenemos que $\pi \subseteq [\![\phi]\!]$. Como también tenemos que $\pi' \subseteq \pi \subseteq [\![\phi]\!]$, con la proposición 19 obtendremos $(\pi'(\alpha_{x_i}), \pi') \in \mathcal{T}[\![\alpha_{x_i}]\!]$, para cada $i \in \{1..n\}$. Esto último contradice

la hipótesis inicial, donde tenemos que $lug\ [\![\alpha_{x_i}]\!]\ \phi = \texttt{none()}$, para cada $i \in \{1..n\}$, lo que implica con la proposición 19 que $\phi(\alpha_{x_i}) = \emptyset$, lo que nos lleva a tener en realidad el conjunto de restricciones $\bot$, cumpliendo así el teorema para esta regla.

- **Caso [CHECK-C]**

  Suponemos un juicio $V_0; C_0 \vdash C \cup \{c \subseteq \tau\} \Longrightarrow \bot$, tal que $gsol\ [\![C_0 \cup C]\!]\ [\,] = \phi$, $lug\ [\![\tau]\!]\ \phi = \tau'$ y $c \notin \mathscr{T}\ [\![\tau']\!]\big|_1$. Vamos a demostrar este caso por reducción al absurdo. Suponemos π' tal que $\pi' \equiv \pi$ (módulo $\setminus X$). Sabemos que $\pi \vDash C_0$ y $\pi \vDash (C \cup \{c \subseteq \tau\})$, que por la definición semántica de las restricciones obtenemos que $\pi \vDash C$ y $\pi \vDash \{c \subseteq \tau\}$. Con la proposición 19, tomando $\pi \vDash C_0$, $\pi \vDash C$ y $gsol\ [\![C_0 \cup C]\!]\ [\,] = \phi$, obtenemos que $\pi \subseteq [\![\phi]\!]$. Además, si $\pi \vDash c \subseteq \tau$, significa que $(c, \pi') \in \mathscr{T}\ [\![\tau]\!]$ para algún $\pi' \subseteq \pi$. Como también tenemos que $\pi' \subseteq \pi \subseteq [\![\phi]\!]$, con la proposición 19 obtendremos $(c, \pi') \in \mathscr{T}\ [\![\tau']\!]$. Esto último contradice la hipótesis de que $c \notin \mathscr{T}\ [\![\tau']\!]\big|_1$, lo que nos lleva a tener en realidad el conjunto de restricciones $\bot$, cumpliendo así el teorema para esta regla.

- **Caso [CHECK-D]**

  Suponemos un juicio $V_0; C_0 \vdash C \cup \{c \subseteq \tau\} \Longrightarrow C$, donde $vars(\tau) = \emptyset$ porque τ es un tipo *ground* y $c \in \tau$, y una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash (C \cup \{c \subseteq \tau\})$. Por la semántica de los conjuntos de restricciones sabemos que se cumple que $\pi \vDash C$ y $\pi \vDash \{c \subseteq \tau\}$. Como podemos prescindir de $\pi \vDash \{c \subseteq \tau\}$, al quedarnos con $\pi \vDash C$, y como $\pi' = \pi$, porque $X = \emptyset$, el teorema se cumple para esta regla.

- **Caso [CALL-F]**

  Suponemos un juicio $V_0; C_0 \vdash C \Longrightarrow \overline{C \cup C_j \cup \{\tau_j \mu_j \Leftarrow \alpha\}}^m$, tal que:

$$C \supseteq \left\{ Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^n\right) \Leftarrow \alpha \right\} \quad \wedge \quad C_0 \cup C \supseteq \left\{ \bigsqcup_{j=1}^m \left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \Leftarrow \alpha_f \right\}$$
$$\wedge \quad \forall j \in \{1..m\} : \mu_j = freshRenaming\left(itv\left(\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \cup \left\{\overline{\alpha_{j,k}}\right\}\right) \quad \wedge$$
$$\forall j \in \{1..m\} : C_j = \left\{\overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^n\right\} \cup \left\{\beta\mu_j \subseteq \beta \mid \beta \in itv\left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)\right\}$$

  Suponemos una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash C$, por lo que tenemos que $\pi \vDash Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^n\right) \Leftarrow \alpha$ y $\pi \vDash \bigsqcup_{j=1}^m \left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right) \Leftarrow \alpha_f$. Aplicando el lema 10 obtendremos que:

$$\pi \vDash \left\{ Z_\alpha\left(\overline{\alpha_{x_i}}^n\right) \Leftarrow \alpha \right\}$$
$$\Updownarrow$$
$$\forall j \in \{1..m\}.\pi_j \vDash \left(\left\{\overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^n, \tau_j\mu_j \Leftarrow \alpha\right\} \cup \left\{\beta\mu_j \subseteq \beta \mid \beta \in itv\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)\right\}\right)$$

  donde la instanciación $\pi_j \equiv \pi$ (módulo $\setminus rng\,\mu_j$), para todo $j \in \{1..m\}$. Definiremos $rng\,\mu_j$ como X_j, para todo $j \in \{1..m\}$, para representar las variables de tipo frescas obtenidas por el renombramiento de cada rama, siendo el conjunto $X = \bigcup_{j=1}^m X_j$. Como las variables libres de C no

existen en X, tendremos $\pi_j \vDash C$, para cada $j \in \{1..m\}$. Entonces, para todo $j \in \{1..m\}$:

$$\pi_j \vDash \left(\left\{\overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^n, \tau_j\mu_j \Leftarrow \alpha\right\} \cup \left\{\beta\mu_j \subseteq \beta \mid \beta \in itv\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)\right\}\right) \quad \wedge \quad \pi_j \vDash C$$
$$\Updownarrow$$
$$\pi_j \vDash \left(\left\{\overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^n\right\} \cup \left\{\beta\mu_j \subseteq \beta \mid \beta \in itv\left(\forall \overline{\alpha_{j,i}}.\left(\overline{\tau_{j,i}}^n\right) \to \tau_j\right)\right\}\right) \quad \wedge \quad \pi_j \vDash \tau_j\mu_j \Leftarrow \alpha \quad \wedge \quad \pi_j \vDash C$$
$$\Updownarrow$$
$$\pi_j \vDash C_j \quad \wedge \quad \pi_j \vDash \tau_j\mu_j \Leftarrow \alpha \quad \wedge \quad \pi_j \vDash C$$
$$\Updownarrow$$
$$\pi_j \vDash \left(C \cup C_j \cup \left\{\tau_j\mu_j \Leftarrow \alpha\right\}\right)$$

Como la intersección $\bigcap_{j=1}^m X_j = \emptyset$, podemos unificar todas las instancias $\overline{\pi_j}^m$, de modo que $\pi' = \bigcup_{j=1}^m \pi_j$. Por lo tanto se cumple que:

$$\bigvee_{j=1}^m \pi' \vDash \left(C \cup C_j \cup \left\{\tau_j\mu_j \Leftarrow \alpha\right\}\right) \quad \wedge \quad \pi' \equiv \pi \text{ (módulo } \setminus X)$$

demostrando así el teorema para esta regla.

- **Caso [CALL-A]**

Suponemos un juicio $V_0; C_0 \vdash C \Longrightarrow C \cup \left\{\overline{\alpha_{x_i} \subseteq \delta_i}^n\right\} \cup \left\{\alpha \subseteq \delta, \left(\overline{\delta_i}^n\right) \to \delta \Leftarrow \alpha_f\right\}$, donde $\overline{\delta_i}^n$ y δ son variables de tipo frescas, y $C \supseteq \left\{Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^n\right) \Leftarrow \alpha\right\}$. Suponemos una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash C$, por lo que tenemos que $\pi \vDash Z_{\alpha_f}\left(\overline{\alpha_{x_i}}^n\right) \Leftarrow \alpha$. Por la definición 4, sabemos que para cada $f \in \pi(\alpha_f)$, $v_i \in \pi(\alpha_{x_i})$ (donde $i \in \{1..n\}$), $v \in \pi(\alpha)$, se cumple que $\left(\left(\overline{v_i}^n\right), v\right) \in f$. Suponemos el tipo $\left(\overline{\delta_i}^n\right) \to \delta$ y una instanciación π' tal que $(\pi(\alpha_f), \pi') \in \mathcal{T}\left[\!\left[\left(\overline{\delta_i}^n\right) \to \delta\right]\!\right]$, donde $\pi' \equiv \pi$ (módulo $\setminus X$) y $X = \left\{\overline{\delta_i}^n, \delta\right\}$, de modo que $\pi' \vDash \left(\overline{\delta_i}^n\right) \to \delta \Leftarrow \alpha_f$. Como $ftv(C) \cap X = \emptyset$, tendremos que $\pi' \vDash C$, así como $f \in \pi'(\alpha_f)$, $v \in \pi'(\alpha)$ y $v_i \in \pi'(\alpha_{x_i})$, para cada $i \in \{1..n\}$. Como sabemos que $\left(\left(\overline{v_i}^n\right), v\right) \in f$, con lo anterior tendremos que $v_i \in \pi'(\delta_i)$, para cada $i \in \{1..n\}$, y $v \in \pi'(\delta)$. Suponiendo una instanciación $\pi^\circ \subset \pi'$ tal que $\pi^\circ(\delta) = \{v\}$, tendremos que:

$$\pi'(\alpha) \subseteq \left\{v \mid (v, \pi^\circ) \in [\![\delta]\!], \pi^\circ \subset \pi'\right\} \qquad \Longleftrightarrow \qquad \pi' \vDash \alpha \subseteq \delta$$

De forma análoga, suponiendo para cada $i \in \{1..n\}$ una instanciación $\pi_i^\bullet \subset \pi'$ tal que $\pi_i^\bullet(\delta_i) = \{v_i\}$, tendremos que:

$$\pi'(\alpha_{x_i}) \subseteq \left\{v \mid (v, \pi_i^\bullet) \in [\![\delta_i]\!], \pi_i^\bullet \subset \pi'\right\} \qquad \Longleftrightarrow \qquad \pi' \vDash \alpha_{x_i} \subseteq \delta_i$$

Por lo tanto tenemos que:

$$\pi' \vDash \left(C \cup \left\{\overline{\alpha_{x_i} \subseteq \delta_i}^n\right\} \cup \{\alpha \subseteq \delta\} \cup \left\{\left(\overline{\delta_i}^n\right) \to \delta \Leftarrow \alpha_f\right\}\right)$$

demostrando así el teorema para esta regla.

- **Caso [INFIMUM]**

  Suponemos un juicio $V_0; C_0 \vdash C \cup \{\tau \Leftarrow \alpha, \tau' \Leftarrow \alpha\} \Longrightarrow C \cup \{\tau'' \Leftarrow \alpha\} \cup C'$, tal que $\tau \sqcap \tau' = \tau''$ **when** C' y $ftv(\tau) \cap ftv(\tau') = \emptyset$, y una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash (C \cup \{\tau \Leftarrow \alpha, \tau' \Leftarrow \alpha\})$. Por la semántica de los conjuntos de restricciones obtendremos que $\pi \vDash C$, $\pi \vDash \tau \Leftarrow \alpha$, $\pi \vDash \tau' \Leftarrow \alpha$ y $\pi(\alpha) = \{v_1, \ldots, v_n\}$, que siguiendo la semántica de las restricciones de encaje, para cada $v_i \in \pi(\alpha)$, tendremos que:

  $$\pi = \bigcup_{i=1}^{n} \pi_i \quad (v_i, \pi_i) \in \mathcal{T}[\![\tau]\!] \qquad \pi = \bigcup_{i=1}^{n} \pi'_i \quad (v_i, \pi'_i) \in \mathcal{T}[\![\tau']\!]$$

  Como sabemos que $ftv(\tau) \cap ftv(\tau') = \emptyset$, podemos definir una instanciación $\pi^\star_i$, para cada $i \in \{1..n\}$, de la siguiente manera:

  $$\forall \alpha \in \textbf{TypeVar}.\ \pi^\star_i(\alpha) = \begin{cases} \pi_i(\alpha) & \text{si } \alpha \in ftv(\tau) \\ \pi'_i(\alpha) & \text{e.o.c.} \end{cases}$$

  Por lo tanto, tendremos que $(v_i, \pi^\star_i) \in \mathcal{T}[\![\tau]\!]$ y $(v_i, \pi^\star_i) \in \mathcal{T}[\![\tau']\!]$, así como $\pi = \bigcup_{i=1}^{n} \pi^\star_i$. Aplicando ahora la propiedad 30, obtendremos que para todo $i \in \{1..n\}$ existe un π°_i tal que $(v_i, \pi^\circ_i) \in \mathcal{T}[\![\tau \sqcap \tau']\!]$ y $\pi^\circ_i \equiv \pi^\star_i$ (módulo $\setminus X'$), donde X' es el conjunto de variables de tipo frescas generadas por $\tau \sqcap \tau'$. Si definimos la instanciación $\pi^\circ = \bigcup_{i=1}^{n} \pi^\circ_i$, entonces se cumple que $\pi^\circ \vDash \tau \sqcap \tau' \Leftarrow \alpha$. Como tenemos que $\pi^\circ = \bigcup_{i=1}^{n} \pi^\circ_i$ y $\pi = \bigcup_{i=1}^{n} \pi^\star_i$, tendremos que $\pi^\circ \equiv \pi$ (módulo $\setminus X'$), por lo tanto obtenemos que $\pi^\circ \vDash C$. En la regla suponemos que $\tau \sqcap \tau'$ tiene la forma τ'' **when** C', por lo que se cumple que $\pi^\circ \vDash \tau''$ **when** $C' \Leftarrow \alpha$, con lo que aplicando la propiedad 32 obtendremos que $\pi^\circ \vDash \{\tau'' \Leftarrow \alpha\} \cup C'$, que junto a $\pi^\circ \vDash C$ lo podemos transformar de modo que:

  $$\pi^\circ \vDash (C \cup \{\tau'' \Leftarrow \alpha\} \cup C') \qquad \wedge \qquad \pi^\circ \equiv \pi \text{ (módulo } \setminus X')$$

  Por lo que $\pi' = \pi^\circ$ y $X = X'$, demostrando así el teorema para esta regla.

- **Caso [INFIMUM-X]**

  Suponemos un juicio $V_0; C_0 \vdash C \cup \left\{\tau_f \Leftarrow \alpha, \left(\overline{\beta_i}^{\,n}\right) \to \beta \Leftarrow \alpha\right\} \Longrightarrow \overline{C \cup \{\tau_f \Leftarrow \alpha\} \cup C'_j}^{\,m}$, donde $\tau_f = \bigsqcup_{j=1}^{m} \left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau'_{j,i}}^{\,n}\right) \to \tau'_j\right)$, $ftv(\tau_f) \cap ftv\left(\left(\overline{\beta_i}^{\,n}\right) \to \beta\right) = \emptyset$ y además, para cada $j \in \{1..m\}$, tenemos $\mu_j = \left[\overline{\alpha_{j,k}/\delta_{j,k}}\right]$, donde $\overline{\delta_{j,k}}$ son variables frescas, junto a $C'_j = \left\{\overline{\tau'_{j,i}\mu_j \Leftarrow \beta_i}^{\,n}, \tau'_j\mu_j \Leftarrow \beta\right\}$, y también suponemos una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash \left(C \cup \left\{\tau_f \Leftarrow \alpha, \left(\overline{\beta_i}^{\,n}\right) \to \beta \Leftarrow \alpha\right\}\right)$. Por la semántica de los conjuntos de restricciones obtenemos que $\pi \vDash C$, $\pi \vDash \tau_f \Leftarrow \alpha$ y $\pi \vDash \left(\overline{\beta_i}^{\,n}\right) \to \beta \Leftarrow \alpha$. Entonces, suponemos que cada $v \in \pi(\alpha)$ es un grafo de función, para que se puedan cumplir ambas restricciones, por lo que para cada v tendremos que:

  $$\pi = \bigcup_{v \in \pi(\alpha)} \pi_v \quad (v, \pi_v) \in \mathcal{T}\left[\!\!\left[\bigsqcup_{j=1}^{m} \left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau'_{j,i}}^{\,n}\right) \to \tau'_j\right)\right]\!\!\right]$$
  $$\pi = \bigcup_{v \in \pi(\alpha)} \pi'_v \quad (v, \pi'_v) \in \mathcal{T}\left[\!\!\left[\left(\overline{\beta_i}^{\,n}\right) \to \beta\right]\!\!\right]$$

Como sabemos que $ftv(\tau_f) \cap ftv\left(\left(\overline{\beta_i}^n\right) \to \beta\right) = \emptyset$, podemos definir una instanciación $\pi_v^\star$, para cada $v \in \pi(\alpha)$, de la siguiente manera:

$$\forall \alpha \in \mathbf{TypeVar}.\, \pi_v^\star(\alpha) = \begin{cases} \pi_v(\alpha) & \text{si } \alpha \in ftv(\tau_f) \\ \pi'_v(\alpha) & \text{e.o.c.} \end{cases}$$

Por lo tanto, tendremos que $\pi = \bigcup_{v \in \pi(\alpha)} \pi_v^\star$ y que para toda $v \in \pi(\alpha)$:

$$\begin{array}{c} (v, \pi_v^\star) \in \mathcal{T}\left[\!\left[\bigsqcup_{j=1}^m \left(\forall \overline{\alpha_{j,k}}.\left(\overline{\tau'_{j,i}}^n\right) \to \tau'_j\right)\right]\!\right] \quad \wedge \quad (v, \pi_v^\star) \in \mathcal{T}\left[\!\left[\left(\overline{\beta_i}^n\right) \to \beta\right]\!\right] \\ \Updownarrow \\ (v, \pi_v^\star) \in \bigcup_{j=1}^m \mathcal{T}\left[\!\left[\forall \overline{\alpha_{j,k}}.\left(\overline{\tau'_{j,i}}^n\right) \to \tau'_j\right]\!\right] \quad \wedge \quad (v, \pi_v^\star) \in \mathcal{T}\left[\!\left[\left(\overline{\beta_i}^n\right) \to \beta\right]\!\right] \end{array}$$

Es decir, que $(v, \pi_v^\star) \in \mathcal{T}\left[\!\left[\forall \overline{\alpha_{j,k}}.\left(\overline{\tau'_{j,i}}^n\right) \to \tau'_j\right]\!\right]$ para algún $j \in \{1..m\}$. Entonces, suponiendo un $v \in \pi(\alpha)$ y una $j \in \{1..m\}$, tendremos la sustitución $\mu_j = \left[\overline{\alpha_{j,k}/\delta_{j,k}}\right]$, donde llamaremos X'_j al conjunto de variables de tipo nuevas que va a componer la X final, de modo que $X'_j = \left\{\overline{\delta_{j,k}}\right\}$ y $X = \bigcup_{j=1}^m X'_j$, por lo que tendremos una instanciación $\pi_v^\circ \equiv \pi_v^\star$ (módulo $\setminus \left\{\overline{\delta_{j,k}}\right\}$). Por la semántica de los esquemas de tipo funcional obtendremos que $\pi_v^\star \equiv \pi_v^\bullet$ (módulo $\setminus \{\overline{\alpha_{j,k}}\}$) para algún $\pi_v^\bullet$. De este modo, para cada $w \in v$, donde $w = \left(\left(v'_1, \ldots, v'_n\right), v'\right)$, podemos descomponer $\pi_v^\bullet = \bigcup_{w \in v} \pi_{v,w}^\bullet$ y $\pi_v^\star = \bigcup_{w \in v} \pi_{v,w}^\star$, obteniendo así que:

$$\begin{array}{c} \forall i \in \{1..n\}.(v'_i, \pi_{v,w}^\bullet) \in \mathcal{T}\left[\!\left[\tau'_{j,i}\right]\!\right] \quad \wedge \quad (v', \pi_{v,w}^\bullet) \in \mathcal{T}\left[\!\left[\tau'_j\right]\!\right] \\ \wedge \quad \forall i \in \{1..n\}.(v'_i, \pi_{v,w}^\star) \in \mathcal{T}\left[\!\left[\beta_i\right]\!\right] \quad \wedge \quad (v', \pi_{v,w}^\star) \in \mathcal{T}\left[\!\left[\beta\right]\!\right] \end{array}$$

Si descomponemos $\pi_v^\circ = \bigcup_{w \in v} \pi_{v,w}^\circ$ y aplicamos la sustitución, podremos obtener que:

$$\begin{array}{c} \forall i \in \{1..n\}.(v'_i, \pi_{v,w}^\circ) \in \mathcal{T}\left[\!\left[\tau'_{j,i}\mu_j\right]\!\right] \quad \wedge \quad (v', \pi_{v,w}^\circ) \in \mathcal{T}\left[\!\left[\tau'_j\mu_j\right]\!\right] \\ \wedge \quad \forall i \in \{1..n\}.(v'_i, \pi_{v,w}^\circ) \in \mathcal{T}\left[\!\left[\beta_i\right]\!\right] \quad \wedge \quad (v', \pi_{v,w}^\circ) \in \mathcal{T}\left[\!\left[\beta\right]\!\right] \end{array}$$

De forma análoga a la demostración de aplicar el ínfimo entre un tipo y una variable de tipo, obtendremos que:

$$\forall i \in \{1..n\}.\pi_{v,w}^\circ \vDash \tau'_{j,i}\mu_j \Leftarrow \beta_i \quad \wedge \quad \pi_{v,w}^\circ \vDash \tau'_j\mu_j \Leftarrow \beta$$

Esto último equivale a $\pi_{v,w}^\circ \vDash C_j$. Aplicando el lema 14, tendremos que $\pi_v^\circ \vDash C_j$. Si definimos $\pi^\circ = \bigcup_{v \in \pi(\alpha)} \pi_v^\circ$, aplicando de nuevo el lema 14, tendremos que $\pi^\circ \vDash C_j$. Con la instanciación π° sabemos que se cumple $\pi^\circ \equiv \pi$ (módulo $\setminus \bigcup_{j=1}^m \left\{\overline{\delta_{j,k}}\right\}$), por lo tanto $\pi \vDash C$ y $\pi \vDash \tau_f \Leftarrow \alpha$ cumplen también que $\pi^\circ \vDash C$ y $\pi^\circ \vDash \tau_f \Leftarrow \alpha$, que junto a $\pi^\circ \vDash C_j$ obtendremos que:

$$\pi^\circ \vDash \left(C \cup \{\tau_f \Leftarrow \alpha\} \cup C_j\right)$$

donde $\pi' = \pi^\circ$ y $X = \bigcup_{j=1}^{m} X'_j$, cumpliendo así el teorema para esta regla.

- **Caso [UPGRADE]**

  Suponemos un juicio $V_0; C_0 \vdash C \cup \{\alpha \subseteq \tau\} \Longrightarrow C \cup \{\tau \Leftarrow \alpha\}$, donde $vars(\tau) = \emptyset$ porque τ es un tipo *ground*, y una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash (C \cup \{\alpha \subseteq \tau\})$. Por la semántica de los conjuntos de restricciones sabemos que se cumple que $\pi \vDash C$ y $\pi \vDash \{\alpha \subseteq \tau\}$. Entonces, con $\pi \vDash \alpha \subseteq \tau$, siguiendo su semántica, tenemos lo siguiente:

  $$\pi(\alpha) \subseteq \left\{ v \mid (v, \pi'') \in \mathcal{T}[\![\tau]\!], \pi'' \subseteq \pi \right\}$$

  Como τ es un tipo *ground*, se cumple que $(v, \pi'') \in \mathcal{T}[\![\tau]\!]$ para todo π''. Por lo tanto, para cada $v \in \pi(\alpha)$, existe una instanciación $\pi_v \subseteq \pi$ tal que $(v, \pi_v) \in \mathcal{T}[\![\tau]\!]$, en concreto donde $\pi_v(\alpha) = \{v\}$. Como $\pi_v \subseteq \pi$, para cada $v \in \pi(\alpha)$, entonces $\pi = \bigcup_{v \in \pi(\alpha)} \pi_v$. Entonces, tenemos que:

  $$\exists (\pi_v)_{v \in \pi(\alpha)} : \pi = \bigcup_{v \in \pi(\alpha)} \pi_v \wedge (\forall v \in \pi(\alpha) : (v, \pi_v) \in \mathcal{T}[\![\tau]\!])$$

  Por lo tanto, obtenemos que se cumple $\pi \vDash \tau \Leftarrow \alpha$, que junto a $\pi \vDash C$, por la semántica de los conjuntos de restricciones, obtendríamos que $\pi \vDash (C \cup \tau \Leftarrow \alpha)$, y como $\pi' = \pi$, porque $X = \emptyset$, se cumple así el teorema para esta regla.

- **Caso [SWAP]**

  Suponemos un juicio $V_0; C_0 \vdash C \cup \{\beta \Leftarrow \alpha\} \Longrightarrow C \cup \{\alpha \Leftarrow \beta\}$ y una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash \left(C \cup \{\beta \Leftarrow \alpha\}\right)$. Por la definición semántica de los conjuntos de restricciones tenemos que $\pi \vDash C$ y $\pi \vDash \beta \Leftarrow \alpha$. Con la definición semántica de las restricciones de encaje sabemos que $\pi \vDash \beta \Leftarrow \alpha$ implica que $\pi(\alpha) = \pi(\beta)$, que a su vez implica que $\pi \vDash \alpha \Leftarrow \beta$. Por último, obtendremos que $\pi \vDash \left(C \cup \{\alpha \Leftarrow \beta\}\right)$, y como $\pi' = \pi$, porque $X = \emptyset$, se demuestra así el teorema para esta regla.

- **Caso [ADD-ANY]**

  Suponemos un juicio $V_0; C_0 \vdash C \Longrightarrow C \cup \{\mathsf{any}() \Leftarrow \beta\}$, para alguna variable de tipo β, y una instanciación π. Sabemos que $\pi \vDash C_0$ y $\pi \vDash C$. Suponiendo que β corresponde a una variable de tipo no fresca. Suponemos ahora la restricción de encaje $\mathsf{any}() \Leftarrow \beta$, sabemos que esta se cumple para cualquier instanciación de forma trivial, por lo tanto se cumple que $\pi \vDash \{\mathsf{any}() \Leftarrow \beta\}$. Entonces con $\pi \vDash C$ y $\pi \vDash \{\mathsf{any}() \Leftarrow \beta\}$, por la definición semántica de los conjuntos de las restricciones, tendremos que $\pi \vDash \left(C \cup \{\mathsf{any}() \Leftarrow \beta\}\right)$. En este caso tanto al ser $X = \emptyset$, tendríamos que $\pi' = \pi$

  Sin embargo, si suponemos que β es una variable fresca, entonces $X = \{\beta\}$, pero la instanciación π' cumpliría que $\pi' = \pi$, porque la restricción $\mathsf{any}() \Leftarrow \beta$ hace que se cumpla trivialmente cualquier π posible, ya que siempre se cumple que $\pi(\alpha) \subseteq \mathbf{DVal}$, para cualquier variable de tipo α.

- **Caso [SPLIT]**

  Suponemos un juicio $V_0; C_0 \vdash \tau \cup \tau' \Longrightarrow \tau, \tau'$ y una instanciación π. Sabemos que $\pi \vDash C_0$ y $(v, \pi) \in [\![\tau \cup \tau']\!]$. Por la semántica de los tipos unión obtenemos:

  $$(v, \pi) \in \left([\![\tau]\!] \setminus \left(itv\left(\tau'\right) \setminus ftv(\tau)\right) \cup [\![\tau']\!] \setminus \left(itv(\tau) \setminus ftv\left(\tau'\right)\right)\right)$$

  Con esta definición tendremos los siguientes dos casos:

  - Si $(v, \pi) \in [\![\tau]\!] \setminus \left(itv\left(\tau'\right) \setminus ftv(\tau)\right)$, entonces $(v, \pi) \in [\![\tau]\!]$, porque $[\![\tau]\!] \setminus \left(itv\left(\tau'\right) \setminus ftv(\tau)\right) \subseteq [\![\tau]\!]$.
  - Si $(v, \pi) \in [\![\tau']\!] \setminus \left(itv(\tau) \setminus ftv\left(\tau'\right)\right)$, entonces $(v, \pi) \in [\![\tau']\!]$, porque $[\![\tau']\!] \setminus \left(itv(\tau) \setminus ftv\left(\tau'\right)\right) \subseteq [\![\tau']\!]$.

  Tomando $(v, \pi) \in [\![\tau]\!]$ y $(v, \pi) \in [\![\tau']\!]$, obtenemos $(v, \pi) \in \left([\![\tau]\!] \cup [\![\tau']\!]\right)$, que junto a $\pi \vDash C_0$ se demuestra el teorema para esta regla, porque como $X = \emptyset$ tenemos que $\pi' = \pi$.

- **Caso [REPLACE]**

  Suponemos un juicio $V_0; C_0 \vdash \tau \Longrightarrow \tau\left[\alpha/\tau'\right]$, tal que $C_0 = C \cup \left\{\tau' \Leftarrow \alpha\right\}$ y $\alpha \notin V_0 \cup ftv(C)$, y también una instanciación π. Sabemos que $\pi \vDash C_0$ y $(v, \pi) \in [\![\tau]\!]$. Como $C_0 = C \cup \left\{\tau' \Leftarrow \alpha\right\}$, tendremos que $\pi \vDash C$ y $\pi \vDash \tau' \Leftarrow \alpha$. Por lo tanto, tomando $(v, \pi) \in [\![\tau]\!]$ y $\pi \vDash \tau' \Leftarrow \alpha$, aplicaremos el lema 11 para obtener $(v, \pi) \in [\![\tau\left[\alpha/\tau'\right]]\!]$, y como $\pi' = \pi$, porque $X = \emptyset$, se demuestra así el teorema para esta regla.

- **Caso [SCHEME]**

  Suponemos un juicio $V_0; C_0 \vdash \forall \overline{\alpha_i}.\tau \Longrightarrow \forall \overline{\beta_{1,j}}.\tau'_1, \ldots, \forall \overline{\beta_{m,j}}.\tau'_m$, donde $\overline{\alpha_i} \nsubseteq V_0$, $V_0; C_0 \vdash \tau \Longrightarrow \tau'_1, \ldots, \tau'_m$ y $\left\{\overline{\beta_{k,j}}\right\} = \left\{\overline{\alpha_i}\right\} \cup \left(ftv\left(\tau'_k\right) \setminus ftv(\tau)\right)$, para cada $k \in \{1..m\}$, y una instanciación π. Aunque estamos manejando esquemas de tipo funcional, como abuso de notación usaremos τ en vez de la forma de tipo funcional para simplificar la regla y la demostración. Sabemos que $\pi \vDash C_0$ y $(v, \pi) \in \mathscr{S}\,[\![\forall \overline{\alpha_i}.\tau]\!]$. Por la semántica de los esquemas de tipo tenemos $(v, \pi^\circ) \in \mathscr{F}\,[\![\tau]\!]$ y $\pi^\circ \equiv \pi$ (módulo $\setminus \left\{\overline{\alpha_i}\right\}$). Como tenemos que $\overline{\alpha_i} \nsubseteq V_0$ y que $ftv(C_0) \subseteq V_0$, sabemos que las instancias para cada α_i en π son irrelevantes porque no aparecen libres en el contexto global del tipo, por lo tanto $\pi^\circ \vDash C_0$. Tenemos que $(v, \pi^\circ) \in \mathscr{F}\,[\![\tau]\!]$ se puede expresar también como $(v, \pi^\circ) \in \mathscr{S}\,[\![\tau]\!]$. Por hipótesis de inducción, con $\pi^\circ \vDash C_0$ y $(v, \pi^\circ) \in \mathscr{S}\,[\![\tau]\!]$, obtendremos $(v, \pi^\bullet) \in \mathscr{S}\,[\![\tau'_k]\!]$, para alguna $k \in \{1..m\}$, y $\pi^\bullet \equiv \pi^\circ$ (módulo $\setminus X'$), donde X' es la unión de las variables de tipo libres frescas de cada τ'_k. Con $\left\{\overline{\beta'_{k,j}}\right\} = \left(ftv\left(\tau'_k\right) \setminus ftv(\tau)\right)$, para cada $k \in \{1..m\}$, tenemos que $X' = \bigcup_{k=1}^{m} \left\{\overline{\beta'_{k,j}}\right\}$, de este modo podemos definir la instanciación $\pi^\star_k$ de la siguiente manera:

  $$\forall \alpha \in \mathbf{TypeVar}.\, \pi^\star_k(\alpha) = \begin{cases} \pi^\bullet(\alpha) & \text{si } \alpha \in \left\{\overline{\beta'_{k,j}}\right\} \\ \pi^\circ(\alpha) & \text{si } \alpha \in \left\{\overline{\alpha_i}\right\} \\ \pi(\alpha) & \text{e.o.c.} \end{cases}$$

  Por lo tanto, $\pi^\star_k \equiv \pi^\bullet$ (módulo $\setminus \left\{\overline{\beta'_{k,j}}\right\}$), para cada $k \in \{1..m\}$, y tendremos también que $\left(v, \pi^\star_k\right) \in$

$\mathscr{S}[\![\tau'_k]\!]$, que equivale a $(v,\pi^\star_k) \in \mathscr{F}[\![\tau'_k]\!]$. Con $\left\{\overline{\beta_{k,j}}\right\} = \{\overline{\alpha_i}\} \cup \left\{\overline{\beta'_{k,j}}\right\}$, para cada $k \in \{1..m\}$, tenemos todas las variables de tipo que queremos cerrar, así teniendo que $\pi^\star_k \equiv \pi$ (módulo $\setminus \left\{\overline{\beta_{k,j}}\right\}$), que junto a $(v,\pi^\star_k) \in \mathscr{F}[\![\tau'_k]\!]$ obtendremos que $(v,\pi) \in \mathscr{S}\left[\!\left[\forall\overline{\beta_{k,j}}.\tau'_k\right]\!\right]$, y como $\pi' = \pi$, porque $X = \emptyset$, queda demostrado así el teorema para esta regla.

El resto de reglas de transformación que quedan se demuestran de forma trivial aplicando la hipótesis por inducción y el hecho de que la semántica de un tipo anotado, restricción o tipo es monótona con respecto a la de los tipos que lo componen. □

5.6. Transformación de funciones recursivas

En la sección anterior vimos cómo podíamos transformar los tipos y los conjuntos de restricciones provenientes del análisis de funciones no recursivas para simplificarlos, pero no abordamos el tratamiento de funciones recursivas. Cuando se aplica el proceso de generación de restricciones a una función f recursiva, el resultado contiene una restricción de encaje $\tau \Leftarrow \alpha_f$ en el que τ contiene una restricción de aplicación simbólica que utiliza α_f como tipo de la función aplicada. Este tipo de restricciones no pueden simplificarse utilizando únicamente las reglas vistas anteriormente. Para poder simplificar τ tenemos que realizar una serie de iteraciones para calcular un punto fijo que represente el tipo funcional que corresponde a α_f. Para que el algoritmo sea terminante, suponemos que el número de iteraciones está limitado por una constante K. Si se supera este número de iteraciones, tendríamos que aplicar una función de *widening* que, mediante una heurística, nos devolviera un *success type* con el que comprobar que se ha alcanzado un punto fijo, o un supertipo de este.

El primer paso para iniciar la transformación del tipo de la función recursiva, es introducir una restricción de encaje que asigne a α_f un tipo funcional τ_0 en el conjunto C_0 que se va a usar en la regla [MATCH] para transformar el tipo τ. Si la función tiene n parámetros de entrada, el tipo funcional τ_0 que α_f tendrá en la primera iteración será $\left(\overline{\texttt{none()}}^n\right) \to \texttt{none()}$. Este tipo inicial de entrada que va a representar a α_f, nos servirá para calcular la sobrecarga correspondiente al caso base de la función analizada, que será inyectada en la siguiente iteración para volver a transformar τ. Es decir, si estamos en la iteración k, nuestro juicio de transformación tendrá la siguiente forma:

$$\forall k \geq 1. \qquad V_0; C_0 \cup \{\tau_{k-1} \Leftarrow \alpha_f\} \vdash \tau \Longrightarrow^* \tau_k$$

donde $\Longrightarrow^*$ representa el cierre reflexivo y transitivo de la relación $\Longrightarrow$. Lo que esta secuencia de pasos quiere decir, es que en C_0 estaremos añadiendo una restricción para α_f con el tipo obtenido en la iteración anterior, para transformar τ en la siguiente τ_k. Cada nueva iteración empieza siempre con τ, y no con τ_{k-1}, porque el tipo devuelto por la transformación carece de las aplicaciones simbólicas obtenidas por la generación inicial de restricciones en la inferencia, con lo que al haber borrado en τ_{k-1} las restricciones que tenían las aplicaciones simbólicas, no podríamos utilizar la regla [CALL-F] para introducir la nueva información relativa a aplicar la función f de tipo α_f.

Para cada tipo τ_k que obtenemos, al terminar la iteración actual de transformación, debemos comprobar si se ha alcanzado el punto fijo para el tipo de f. Usando la aproximación de relación de subtipado polimórfico, postulada en la sección 5.3.4, nos preguntamos si el tipo τ_k es un subtipo de τ_{k-1}. Si $\tau_k \overset{\circ}{\subseteq} \tau_{k-1}$ es cierta, el tipo final para f será τ_{k-1} y α_f encajará con este tipo en el resultado final de nuestra inferencia. Si no es cierta la comparación y hemos superado el número máximo K de iteraciones, tendremos que aplicar a τ_k la función de *widening* para obtener el punto fijo.

5.6.1. Aplicando la función de widening

El proceso de *widening* es también iterativo y recibe un tipo τ_k, que es el último tipo obtenido al exceder el número máximo de iteraciones que teníamos para ello. A este tipo se le aplica la siguiente regla:

$$[\text{WIDENING}]\ \frac{height(\tau) = h \qquad h > 0}{V_0; C_0 \vdash \tau \Longrightarrow cut(\tau, h-1)}$$

Con esta regla obtenemos un tipo τ'_k, que es el resultado de haber recortado la altura del árbol sintáctico que representa el tipo τ_k. Aunque lo vamos a explicar luego con más detalle, conviene introducir aquí qué se entiende por recortar la altura de τ_k. Supongamos que h es la altura del tipo τ_k, recortar τ_k en una unidad consistiría en la sustitución de todos los nodos del árbol abstracto que estén a profundidad $h-1$ por el tipo `any()`, obteniendo así un tipo τ'_k de altura $h-1$ como resultado, que es un supertipo de τ_k. El hecho de que τ'_k sea un supertipo del anterior, no significa que sea una aproximación superior del punto fijo del proceso iterativo. Así que necesitamos iterar de nuevo en la transformación, obteniendo lo siguiente:

$$V_0; C_0 \cup \{\tau'_k \Leftarrow \alpha_f\} \vdash \tau \Longrightarrow^* \tau_{k+1}$$

Usamos τ'_k como tipo funcional para α_f en la transformación, recibiendo como resultado un τ_{k+1}. De nuevo, como hacíamos en la iteración inicial para calcular el punto fijo, tendremos que comprobar si $\tau_{k+1} \overset{\circ}{\subseteq} \tau'_k$, porque si no lo fuera tendremos que recortar la altura de τ_{k+1} para volver a iterar hasta alcanzar el punto fijo.

En la regla [WIDENING] hay dos funciones que todavía no hemos explicado. La función *height* nos calcula la altura que tiene el árbol sintáctico de un tipo τ dado como parámetro. La función *cut* recibe un tipo τ y una altura arbitraria, y nos devuelve un nuevo tipo τ' sustituyendo por `any()` todos los componentes de τ que se encuentren a una profundidad mayor o igual que h en el árbol abstracto.

La definición completa de *height* es la siguiente:

$$height(\tau) = \begin{cases} max\left(\overline{height(\tau_i)}^n\right) + 1 & \tau = \{\overline{\tau_i}^n\} \wedge n > 0 \\ max\left(height(\tau_1), height(\tau_2)\right) + 1 & \tau = \texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)} \\ max\left(\overline{height(\tau_i)}^n, \tau'\right) + 1 & \tau = \left(\overline{\tau_i}^n\right) \rightarrow \tau' \\ height\left(\tau'\right) & \tau = \forall\overline{\alpha_i}.\tau' \\ max\left(height(\sigma_1), \ldots, height(\sigma_n)\right) & \tau = \bigsqcup_{i=1}^{n} \sigma_i \\ max\left(height(\tau_1), height(\tau_2)\right) & \tau = \tau_1 \cup \tau_2 \\ max\left(height\left(\tau'\right), \overline{height\left(\varphi_i\right)}^n\right) & \tau = \tau' \textbf{when } \{\overline{\varphi_i}^n\} \\ 0 & \text{e.o.c.} \end{cases}$$

$$height\left(\varphi\right) = \begin{cases} height(\tau) & \varphi = c \subseteq \tau \\ max\left(height(\alpha), height(\tau)\right) & \varphi = \alpha \subseteq \tau \\ max\left(height(\alpha), height(\tau)\right) & \varphi = \tau \Leftarrow \alpha \\ 0 & \text{e.o.c.} \end{cases}$$

En realidad, como se puede observar, son dos funciones porque una acepta tipos y la otra restricciones. Las dos funciones hacen un recorrido en profundidad del árbol sintáctico abstracto del tipo de entrada, para obtener su altura máxima. Solo se aumenta la altura cuando se trata de un tipo de estructura de datos (tuplas, listas no vacías y funciones), ya que—durante la transformación de tipos funcionales recursivos—el crecimiento en altura depende directamente de estos tipos que representan estructuras de datos de programa, frente a otros tipos del árbol sintáctico que crecen a lo ancho, como por ejemplo el caso de las uniones de tipos.

La definición completa de *cut* es la siguiente:

$$cut(\tau, h) = \begin{cases} \texttt{any()} & h < 1 \\ \{\overline{cut(\tau_i, h-1)}^n\} & \tau = \{\overline{\tau_i}^n\} \\ \texttt{nelist(}cut(\tau_1, h-1)\texttt{,}cut(\tau_2, h-1)\texttt{)} & \tau = \texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)} \\ \left(\overline{cut(\tau_i, h-1)}^n\right) \rightarrow cut\left(\tau' h-1\right) & \tau = \left(\overline{\tau_i}^n\right) \rightarrow \tau' \\ \forall\overline{\alpha_i}.cut\left(\tau', h\right) & \tau = \forall\overline{\alpha_i}.\tau' \\ \bigsqcup_{i=1}^{n} cut(\sigma_i, h) & \tau = \bigsqcup_{i=1}^{n} \sigma_i \\ cut(\tau_1, h) \cup cut(\tau_2, h) & \tau = \tau_1 \cup \tau_2 \\ cut\left(\tau', h\right) \textbf{when } \left\{\overline{cut\left(\varphi_i, h\right)}^n\right\} & \tau = \tau' \textbf{when } \{\overline{\varphi_i}^n\} \\ \tau & \text{e.o.c.} \end{cases}$$

$$cut(\varphi, h) = \begin{cases} c \subseteq cut(\tau, h) & \varphi = c \subseteq \tau \\ \alpha \subseteq cut(\tau, h) & \varphi = \alpha \subseteq \tau \\ cut(\tau, h) \Leftarrow \alpha & \varphi = \tau \Leftarrow \alpha \\ \varphi & \text{e.o.c.} \end{cases}$$

De nuevo, tenemos dos funciones, una para tipos y otra para restricciones, que realizan un recorrido en profundidad del tipo. Aquí se va restando una unidad al parámetro de la altura, cuando entramos en un tipo tupla, lista no vacía o funcional, a medida que vamos recorriendo el tipo. Si la altura es menor o igual que cero, se devuelve el tipo `any()`. De lo contrario se sigue con el recorrido.

El proceso iterativo que usa nuestra estrategia de *widening*, en el peor de los casos, nos daría para una función f de n parámetros el tipo funcional $\left(\overline{\texttt{any()}}^n\right) \rightarrow \texttt{any()}$ como *success type*, que engloba a todas las posibles funciones de n parámetros que existen en nuestro lenguaje. Este tipo es, con toda seguridad, una aproximación superior al punto fijo buscado en el proceso iterativo descrito en esta sección.

5.7. Ejemplos de inferencia de tipos polimórficos

En esta sección mostraremos paso a paso cómo se puede inferir un tipo para una función recursiva y también analizaremos los tipos que hemos inferido para diversas funciones usando nuestra herramienta que implementa el algoritmo de inferencia.

5.7.1. Infiriendo un tipo para la función `map`

En la sección 4.6.2 vimos cómo derivar un *success type* polimórfico para la función `Map` cuyo código en *Mini Erlang* es el siguiente:

```
letrec Map = fun(F,L) → case L of
    [] when 'true' → []
    [X|XS] when 'true' → [F(X)|Map(F,XS)]
end in Map
```

El tipo que obteníamos en la derivación era:

$$\begin{array}{l} (\texttt{any()},\texttt{[]}) \rightarrow \texttt{[]} \sqcup \\ \forall \alpha, \alpha', \beta, \beta'.((\alpha) \rightarrow \beta, \texttt{nelist(}\alpha'\texttt{,[])}) \rightarrow \texttt{nelist(}\beta'\texttt{,[])} \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta \end{array}$$

En esta sección vamos a ver cómo podemos inferir este tipo con nuestras reglas de generación y transformación. Empezaremos mostrando primero el tipo obtenido con la generación de restricciones. Normalmente supondríamos tener un entorno Γ_0 que contendría los tipos de todas las funciones de

(1) [APP] $\texttt{F(X)} \vdash \langle \delta_0; \{ Z_{\alpha_\texttt{F}}(\alpha_\texttt{X}) \Leftarrow \delta_0 \} \rangle$

(2) [APP] $\texttt{Map(F,XS)} \vdash \langle \delta_1; \{ Z_{\alpha_\texttt{Map}}(\alpha_\texttt{F}, \alpha_\texttt{XS}) \Leftarrow \delta_1 \} \rangle$

(3) [LST] $\texttt{[F(X)|Map(F,XS)]} \vdash \langle \texttt{nelist}(\delta_0, \delta_1); \{ \alpha_\texttt{F}(\alpha_\texttt{X}) \Leftarrow \delta_0, Z_{\alpha_\texttt{Map}}(\alpha_\texttt{F}, \alpha_\texttt{XS}) \Leftarrow \delta_1 \} \rangle$

(4) [LST] $\texttt{[X|XS]} \vdash \langle \texttt{nelist}(\alpha_\texttt{X}, \alpha_\texttt{XS}); \emptyset \rangle$

(5) [CLS] $\texttt{[X|XS]}\ \textbf{when}\ \texttt{'true'} \rightarrow \texttt{[F(X)|Map(F,XS)]} \vdash_{\alpha_\texttt{L}} \langle \texttt{nelist}(\delta_0, \delta_1); C \rangle$
donde $C = \{ \alpha_\texttt{F}(\delta_2) \Leftarrow \delta_0, Z_{\alpha_\texttt{Map}}(\alpha_\texttt{F}, \delta_3) \Leftarrow \delta_1, \texttt{nelist}(\delta_2, \delta_3) \Leftarrow \alpha_\texttt{L}, \texttt{'true'} \subseteq \texttt{'true'} \}$

(6) [CLS] $\texttt{[]}\ \textbf{when}\ \texttt{'true'} \rightarrow \texttt{[]} \vdash_{\alpha_\texttt{L}} \langle \texttt{[]}; \{ \texttt{[]} \Leftarrow \alpha_\texttt{L}, \texttt{'true'} \subseteq \texttt{'true'} \} \rangle$

(7) [CAS] $\textbf{case}\ \texttt{L}\ \textbf{of}\ \cdots\ \textbf{end} \vdash \langle \texttt{[]}; \{ \texttt{[]} \Leftarrow \alpha_\texttt{L}, \texttt{'true'} \subseteq \texttt{'true'} \} \rangle ; \langle \texttt{nelist}(\delta_0, \delta_1); C \rangle$
donde $C = \{ \alpha_\texttt{F}(\delta_2) \Leftarrow \delta_0, Z_{\alpha_\texttt{Map}}(\alpha_\texttt{F}, \delta_3) \Leftarrow \delta_1, \texttt{nelist}(\delta_2, \delta_3) \Leftarrow \alpha_\texttt{L}, \texttt{'true'} \subseteq \texttt{'true'} \}$

(8) [ABS] $\textbf{fun}(\texttt{F}, \texttt{L}) \rightarrow \cdots \vdash \langle \sigma_0 \sqcup \sigma_1; \emptyset \rangle$
donde $\sigma_0 = \forall \alpha_\texttt{F}, \alpha_\texttt{L}. (\alpha_\texttt{F}, \alpha_\texttt{L}) \rightarrow \texttt{[]}\ \textbf{when}\ \texttt{[]} \Leftarrow \alpha_\texttt{L}, \texttt{'true'} \subseteq \texttt{'true'}$
$\sigma_1 = \forall \alpha_\texttt{F}, \alpha_\texttt{L}, \delta_0, \delta_1, \delta_2, \delta_3. (\alpha_\texttt{F}, \alpha_\texttt{L}) \rightarrow \texttt{nelist}(\delta_0, \delta_1)\ \textbf{when}\ C$
$C = \{ \alpha_\texttt{F}(\delta_2) \Leftarrow \delta_0, Z_{\alpha_\texttt{Map}}(\alpha_\texttt{F}, \delta_3) \Leftarrow \delta_1, \texttt{nelist}(\delta_2, \delta_3) \Leftarrow \alpha_\texttt{L}, \texttt{'true'} \subseteq \texttt{'true'} \}$

(9) [LRC] $\textbf{letrec}\ \texttt{Map} = \cdots\ \textbf{in}\ \texttt{Map} \vdash \langle \delta_4; \{ \sigma_0 \sqcup \sigma_1 \Leftarrow \delta_4 \} \rangle$
donde $\sigma_0 = \forall \alpha_\texttt{F}, \alpha_\texttt{L}. (\alpha_\texttt{F}, \alpha_\texttt{L}) \rightarrow \texttt{[]}\ \textbf{when}\ \texttt{[]} \Leftarrow \alpha_\texttt{L}, \texttt{'true'} \subseteq \texttt{'true'}$
$\sigma_1 = \forall \alpha_\texttt{F}, \alpha_\texttt{L}, \delta_0, \delta_1, \delta_2, \delta_3. (\alpha_\texttt{F}, \alpha_\texttt{L}) \rightarrow \texttt{nelist}(\delta_0, \delta_1)\ \textbf{when}\ C$
$C = \{ \alpha_\texttt{F}(\delta_2) \Leftarrow \delta_0, Z_{\delta_4}(\alpha_\texttt{F}, \delta_3) \Leftarrow \delta_1, \texttt{nelist}(\delta_2, \delta_3) \Leftarrow \alpha_\texttt{L}, \texttt{'true'} \subseteq \texttt{'true'} \}$

Figura 5.7: Generación de restricciones para la función `Map`

la biblioteca estándar de *Erlang* u otros módulos que hubiéramos analizado previamente, pero para este ejemplo nos basta con el entorno vacío, puesto que no vamos a usar ninguna función externa al propio código del ejemplo. En la figura 5.7 vemos las reglas utilizadas en el procedimiento de la generación de restricciones, aunque el uso de las reglas [CNS] y [VAR] se ha obviado porque su aplicación es trivial. Aplicando un renombramiento de variables, obtenemos lo siguiente como resultado para la expresión **letrec**:

$$\begin{aligned} &\langle \alpha_{map}; \{ \sigma_0 \sqcup \sigma_1 \Leftarrow \alpha_{map} \} \rangle \\ &\text{donde } \sigma_0 = \forall \alpha_f, \alpha_l. (\alpha_f, \alpha_l) \rightarrow \texttt{[]}\ \textbf{when}\ \texttt{[]} \Leftarrow \alpha_l, \texttt{'true'} \subseteq \texttt{'true'} \\ &\qquad \sigma_1 = \forall \alpha_f, \alpha_l, \alpha_a, \alpha_b, \alpha_x, \alpha_{xs}. (\alpha_f, \alpha_l) \rightarrow \texttt{nelist}(\alpha_a, \alpha_b)\ \textbf{when}\ C \\ &\qquad C = \left\{ Z_{\alpha_f}(\alpha_x) \Leftarrow \alpha_a, Z_{\alpha_{map}}(\alpha_f, \alpha_{xs}) \Leftarrow \alpha_b, \texttt{nelist}(\alpha_x, \alpha_{xs}) \Leftarrow \alpha_l, \texttt{'true'} \subseteq \texttt{'true'} \right\} \end{aligned} \tag{5.9}$$

Al salir del **letrec**, la variable `Map` deja de ser libre y por ello perdemos la variable de tipo $\alpha_\texttt{Map}$, siendo sustituida por la variable fresca δ_4 en el juicio donde se aplica la regla de generación [LRC]. Sin embargo, al renombrar hemos escogido α_{map} para que se pueda entender que dicha variable de tipo representa a la función asignada a `Map`, sin volver a usar $\alpha_\texttt{Map}$ porque ya está siendo usada en el entorno de tipos Γ asociado al par y que por simplificar con la notación se omite, ya que no es información realmente relevante.

Dentro de las reglas aplicadas en la figura 5.7, podemos ver en (1) y (2) cómo se generan las aplicaciones simbólicas con la regla [APP], generando variables frescas para asociar el resultado de la aplicación a cada aplicación simbólica. Con la regla [LST] en (3) podemos ver cómo tomando los juicios (1) y (2), donde se utiliza [APP], componemos una lista no vacía y unimos los conjuntos de restricciones en uno solo. En las reglas [CLS] de (5) y (6), como el tipo de la guarda es el átomo `'true'`, vemos que se ha generado una restricción $\texttt{'true'} \subseteq \texttt{'true'}$, que luego veremos cómo se elimina en la transformación. En (5), al aplicar la regla [CLS], la variable de tipo α_{L}, que representa el tipo del discriminante, encaja con el tipo del patrón obtenido en el juicio (4), y como resultado devolvemos el tipo obtenido en (3) añadiendo las nuevas restricciones. En la aplicación de la regla [CLS] en (6), α_{L} encaja con el tipo literal `[]` y se devuelve también el literal `[]` junto a las restricciones creadas para la guarda y el patrón. Con la regla [CAS] en (7) nos limitamos a unir en una secuencia de tipos anotados los resultados de (5) y (6). Luego, en (8), con la regla [ABS] creamos un esquema de tipo funcional sobrecargado, donde se aplica el cierre a toda variable de tipo que no represente una variable libre de programa en la función. Es decir, se aplica el cierre a todas excepto $\alpha_{\texttt{Map}}$. Por último terminamos con (9), aplicando [LRC], que nos devuelve el tipo para `Map`, dejando todas las variables cerradas en la expresión.

La siguiente etapa de la inferencia consiste en tomar el tipo generado (5.9) y transformarlo para eliminar las aplicaciones simbólicas y simplificar el resultado final. Dado que el tipo está ya normalizado, pasaremos a recorrer el tipo para transformarlo, suponiendo que V_0 y C_0 son conjuntos vacíos. Como solo tenemos un único par no vamos a usar la regla [PAIRS], así que directamente usaremos la regla [PAIR-C] para transformar las restricciones, añadiendo a V_0 la variable de tipo α_{map}. Como no hay ninguna restricción de subconjunto, no se van a aplicar las reglas [CHECK-D] o [CHECK-C]. De hecho solo hay una única restricción de encaje de la variable de tipo α_{map} con un esquema de tipo funcional sobrecargado. Al calcular el grafo de dependencias, encontramos que α_{map} es aplicada como función dentro de la restricción de encaje que define a α_{map}, por lo que sabemos que se trata de una función recursiva. Por ello, el siguiente paso es transformar el tipo asociado en la restricción de encaje para simplificarlo, utilizando la estrategia de transformación de tipos en funciones recursivas. Entonces, tenemos el siguiente tipo:

$$\begin{aligned}
&\forall \alpha_f, \alpha_l. (\alpha_f, \alpha_l) \to \texttt{[]} \textbf{ when } \texttt{[]} \Leftarrow \alpha_l, \texttt{'true'} \subseteq \texttt{'true'} \sqcup \\
&\forall \alpha_f, \alpha_l, \alpha_a, \alpha_b, \alpha_x, \alpha_{xs}. (\alpha_f, \alpha_l) \to \texttt{nelist(}\alpha_a\texttt{,}\alpha_b\texttt{)} \textbf{ when } \\
&\quad Z_{\alpha_f}(\alpha_x) \Leftarrow \alpha_a, Z_{\alpha_{map}}(\alpha_f, \alpha_{xs}) \Leftarrow \alpha_b, \texttt{nelist(}\alpha_x\texttt{,}\alpha_{xs}\texttt{)} \Leftarrow \alpha_l, \texttt{'true'} \subseteq \texttt{'true'}
\end{aligned} \tag{5.10}$$

Para transformar (5.10) vamos a iniciar un proceso iterativo donde vamos a introducir en C_0 la restricción $(\texttt{none()},\texttt{none()}) \to \texttt{none()} \Leftarrow \alpha_{map}$ para tener un tipo funcional inicial que pueda utilizarse en la simplificación de la restricción de la aplicación simbólica que representa la aplicación a `Map`. En esta primera iteración vamos a transformar cada esquema de tipo funcional que hay en el tipo sobrecargado. La primera rama es:

$$\forall \alpha_f, \alpha_l. (\alpha_f, \alpha_l) \to \texttt{[]} \textbf{ when } \texttt{[]} \Leftarrow \alpha_l, \texttt{'true'} \subseteq \texttt{'true'} \tag{5.11}$$

Transformamos las restricciones del tipo, porque simplificar el resto del tipo lo dejaría tal cual está actualmente. Al aplicar la regla [CHECK-D] se elimina la restricción `'true'` $\subseteq$ `'true'` y nos quedamos con una única restricción `[]` $\Leftarrow \alpha_l$, por lo que no hay nada más que hacer. Tras terminar con el conjunto de restricciones, vemos en el tipo funcional que las variables α_f y α_l solo aparecen una única vez y no pertenecen a V_0, por lo que vamos a sustituirlas. En el conjunto de restricciones tenemos información para α_l, que encaja en el tipo `[]`, por lo que para α_f usaremos la regla [ADD-ANY] para obtener que encaja en el tipo `any()`. Con la información preparada usaremos la regla [REPLACE] para sustituir α_f y α_l, y luego la regla [DELETE-C] para eliminar las restricciones que involucran a α_f y α_l, pues ya no las necesitamos. Al no quedar variables de tipo, no necesitamos ligar ninguna variable en el esquema, por lo que el resultado final es:

$$(\texttt{any()}, \texttt{[]}) \rightarrow \texttt{[]} \tag{5.12}$$

Este tipo que hemos obtenido para `Map` es el tipo del caso base de la función. Pasamos a ver el tipo de la segunda rama:

$$\begin{array}{l} \forall \alpha_f, \alpha_l, \alpha_a, \alpha_b, \alpha_x, \alpha_{xs}. (\alpha_f, \alpha_l) \rightarrow \texttt{nelist(}\alpha_a\texttt{,}\alpha_b\texttt{)} \textbf{ when} \\ \quad Z_{\alpha_f}(\alpha_x) \Leftarrow \alpha_a, Z_{\alpha_{map}}(\alpha_f, \alpha_{xs}) \Leftarrow \alpha_b, \texttt{nelist(}\alpha_x\texttt{,}\alpha_{xs}\texttt{)} \Leftarrow \alpha_l, \texttt{'true'} \subseteq \texttt{'true'} \end{array} \tag{5.13}$$

Como hicimos con la rama anterior, aplicamos [CHECK-D], para eliminar la restricción `'true'` $\subseteq$ `'true'`. Calculamos el grafo de dependencias y vemos que no hay ninguna dependencia entre restricciones. Ahora vamos a seleccionar la restricción $Z_{\alpha_f}(\alpha_x) \Leftarrow \alpha_a$ para aplicar la regla [CALL-A], porque no tenemos ningún tipo para α_f. Como resultado, obtenemos las siguientes restricciones:

$$\{(\alpha) \rightarrow \alpha' \Leftarrow \alpha_f, \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha'\} \tag{5.14}$$

Las añadimos al conjunto de restricciones correspondiente a la rama actual del tipo sobrecargado y con [DELETE-C] eliminamos la restricción con la aplicación simbólica, porque no la vamos a necesitar más en esta transformación. El siguiente paso es tomar $Z_{\alpha_{map}}(\alpha_f, \alpha_{xs}) \Leftarrow \alpha_b$ para aplicar la regla [CALL-F], dado que tenemos un tipo para α_{map} con la restricción $(\texttt{none()},\texttt{none()}) \rightarrow \texttt{none()} \Leftarrow \alpha_{map}$ que fue creada para iniciar el cálculo del punto fijo para `Map`. Como no hay variables de tipo, no va a hacer falta realizar renombramientos, por lo que obtenemos las siguientes restricciones:

$$\{\texttt{none()} \Leftarrow \alpha_b, \texttt{none()} \Leftarrow \alpha_f, \texttt{none()} \Leftarrow \alpha_{xs}\} \tag{5.15}$$

Añadiendo las restricciones y quitando con [DELETE-C] la que contiene la aplicación simbólica tendremos que:

$$\{\texttt{nelist(}\alpha_x\texttt{,}\alpha_{xs}\texttt{)} \Leftarrow \alpha_l, (\alpha) \rightarrow \alpha' \Leftarrow \alpha_f, \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \texttt{none()} \Leftarrow \alpha_b, \texttt{none()} \Leftarrow \alpha_f, \texttt{none()} \Leftarrow \alpha_{xs}\}$$

Dentro del conjunto tenemos dos restricciones de encaje para la variable α_f, por lo que usando la

regla [INFIMUM], tomaremos los tipos $(\alpha) \to \alpha'$ y `none()` y aplicaremos el ínfimo. El resultado es el tipo `none()`, transformando las restricciones de la siguiente manera:

$$\{\texttt{nelist(}\alpha_x\texttt{,}\alpha_{xs}\texttt{)} \Leftarrow \alpha_l, \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \texttt{none()} \Leftarrow \alpha_b, \texttt{none()} \Leftarrow \alpha_f, \texttt{none()} \Leftarrow \alpha_{xs}\}$$

El siguiente paso consiste en aplicar [REPLACE] para sustituir α_{xs} dentro de las restricciones, porque esta variable de tipo solo aparece una única vez más dentro del esquema de tipo funcional. El resultado, después de usar [DELETE-C] para descartar la restricción una vez hecha la sustitución, es el siguiente:

$$\{\texttt{nelist(}\alpha_x\texttt{,none())} \Leftarrow \alpha_l, \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \texttt{none()} \Leftarrow \alpha_b, \texttt{none()} \Leftarrow \alpha_f\}$$

Una vez terminadas las transformaciones con el conjunto de restricciones, se pueden sustituir las variables α_l, α_f y α_b mediante la regla [REPLACE], quedando el tipo de la siguiente manera:

$$\forall \alpha, \alpha', \alpha_a, \alpha_x.(\texttt{none()},\texttt{nelist(}\alpha_x\texttt{,none())}) \to \texttt{nelist(}\alpha_a\texttt{,none())} \textbf{ when } \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha'$$

Al normalizar el tipo obtenemos $(\texttt{none()},\texttt{none()}) \to \texttt{none()}$, que al incluirse dentro de un tipo sobrecargado es descartado, quedando el obtenido en (5.12) como resultado final. Es decir, solamente queda la rama correspondiente al caso base de la función. Para comprobar si se ha alcanzado el punto fijo, usando la relación de subconjunto de tipos polimórficos, preguntaremos si (5.12) está en el tipo de entrada de `Map`, que es $(\texttt{none()},\texttt{none()}) \to \texttt{none()}$. El resultado es que (5.12) no está contenido en el tipo de entrada, con lo que realizaremos otra iteración, usando (5.12) como tipo de entrada.

En la segunda iteración volvemos a tomar (5.10), el tipo sobrecargado obtenido inicialmente a partir del proceso de generación de restricciones, y usamos (5.12) como tipo de entrada para `Map`. El resultado de transformar (5.11) es, de nuevo, (5.12). Sin embargo, el resultado de transformar (5.13) difiere del obtenido en la anterior iteración, porque depende del tipo que tenga `Map` asignado. De nuevo con [CHECK-D] eliminamos la restricción que ejerce de condición y con [CALL-A] obtenemos las restricciones en (5.14) para la aplicación de `F`. Entonces, al aplicar [CALL-F] con la restricción $Z_{\alpha_{map}}(\alpha_f, \alpha_{xs}) \Leftarrow \alpha_b$, obtenemos las siguientes restricciones:

$$\{\texttt{[]} \Leftarrow \alpha_b, \texttt{any()} \Leftarrow \alpha_f, \texttt{[]} \Leftarrow \alpha_{xs}\} \tag{5.16}$$

Añadiendo las restricciones que se han generado y quitando con [DELETE-C] las que contienen la aplicaciones simbólicas tendremos que:

$$\{\texttt{nelist(}\alpha_x\texttt{,}\alpha_{xs}\texttt{)} \Leftarrow \alpha_l, (\alpha) \to \alpha' \Leftarrow \alpha_f, \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \texttt{[]} \Leftarrow \alpha_b, \texttt{any()} \Leftarrow \alpha_f, \texttt{[]} \Leftarrow \alpha_{xs}\}$$

Aplicaremos la regla [INFIMUM] para reducir las restricciones de α_f y [REPLACE] para sustituir α_{xs}

por `[]`, pasando a tener lo siguiente:

$$\{\texttt{nelist(}\alpha_x\texttt{,[])} \Leftarrow \alpha_l, (\alpha) \to \alpha' \Leftarrow \alpha_f, \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \texttt{[]} \Leftarrow \alpha_b\}$$

Volveremos a aplicar [REPLACE] para sustituir las variables α_l, α_f y α_b, en el tipo funcional que tenemos. El resultado que obtenemos es:

$$\forall \alpha, \alpha', \alpha_a, \alpha_x.\big((\alpha) \to \alpha', \texttt{nelist(}\alpha_x\texttt{,[])}\big) \to \texttt{nelist(}\alpha_a\texttt{,[])} \textbf{ when } \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha' \tag{5.17}$$

Terminada la transformación unimos los dos tipos obtenidos de la transformación en un único tipo sobrecargado. Para mejorar la comprensión hemos renombrado algunas de las variables de tipo ligadas, obteniendo lo siguiente como resultado para `Map` en esta iteración:

$$\begin{array}{l}(\texttt{any()},\texttt{[]}) \to \texttt{[]} \sqcup \\ \forall \alpha, \alpha', \beta, \beta'.\big((\alpha) \to \beta, \texttt{nelist(}\alpha'\texttt{,[])}\big) \to \texttt{nelist(}\beta'\texttt{,[])} \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta\end{array} \tag{5.18}$$

Al usar la relación de subconjunto, para calcular el punto fijo, tenemos que (5.18) no está contenido en (5.12). Por lo tanto, tenemos que iterar de nuevo, tomando (5.18) como tipo inicial para `Map` y transformando (5.10). De nuevo el proceso es similar hasta que tenemos que aplicar [CALL-F] con la restricción $Z_{\alpha_{map}}(\alpha_f, \alpha_{xs}) \Leftarrow \alpha_b$. A la hora de aplicar [CALL-F] vamos a obtener dos nuevas ramas de sobrecarga, porque tenemos una restricción que encaja a `Map` con un tipo sobrecargado. En la primera rama las restricciones de (5.16), se transforman en el tipo (5.17), mientras que en la segunda rama las restricciones que obtenemos son:

$$\{\texttt{nelist(}\beta_2'\texttt{,[])} \Leftarrow \alpha_b, (\alpha_2) \to \beta_2 \Leftarrow \alpha_f, \texttt{nelist(}\alpha_2'\texttt{,[])} \Leftarrow \alpha_{xs}, \alpha_2' \subseteq \alpha_2, \beta_2' \subseteq \beta_2\}$$

Juntando las restricciones y eliminando las aplicaciones simbólicas tenemos:

$$\begin{array}{c}\{\texttt{nelist(}\alpha_x\texttt{,}\alpha_{xs}\texttt{)} \Leftarrow \alpha_l, (\alpha) \to \alpha' \Leftarrow \alpha_f, \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \\ \texttt{nelist(}\beta_2'\texttt{,[])} \Leftarrow \alpha_b, (\alpha_2) \to \beta_2 \Leftarrow \alpha_f, \texttt{nelist(}\alpha_2'\texttt{,[])} \Leftarrow \alpha_{xs}, \alpha_2' \subseteq \alpha_2, \beta_2' \subseteq \beta_2\}\end{array}$$

Al aplicar [INFIMUM-X] para simplificar α_f obtenemos como resultado del ínfimo entre $(\alpha) \to \alpha'$ y $(\alpha_2) \to \beta_2$ el tipo $(\alpha) \to \alpha'$ junto a un nuevo conjunto de restricciones $\{\alpha_2 \Leftarrow \alpha, \beta_2 \Leftarrow \alpha'\}$, pasando a tener las siguientes restricciones:

$$\begin{array}{c}\{\texttt{nelist(}\alpha_x\texttt{,}\alpha_{xs}\texttt{)} \Leftarrow \alpha_l, (\alpha) \to \alpha' \Leftarrow \alpha_f, \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \\ \texttt{nelist(}\beta_2'\texttt{,[])} \Leftarrow \alpha_b, \texttt{nelist(}\alpha_2'\texttt{,[])} \Leftarrow \alpha_{xs}, \alpha_2' \subseteq \alpha_2, \beta_2' \subseteq \beta_2, \alpha_2 \Leftarrow \alpha, \beta_2 \Leftarrow \alpha'\}\end{array}$$

Utilizando [SWAP] podemos aplicar [REPLACE] para sustituir α_2 y β_2 por α y α', respectivamente, que

sumado a la sustitución de α_{xs} por el tipo `[]` da lugar a:

$$\{\texttt{nelist}(\alpha_x,\texttt{nelist}(\alpha'_2,[])) \Leftarrow \alpha_l, (\alpha) \to \alpha' \Leftarrow \alpha_f, \texttt{nelist}(\beta'_2,[]) \Leftarrow \alpha_b, \\ \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \alpha'_2 \subseteq \alpha, \beta'_2 \subseteq \alpha'\}$$

Al tomar las restricciones y transformar el tipo final, sustituyendo las variables α_l, α_f y α_b por los respectivos tipos en los que encajan, tenemos el siguiente resultado:

$$\forall \alpha, \alpha', \alpha_x, \alpha'_2, \alpha_a, \beta'_2.\big((\alpha) \to \alpha', \texttt{nelist}(\alpha_x,\texttt{nelist}(\alpha'_2,[]))\big) \to \texttt{nelist}(\alpha_a,\texttt{nelist}(\beta'_2,[])) \\ \textbf{when } \alpha_x \subseteq \alpha, \alpha_a \subseteq \alpha', \alpha'_2 \subseteq \alpha, \beta'_2 \subseteq \alpha'$$

Al unir los diferentes esquemas de tipo funcional, y renombrando variables para simplificar la lectura, obtenemos como tipo final de la transformación el siguiente:

$$\begin{array}{l} (\texttt{any}(),[]) \to [] \sqcup \\ \forall \alpha, \alpha', \beta, \beta'.\big((\alpha) \to \beta, \texttt{nelist}(\alpha',[])\big) \to \texttt{nelist}(\beta',[]) \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta \sqcup \\ \forall \alpha, \beta, \alpha_1, \alpha_2, \beta_1, \beta_2.\big((\alpha) \to \beta, \texttt{nelist}(\alpha_1,\texttt{nelist}(\alpha_2,[]))\big) \to \\ \quad \texttt{nelist}(\beta_1,\texttt{nelist}(\beta_2,[])) \textbf{ when } \alpha_1 \subseteq \alpha, \beta_1 \subseteq \beta, \alpha_2 \subseteq \alpha, \beta_2 \subseteq \beta \end{array} \tag{5.19}$$

Al usar la relación de subconjunto, para calcular el punto fijo, tenemos que (5.19) sí está contenida en (5.18), por lo que hemos alcanzado el punto fijo y no necesitamos más iteraciones. Ahora sabemos que el tipo final para `Map` es (5.18), por lo que podremos usar [REPLACE] para sustituir α_{map} en el par final que se ofrecerá como resultado de analizar la expresión **letrec**, y así terminar con el algoritmo de inferencia.

5.7.2. Analizando los tipos generados con la herramienta

En esta sección mostraremos los tipos que hemos obtenido usando la herramienta que implementa el algoritmo de inferencia. Todos los ejemplos están escritos en *Mini Erlang*. En primer lugar se mostrarán los resultados del algoritmo cuando se aplica a funciones no recursivas sencillas, para mostrar algunas funciones recursivas que, o bien realizan operaciones aritméticas, o bien manipulan listas.

Operaciones aritméticas

Antes de mostrar los ejemplos, consideramos el siguiente tipo funcional sobrecargado para el operador suma:

$$\begin{array}{l} (\texttt{integer}(),\texttt{integer}()) \to \texttt{integer}() \\ \sqcup\ (\texttt{integer}(),\texttt{float}()) \to \texttt{float}() \\ \sqcup\ (\texttt{float}(),\texttt{integer}()) \to \texttt{float}() \\ \sqcup\ (\texttt{float}(),\texttt{float}()) \to \texttt{float}() \end{array}$$

En el capítulo 3 usábamos el tipo `number()` como representación de la unión de `integer()` y `float()`, porque no teníamos la capacidad de tener tipos funcionales sobrecargados. Pero como en el sistema de tipos polimórfico y en el algoritmo de inferencia sí los tenemos, podemos usarlos para poder distinguir casos en función de los argumentos pasados al operador. Por ejemplo, si ambos argumentos son números enteros, podemos deducir que el resultado de la suma también lo es.

El primer ejemplo que vamos a considerar es una función simple que recibe un valor y le suma una unidad:

```
fun(X) -> X + 1
```

El tipo inferido que tenemos es:

$$(\texttt{integer()}) \rightarrow \texttt{integer()} \sqcup (\texttt{float()}) \rightarrow \texttt{float()}$$

Podemos ver que, dependiendo del tipo de `X`, el resultado podrá ser `integer()` o `float()`. Dado que el operando `1` es de tipo `integer()` se descartan automáticamente dos de las ramas del operador suma: $(\texttt{integer()}, \texttt{float()}) \rightarrow \texttt{float()}$ y $(\texttt{float()}, \texttt{float()}) \rightarrow \texttt{float()}$.

El siguiente ejemplo consiste en una función que recibe un valor `X`, y devuelve otra función que recibe un valor en `Y` y se lo suma al valor que hemos pasado con el parámetro:

```
fun(X) -> fun(Y) -> X + Y
```

El tipo inferido obtenido es:

$$\begin{aligned}\forall\alpha.(\alpha) \rightarrow (\texttt{integer()}) \rightarrow \texttt{integer()} \ \textbf{when}\ \texttt{integer()} \Leftarrow \alpha \\ \sqcup\ (\texttt{integer()}) \rightarrow \texttt{float()} \ \textbf{when}\ \texttt{float()} \Leftarrow \alpha \\ \sqcup\ (\texttt{float()}) \rightarrow \texttt{float()} \ \textbf{when}\ \texttt{integer()} \Leftarrow \alpha \\ \sqcup\ (\texttt{float()}) \rightarrow \texttt{float()} \ \textbf{when}\ \texttt{float()} \Leftarrow \alpha\end{aligned}$$

Esencialmente estamos replicando el tipo del operador suma, pero una parte de él queda reflejado en forma de restricciones que dependen del tipo usado para la aplicación de la λ-abstracción externa.

Función identidad y variantes

El código de la función identidad es el siguiente:

```
fun(X) -> X
```

Cuyo tipo inferido es:

$$\forall\alpha.(\alpha) \rightarrow \alpha$$

Como se puede ver, el tipo lo que nos dice es que el valor que se recibe como parámetro, representado por el tipo α, es el que se devuelve. Esto es así porque al estar en una posición donde su semántica es singular, tanto la variable posicionada en la entrada, como la que está posicionada en la salida, comparten un mismo único valor dentro de cada tupla del grafo de la función.

Antes de ver el siguiente ejemplo, vamos a introducir el tipo para la función `is_integer`:

$$(\texttt{integer()}) \rightarrow \texttt{'true'} \sqcup (\texttt{any()}) \rightarrow \texttt{'false'}$$

Esta función devuelve `'true'` si recibe un valor de tipo `integer()`, y devuelve `'false'` si recibe cualquier otro valor. Este tipo de funciones se utiliza en las guardas de funciones y cláusulas de *Erlang*, para distinguir si una variable es de un tipo determinado, cuando el comportamiento de una función depende del tipo de los parámetros.

Teniendo en cuenta lo anterior, la siguiente función es una función identidad que solamente admite números enteros:

```
fun(X) -> case X of
   Y when is_integer(Y) -> Y
end
```

El tipo inferido que tenemos es:

$$\forall \alpha.(\alpha) \rightarrow \alpha \textbf{ when } \texttt{integer()} \Leftarrow \alpha$$

Igual que con el tipo inferido para la función identidad, tenemos una variable de tipo que está como parámetro de entrada y como resultado del tipo funcional. La diferencia con el anterior consiste en que existe una restricción que obliga a α a ser de tipo `integer()`. Esta restricción será utilizada sobre los argumentos a los que la función se aplique.

Aplicación de funciones

Este es el código de la función que aplica la función recibida como primer parámetro al argumento recibido como segundo parámetro:

```
fun(F, X) -> F(X)
```

El tipo inferido para esta función es:

$$\forall \alpha_1, \alpha_2, \alpha_3, \alpha_4.((\alpha_1) \rightarrow \alpha_2, \alpha_3) \rightarrow \alpha_4 \textbf{ when } \alpha_3 \subseteq \alpha_1, \alpha_4 \subseteq \alpha_2$$

El tipo obtenido especifica que el valor recibido como segundo parámetro ha de pertenecer al dominio del tipo funcional recibido como primer parámetro. Además, el resultado de la función ha de estar

contenido en el rango de la función recibida como primer parámetro. Las restricciones obtenidas son restricciones de subconjunto, y no de encaje. Si fueran restricciones de encaje, se estaría indicando que tanto el dominio como el rango de la función pasada como parámetro contienen un único valor, cosa que no es cierta.

Si ponemos como requisito que `X` sea un número entero, la función tomará esta forma:

```
fun(F, X) -> case X of
   Y when is_integer(Y) -> F(Y)
end
```

Tras realizar esta modificación, obtenemos el siguiente tipo inferido:

$$\forall \alpha_1, \alpha_2, \alpha_3, \alpha_4.((\alpha_1) \rightarrow \alpha_2, \alpha_3) \rightarrow \alpha_4 \textbf{ when } \alpha_3 \subseteq \alpha_1, \alpha_4 \subseteq \alpha_2, \texttt{integer()} \Leftarrow \alpha_3$$

El tipo que tenemos es igual que el anterior, con la salvedad de tener una restricción adicional que pide que el segundo parámetro de entrada de la función sea de tipo `integer()`.

Operaciones aritméticas recursivas

En ejemplos anteriores vimos el tipo que habíamos supuesto para el operador suma, el cual es idéntico al de los operadores resta y multiplicación, que vamos a usar en los siguientes ejemplos. La siguiente función toma un número y devuelve el sumatorio de ese número con todos sus predecesores hasta llegar a cero:

```
letrec Sum = fun(X) ->
   case X of
      0 when 'true' -> 0
      Z when 'true' ->
         let N = X - 1 in
         let Y = Sum(N) in Y + X
   end
in Sum
```

El tipo inferido es el siguiente:

$$(\texttt{0}) \rightarrow \texttt{0} \sqcup (\texttt{integer()}) \rightarrow \texttt{integer()}$$

Lo interesante de este ejemplo es que el algoritmo de inferencia deduce que para que la función `Sum` termine necesita recibir un número entero. Al ser una función recursiva, el proceso de inferencia utiliza el algoritmo iterativo de búsqueda de punto fijo. Tras la primera iteración se obtiene el tipo $(\texttt{0}) \rightarrow \texttt{0}$,

que se corresponde con el caso base de la función. En la siguiente iteración, la variable `N`, que está en la llamada recursiva a `Sum`, tiene tipo `0` (que es un número entero), lo cual provoca que `X` sea de tipo `integer()` y, a su vez, que la suma de `X` e `Y` resulte en otro número entero.

La siguiente función, similar a la anterior, calcula el factorial de un número:

```
letrec Fact = fun(X) ->
   case X of
      0 when 'true' -> 1
      Z when 'true' ->
         let N = X - 1 in
         let Y = Fact(N) in Y * X
   end
in Fact
```

El tipo inferido es:

$$(\texttt{0}) \rightarrow \texttt{1} \sqcup (\texttt{integer()}) \rightarrow \texttt{integer()}$$

Lo mismo que ocurría con la función de sumatorio ocurre con la función factorial, aunque el caso base es la función $(\texttt{0}) \rightarrow \texttt{1}$, pero el proceso de propagación del tipo `integer()` a la variable `X` es el mismo.

Operaciones recursivas con listas

En esta sección mostraremos los resultados del algoritmo de inferencia para funciones que trabajan con listas. La primera calcula recursivamente el tamaño de una lista dada:

```
letrec Length = fun(L) ->
   case L of
      [] when 'true' -> 0
      [X|XS] when 'true' ->
         let N = Length(XS) in N + 1
   end
in Length
```

El tipo inferido obtenido es:

$$(\texttt{[]}) \rightarrow \texttt{0} \sqcup (\texttt{nelist(any(),[])}) \rightarrow \texttt{integer()}$$

El tipo especifica que si le pasamos una lista vacía el tamaño devuelto es cero, y si le pasamos una lista no vacía, su tamaño será un número entero.

La siguiente función concatena dos listas:

```
letrec Append = fun(L, YS) ->
   case L of
      [] when 'true' -> YS
      [X|XS] when 'true' -> [X|Append(XS, YS)]
   end
in Append
```

El tipo inferido que tenemos es:

$$\begin{array}{l} \forall\alpha.([],\alpha)\rightarrow\alpha \\ \sqcup\ \forall\alpha,\beta.\big(\texttt{nelist}(\alpha,[]),\beta\big)\rightarrow\texttt{nelist}(\alpha,\beta) \end{array}$$

La primera rama de la sobrecarga nos dice que es una función que recibe una lista vacía como primer parámetro. En ese caso, se devuelve como resultado el valor pasado como segundo parámetro, que no tiene por qué ser de tipo lista. La segunda rama nos dice que si el primer parámetro es una lista no vacía, tomamos el segundo parámetro para sustituir la terminación de la lista no vacía en el tipo del resultado.

A continuación tenemos la función que recibe una lista y devuelve esa misma lista invertida, utilizando un parámetro acumulador:

```
letrec Reverse2 = fun(L, YS) ->
   case L of
      [] when 'true' -> YS
      [X|XS] when 'true' ->
         let TS = [X|YS] in
         Reverse2(XS, TS)
    end
Reverse = fun(V) -> Reverse2(V, [])
in {Reverse, Reverse2}
```

Los tipos inferidos para las funciones son:

$$\begin{array}{rcl} \texttt{Reverse} & :: & ([])\rightarrow[] \\ & & \sqcup\ \forall\alpha.(\texttt{nelist}(\alpha,[]))\rightarrow\texttt{nelist}(\alpha,[]) \\ \texttt{Reverse2} & :: & \forall\alpha.([],\alpha)\rightarrow\alpha \\ & & \sqcup\ \forall\alpha,\beta.\big(\texttt{nelist}(\alpha,[]),\beta\big)\rightarrow\texttt{nelist}(\alpha,\beta) \end{array}$$

Vemos en el tipo sobrecargado de `Reverse` que el tipo del argumento es el mismo tipo que el del resultado. Como la variable de tipo α está en una posición plural, los mismos valores a los que se instancia en la entrada son los mismos que va a tener en la salida, lo cual indica que la lista devuelta contiene los mismos valores que la lista de entrada, aunque posiblemente duplicados y/o reordenados. Esto

coincide con el comportamiento de la función, que realiza una inversión del orden de los elementos. En el caso de `Reverse2` nos dice que la lista que recibimos de entrada va a contener los mismos elementos que la lista que devolvamos, pero con la terminación modificada con el tipo que represente a β.

La siguiente función dada una lista no vacía y un número `N`, nos devuelve el elemento situado en la posición `N`-ésima de la lista:

```
letrec At = fun(N, L) ->
   case L of
      [X|XS] when 'true' ->
         case N of
            0 when 'true' -> X
            Z when 'true' ->
               let M = N - 1 in At(M, XS)
         end
   end
in At
```

Obtenemos el siguiente tipo inferido:

$$\forall\alpha.(\texttt{0},\texttt{nelist}(\alpha,\texttt{any}())) \rightarrow \alpha$$
$$\sqcup\ \forall\alpha.(\texttt{integer}(),\texttt{nelist}(\texttt{any}(),\texttt{nelist}(\alpha,\texttt{any}()))) \rightarrow \alpha$$

El tipo funcional sobrecargado tiene dos ramas. La primera rama tiene como argumentos el número cero y una lista no vacía, y se devuelve el mismo tipo que tiene el cuerpo de la lista no vacía. La segunda rama recibe un número entero y una lista no vacía, pero lo que devuelve es algún elemento que esté a partir de la posición 1, porque el tipo α que devolvemos es parte del cuerpo de la lista, pero antes hay al menos un elemento de tipo `any()`. Un detalle interesante es que el tipo de la terminación de las listas es `any()` porque nunca se llega a terminar de recorrer la lista de entrada y por ello se desconoce qué puede haber al final de la misma.

La siguiente función busca un elemento dentro de una lista no vacía y devuelve la posición que tiene dentro de la misma:

```
letrec Find2 = fun(N, V, L) ->
   case L of
      [X|XS] when X =:= V -> N
      [Y|YS] when 'true' ->
         let M = N + 1 in Find2(M, V, YS)
   end
Find = fun(V1, L1) -> Find2(0, V1, L1)
```

```
in {Find,Find2}
```

El tipo inferido que tenemos es:

$$
\begin{array}{rcl}
\texttt{Find} & :: & \forall\alpha.(\alpha,\texttt{nelist}(\alpha,\texttt{any}())) \to \texttt{0} \\
 & & \sqcup\ \forall\alpha.(\alpha,\texttt{nelist}(\texttt{any}(),\texttt{nelist}(\alpha,\texttt{any}()))) \to \texttt{integer}() \\
\texttt{Find2} & :: & \forall\alpha,\beta.\big(\alpha,\beta,\texttt{nelist}(\beta,\texttt{any}())\big) \to \alpha \\
 & & \sqcup\ \forall\alpha.(\texttt{integer}(),\alpha,\texttt{nelist}(\texttt{any}(),\texttt{nelist}(\alpha,\texttt{any}()))) \to \texttt{integer}() \\
 & & \sqcup\ \forall\alpha.(\texttt{float}(),\alpha,\texttt{nelist}(\texttt{any}(),\texttt{nelist}(\alpha,\texttt{any}()))) \to \texttt{float}()
\end{array}
$$

La primera sobrecarga del tipo obtenido para `Find` indica que si el tipo del primer parámetro coincide con el tipo de los elementos de la lista, el resultado es `0`. Esto se debe a que α está situada en una posición singular (el primer parámetro de `Find`), y eso obliga a instanciarse a un único valor, que ha de coincidir con el primer elemento de la lista pasada como segundo parámetro (también puede coincidir con los siguientes, en caso de ser iguales al primero). La segunda sobrecarga trata los casos en los que el valor α buscado se encuentra más allá de la cabeza de la lista. El tipo `nelist(any(),nelist(`α`,any()))` engloba a aquellas listas cuyos primeros n elementos ($n \geq 1$) tienen tipo `any()`, y el $n+1$-ésimo elemento tiene tipo α, que coincide con el primer parámetro. El tipo funcional sobrecargado para `Find2` es similar, pero tiene un parámetro adicional en la primera posición, que en la primera rama se devuelve cuando el elemento que se busca está en la primera posición, pero en las dos otras ramas es un número y, en ese caso, se devuelve otro número.

Por último, en cuanto a funciones que trabajan con listas, presentamos una función de orden superior que filtra elementos en base a un predicado. Recibe una función que ejerce de predicado y una lista, para devolver otra lista que solo contiene aquellos elementos que satisfacen el predicado:

```
letrec Filter = fun(F, L) ->
   case L of
      [] when 'true' -> []
      [X|XS] when 'true' ->
         let T = F(X) in case T of
            'false' when 'true' -> Filter(F, XS)
            'true' when 'true' -> [X|Filter(F, XS)]
         end
   end
in Filter
```

El tipo inferido para la función es:

$$
\begin{array}{l}
(\texttt{any}(),[]) \to [] \\
\sqcup\ \forall\alpha,\beta,\alpha'.\big((\alpha) \to \beta,\texttt{nelist}(\alpha',[])\big) \to [] \textbf{ when } \alpha' \subseteq \alpha, \texttt{'false'} \subseteq \beta \\
\sqcup\ \forall\alpha,\beta,\alpha'.\big((\alpha) \to \beta,\texttt{nelist}(\alpha',[])\big) \to \texttt{nelist}(\alpha',[]) \textbf{ when } \alpha' \subseteq \alpha, \texttt{'true'} \subseteq \beta
\end{array}
$$

El tipo funcional sobrecargado en este caso tiene tres ramas. La primera describe el caso en el que el segundo parámetro es una lista vacía, caso en el que el tipo de retorno es una lista vacía también. La segunda rama pide que el tipo del resultado del predicado contenga el tipo `'false'`, así como que el tipo del cuerpo de la lista esté contenido en el dominio del predicado, para devolver la lista vacía. La tercera rama describe el caso en el que el tipo del resultado del predicado es el tipo `'true'`, para devolver una lista no vacía con el mismo tipo que el que tiene la lista no vacía de entrada.

Recursividad con tuplas

Esta función nos va a servir para ver la clase de tipos que obtenemos cuando se aplica el *widening* en el algoritmo de inferencia. En este caso la función recibe un número y va devolviendo tuplas que tienen el número `N` recibido en la primera componente, y en la segunda componente se devuelve el resultado de aplicar la función recursivamente con `N-1`. Se llega al caso base cuando se recibe el número cero, caso en el que se devuelve una tupla vacía. El código es el siguiente:

```
letrec Foo = fun(X) ->
   case X of
      0 when 'true' -> {}
      Z when 'true' ->
         let N = X - 1 in
         let Y = Foo(N) in {X, Y}
   end
in Foo
```

Por ejemplo, `Foo(3)` se evalúa un valor consistente en cuatro tuplas anidadas: `{3, {2, {1, {}}}}`. Con un límite de cuatro iteraciones para la transformación recursiva, el tipo inferido que hemos obtenido es:

$$
\begin{aligned}
&(\texttt{0}) \rightarrow \{\} \\
&\sqcup\ \forall\alpha.(\alpha) \rightarrow \{\alpha, \{\}\}\ \textbf{when}\ \texttt{integer()} \Leftarrow \alpha \\
&\sqcup\ \forall\alpha.(\alpha) \rightarrow \{\alpha, \{\texttt{any()}, \texttt{any()}\}\}\ \textbf{when}\ \texttt{integer()} \Leftarrow \alpha
\end{aligned}
$$

La primera rama del tipo funcional sobrecargado es el caso base de la función, la entrada es el tipo cero y la salida el tipo tupla vacía. En la segunda rama, la entrada es el tipo α, que tiene una restricción que exige que α sea un número entero, y la salida es un tipo tupla, donde la primera componente es α y la segunda la tupla vacía. La tercera rama es como la anterior, pero la segunda componente del tipo tupla devuelto tiene otra tupla que, en principio, hubiera sido de tipo $\{\alpha', \{\}\}$, con una restricción $\texttt{integer()} \Leftarrow \alpha$, pero al sobrepasar el número de iteraciones—en la inferencia de funciones recursivas—se aplica el el proceso de *widening* y se recorta el tipo para alcanzar el punto fijo, obteniendo `{any(),any()}` en su lugar.

5.7.3. Detección de errores de tipo

A continuación vamos a ver varios ejemplos de aplicaciones de funciones y los resultados obtenidos por nuestra herramienta. Los dos primeros ejemplos utilizan la función identidad para ver cómo se comporta el polimorfismo en nuestro sistema. El código del primero es el siguiente:

```
letrec
    Id = fun(X) -> X
    Foo = fun(A) -> A + 1
    Good = fun() -> let V = Id(1) in Foo(V)
in Good
```

Vemos definida la función identidad como `Id`, luego una función `Foo` que recibe un valor y le suma una unidad y, por último, una función `Good` que aplica la identidad al valor `1` y a su vez, aplica el resultado a la función `Foo`. El tipo que nuestra herramienta obtiene para `Good` es $() \rightarrow$ `integer()`, como era de esperar. No obstante, en nuestro segundo ejemplo, en vez de aplicar a la identidad un valor numérico, vamos a aplicarla a un átomo:

```
letrec
    Id = fun(X) -> X
    Foo = fun(A) -> A + 1
    Fail = fun() -> let V = Id('a') in Foo(V)
in Fail
```

El resultado que obtenemos con nuestra herramienta para la función `Fail` es el tipo $() \rightarrow$ `none()`, ya que `Foo` no permite ser aplicada con valores no numéricos. Gracias al polimorfismo podemos conectar el valor del resultado de la función identidad con su parámetro de entrada y así inferir que la aplicación de `Foo` fallaría. En *Erlang* el código de estos dos ejemplos sería el siguiente:

```
id(X) -> X.
foo(X) -> X + 1.
fail() -> foo(id('a')).
good() -> foo(id(1)).
```

Suponiendo que las cuatro funciones son exportadas fuera del módulo y que, por tanto, *Dialyzer* realiza el análisis de manera modular para cada una de ellas, el resultado que nos daría *Dialyzer* para estas funciones sería el siguiente:

```
-spec id(any()) -> any().
-spec foo(number()) -> number().
-spec fail() -> number().
-spec good() -> number().
```

Como se puede observar, la función identidad obtiene como tipo (`any()`) → `any()`, sin polimorfismo ninguno por parte de *Dialyzer*, por lo que el tipo de `fail` devuelve `number()` en vez de `none()` como resultado.

Para el siguiente ejemplo vamos a aplicar la función `Map` con una lista de números y la función `Foo` que acabamos de ver en los ejemplos anteriores:

```
letrec Map = fun(F, L) ->
    case L of
        [] when 'true' -> []
        [X|XS] when 'true' -> [F(X)|Map(F, XS)]
    end
in let Foo = fun(A) -> A + 1
in let L2 = [1, 2, 3, 4, 5]
in Map(Foo, L2)
```

El tipo obtenido para la aplicación `Map(Foo, L2)` es `nelist(integer(),[])`. Sin embargo, cuando `L2` toma como valor `['a', 'b', 'c', 'd']`, el tipo que obtenemos es `none()`, ya que no hay ningún valor posible dentro de la lista que pudiera tener éxito con la función `Foo`. Si `L2` tomara como valor `[1, 'a', 2, 'b']`, volveríamos a obtener como resultado `nelist(integer(),[])` al existir números enteros dentro de la lista de entrada a la aplicación de `Map`. Esto es así porque en principio con el tipo de una función polimórfica—como `Map`—no podemos saber si la función que recibe como argumento se está aplicando a todos los elementos de la lista o solamente a algunos de ellos, así que el algoritmo de inferencia—al saber que existen números enteros en la lista—asume que podría existir una posibilidad de éxito y devuelve como resultado el tipo `nelist(integer(),[])`. El código que tendríamos en *Erlang* para este ejemplo sería el siguiente:

```
map(_, []) -> [];
map(F, [X|XS]) -> [F(X) | map(F, XS)].
foo(X) -> X + 1.
test1() -> map(fun foo/1, [1, 2, 3, 4, 5]).
test2() -> map(fun foo/1, ['a', 'b', 'c', 'd']).
test3() -> map(fun foo/1, [1, 'a', 2, 'b']).
```

De nuevo, suponiendo que todas las funciones son exportadas fuera del módulo, el resultado que *Dialyzer* devolvería, para cada función, sería el siguiente:

```
-spec map(any(),[any()]) -> [any()].
-spec foo(number()) -> number().
-spec test1() -> [any()].
-spec test2() -> [any()].
-spec test3() -> [any()].
```

La ausencia de polimorfismo en *Dialyzer* provoca que todas las aplicaciones de `map` terminen siendo de tipo `[any()]`.

Bajo determinadas circunstancias es posible obtener algunos resultados más precisos con *Dialyzer*. Para aquellas funciones que no son exportadas fuera del módulo, *Dialyzer* aplica los *success types refinados* descritos en [62], obteniendo algo más de precisión para estas funciones no exportadas, pero esta técnica no se puede emplear con las funciones que sí se exportan. Esta limitación hace que no sea práctico su uso al estudiar cómo se comportan diferentes módulos entre sí, como ocurre cuando tenemos una aplicación que utiliza una biblioteca determinada para alguna tarea, por lo que para analizar la interfaz que ofrece un módulo al programador, solo cabe usar los *success types* sin refinar.

Debido a la limitación de los *success types* refinados, *Dialyzer* implementa de forma *ad hoc* cierto análisis polimórfico de las aplicaciones de funciones que pertenecen a la biblioteca estándar de *Erlang/OTP*, como es el caso de la función `map` en el módulo `lists`. Para ello, dentro del fichero `erl_bif_types.erl` de la aplicación *Dialyzer* tenemos la función `erl_bif_types:type/5`, donde son procesadas las aplicaciones de una lista de funciones primitivas del lenguaje y otras pertenecientes a algunos módulos de la biblioteca estándar. Este enfoque conlleva la necesidad de modificar la implementación del fichero mencionado anteriormente para incorporar el tratamiento de aquellas funciones no contempladas en él, ya sean definidas por el programador, ya sean funciones de la librería estándar de *Erlang*. De cara a formalizar esta forma de trabajar de *Dialyzer*, estaríamos creando tantas reglas de la aplicación particulares, como casos necesitemos contemplar, con sus correspondientes resultados de corrección, en lugar de tener las dos reglas de la aplicación que tenemos con nuestro sistema. Por lo tanto, sin la ayuda de este tratamiento *ad hoc*, *Dialyzer* no lograría obtener los resultados mostrados para las aplicaciones de `lists:map/2` mediante su algoritmo de inferencia, que a efectos prácticos es monomórfico. Además, esta solución no está disponible para los módulos de los usuarios, los cuales, para poder analizar con mayor precisión sus aplicaciones seguirían necesitando un sistema de tipos polimórficos, como el que proponemos en nuestro trabajo. También cabe señalar que esta solución dista de ser todo lo efectiva que podría haber sido con un sistema de tipos polimórfico, pues si por ejemplo aplicamos `lists:map/2` con la función identidad (indicando su correspondiente tipo con variables de tipo) y una lista de números, el resultado inferido es una lista de `any()` porque el algoritmo de inferencia de *Dialyzer* es monomórfico.

5.8. Sobre la precisión del algoritmo

Hemos visto el funcionamiento del algoritmo de inferencia propuesto por esta tesis, al igual que los tipos obtenidos por diversos ejemplos analizados con la herramienta implementada. También sabemos, por la sección 1.3, que no siempre es posible obtener el *success type* mínimo para una expresión. Por ello cabe preguntarse cómo de precisos son los *success types* obtenidos con la inferencia de tipos, en concreto cuánta precisión se pierde durante las transformaciones realizadas.

El primer punto de pérdida de precisión lo tenemos en la semántica de los tipos, vista en la sección 4.3. Con las listas no vacías nos encontramos que, si tenemos la lista `[1]`, el tipo que usaríamos sería `nelist(1,[])`, este no solo representa la lista `[1]`, sino todas las listas de tamaño arbitrario que contengan el tipo 1. Para resolver esto se pensó en tener un tipo que fuera el constructor de lista, pero se descartó al ser difícil encontrar ejemplos que no resultaran artificiales a la hora de justificar la necesidad de esta clase de tipos. Por ello se asumió como aceptable la pérdida de precisión que conllevaba el uso del tipo `nelist()`.

Durante la generación de restricciones, en la sección 5.2, la regla [APP] puede añadir indirectamente pérdidas de precisión, ya que depende de lo preciso que sea el tipo que representa a la función f aplicada. En el caso de la expresión **case** o **receive** se devuelve un tipo anotado con tantas ramas como cláusulas haya, conformando una unión de valores que contiene a la semántica de la expresión. Es decir, que podemos perder cierta precisión en estas dos reglas, porque la semántica de estas estructuras de control dependen del orden en el que las cláusulas son declaradas. Sin embargo, el tipo unión no tiene en consideración esta eventualidad, y por ello aquí también se puede perder precisión con respecto a la semántica de las expresiones. Por ejemplo, sea la función:

```
f(X) -> case X of
          0 -> 1;
          _ -> 2
        end
```

El tipo inferido para la expresión sería $(0) \to 1 \sqcup (\texttt{any()}) \to 2$, que incluye la tupla $(0,2)$ que no está en el grafo de la semántica de `f`. Para las reglas de la cláusula y las expresiones **let** y **letrec** no se añade ninguna pérdida de precisión, porque la naturaleza de las restricciones de encaje limita los valores de las variables de tipo que participan en ella. Para el resto de reglas en la generación no se advierten pérdidas de precisión que no vengan de aquellas incorporadas por la semántica de los tipos, ya que se usa el tipo más ajustado posible para la expresión cuando esta es un literal, variable de programa, tupla, lista no vacía o λ-abstracción.

Dentro de la sintaxis normalizada, vista en la figura 5.1, no se permite que el τ de las diferentes formas de restricciones que hay sea un tipo **when**. Por ello en la normalización, descrita en la sección 5.4, tenemos dos reglas para extraer los **when** de restricciones de encaje. Sin embargo, en las restricciones de subconjunto no hay ninguna regla y el curso a tomar es eliminar el conjunto de restricciones para normalizar el tipo, perdiendo precisión con ello. Por ejemplo, si tenemos:

$$\{\alpha, \beta\} \textbf{ when } \alpha \subseteq \big(\{\beta, \beta\} \textbf{ when } \beta \subseteq 1\big)$$

El único valor posible para este tipo es $\{\{1,1\},1\}$, pero como no tenemos ninguna regla para extraer tipos **when** de restricciones de subconjunto, al normalizar el tipo anterior, obtendríamos que β se puede instanciar a cualquier otro valor distinto del 1. En este ejemplo en particular se podría extraer

la restricción, sin cambiar la semántica del tipo, pero no siempre es posible extraer los **when** desde una restricción de subconjunto, como ocurriría en el siguiente caso:

$$\texttt{nelist(\{}\alpha,\beta\texttt{\},[])}\ \textbf{when}\ \alpha \subseteq \big(\texttt{\{}\beta,\beta\texttt{\}}\ \textbf{when}\ \beta \subseteq 1\big)$$

Aquí α tendrá como valor siempre `{1,1}`, mientras que β puede estar instanciada a cualquier conjunto de valores, siempre y cuando este conjunto contenga el literal 1. Por ejemplo, un valor posible para el tipo anterior podría ser la lista `[{{1,1},'a'},{{1,1},1}]`, que no sería posible si se saca las restricciones del **when**. De cara a solventar esta pérdida de precisión, se tendría que estudiar cómo se podría extender el algoritmo de inferencia para tipos que no estén normalizados conforme a la noción de normalización de la sección 5.1.1, o si se puede disponer de una sintaxis para los tipos normalizados que sea más laxa que la actual.

Dentro de las reglas de transformación, en la sección 5.5, la regla [DELETE-C] puede provocar una pérdida de precisión al eliminar restricciones de un conjunto, por ello se procura usar esta regla bajo circunstancias muy concretas, como la eliminación de restricciones con aplicaciones simbólicas después de aplicar [CALL-F] o [CALL-A]. También hay que tener en cuenta, a la hora de resolver las restricciones con aplicaciones simbólicas, que las reglas [CALL-F] y [CALL-A] pueden introducir indirectamente imprecisión si el tipo funcional de f no es lo suficientemente preciso. Esto último puede ser un problema serio al tomar como entrada los tipos descritos por el usuario, para las funciones de sus módulos.

El punto más claro donde se pierde precisión dentro del algoritmo es al aplicar la función de *widening*, vista en la sección 5.6.1, ya que sería aquel caso en el que tenemos una función recursiva que no alcanza un punto fijo en el proceso de inferencia de tipos. En el último ejemplo, de la sección 5.7.2, tenemos un ejemplo de una función recursiva con tuplas que nos muestra un tipo obtenido con *widening*. Como se pudo ver en la sección 1.3, para el ejemplo de la función `g()` que disponía de una cadena decreciente infinita de *success types*, no existe un tipo mínimo que pueda ser representado de forma finita. Una posible solución sería extender el sistema de tipos con tipos recursivos y estudiar las ventajas e inconvenientes que podrían tener sobre el algoritmo de inferencia. Sin tipos recursivos, lo más que podemos hacer—para aumentar la precisión—es aumentar el número de iteraciones a la hora de buscar el punto fijo para la función antes de aplicar *widening*.

Por último, otro punto crítico donde se podría perder precisión en nuestro sistema, es la operación de relación de subconjunto polimórfico. En esta operación lo importante es evitar falsos positivos, pero en el caso de los falsos negativos, es decir, que no seamos capaces de detectar que efectivamente τ está contenido en τ', se continuaría con el proceso iterativo de buscar el punto fijo para la función recursiva analizada, pudiendo alcanzar la función de *widening* y perder precisión con ello de manera innecesaria. Por ello la implementación de esta operación es fundamental para la inferencia de funciones recursivas.

5.9. Implementación de la herramienta

Para poder poner en práctica los resultados obtenidos con el desarrollo teórico de esta investigación, se decidió hacer el prototipo *Mini-Erlang Typing Application*[2] para implementar el algoritmo de inferencia aplicado a *Mini Erlang*. En la sección 5.7.2 se muestran, para una serie de ejemplos disponibles en el repositorio, los tipos obtenidos con la propia herramienta. Su desarrollo comenzó en enero del 2019 y su implementación llevó 18 meses a lo largo de los últimos 3 años. El prototipo está escrito en *Erlang* prácticamente en su totalidad. Está compuesto de 24 módulos que ocupan un total de 12.723 líneas, de las cuales más de seis mil son de código efectivo. En este recuento de líneas no se ha incluido el fichero `minierlang.erl`, que fue generado con el módulo *yecc* de *Erlang/OTP*, un generador de parsers LALR-1 para *Erlang* similar al conocido *yacc* [59]. La sintaxis de *Mini Erlang* está definida en el fichero `minierlang.yrl` y ocupa 100 líneas de código.

La versión actual de la implementación de la herramienta soporta únicamente el lenguaje *Mini Erlang*. Al no soportar *Mini Erlang* la totalidad de las características del lenguaje *Erlang*, resulta imposible poder traducir código de *Erlang* a *Mini Erlang* para poder analizarlo. Esta imposibilidad impidió poder realizar evaluaciones experimentales, analizando aplicaciones y bibliotecas de código abierto conocidas, para obtener resultados con sus respectivas métricas de tiempo. Hubiera sido interesante disponer de estas métricas para realizar comparativas reales de eficiencia frente a *Dialyzer*, por lo que parte del trabajo futuro de la herramienta será incorporar soporte para el lenguaje *Erlang*.

La organización del proyecto se ha hecho en módulos de *Erlang*, siendo el punto de entrada el módulo `meta`. Entre los módulos encontramos las siguientes categorías:

- **Lenguaje *Mini Erlang***: El módulo `minierlang` es el parser generado por el módulo *yecc* de *Erlang* y `language` incorpora algunas funciones auxiliares para trabajar con los árboles sintácticos del lenguaje *Mini Erlang*.
- **Definición de tipos**: Los siguientes módulos representan los tipos del sistema, así como algunas operaciones auxiliares para poder trabajar con ellos.
  - `type`: Este módulo permite definir tipos anotados ρ y tipos normales τ. También permite algunas operaciones básicas para modificar y consultar información dentro de un tipo.
  - `constraint`: Este módulo permite definir restricciones y conjuntos de ellas. Al igual que `type`, este módulo contiene algunas operaciones básicas para modificar y consultar información dentro de las restricciones.
  - `mono`: Este módulo define para tipos monomórficos las operaciones de igualdad, relación de subconjunto, ínfimo y supremo.
  - `poly`: Este módulo define para tipos polimórficos las operaciones de igualdad, relación de subconjunto, ínfimo y supremo.

[2] El código fuente está disponible en el siguiente repositorio: https://github.com/gorkinovich/meta

  - `ground`: Este módulo es el encargado de la transformación de un tipo polimórfico en su correspondiente *cota superior ground*.
  - `query`: Este módulo permite realizar consultas sobre tipos anotados, para obtener, por ejemplo, las variables de tipo libres dentro de un tipo, entre otras operaciones.

- **Inferencia de tipos**: Estos son los módulos que gestionan las operaciones que conforman el núcleo del algoritmo de inferencia de tipos.
  - `analyze`: Este módulo se encarga de obtener la restricciones generadas para un árbol sintáctico abstracto de *Mini Erlang*.
  - `normalize`: Este módulo se encarga de las operaciones que normalizan un tipo durante la transformación de los tipos para reducirlos.
  - `reduce`: Este módulo es el encargado de transformar los tipos para presentar un tipo lo más simplificado y legible al usuario final.
  - `rule_substitution`: Este módulo se ocupa de aplicar la regla de la sustitución sobre un tipo.
  - `rule_symbol_call`: Este módulo se ocupa de aplicar la regla de la aplicación simbólica sobre un conjunto de restricciones.
  - `widening`: Este módulo contiene la función de *widening*, usada durante la transformación de funciones recursivas cuando no se alcanza el punto fijo.

- **Operaciones auxiliares**: `environment` es una estructura de datos que relaciona variables con tipos, `table` es una estructura de datos auxiliar para organizar conjuntos de restricciones de cara a la inferencia, `text` permite representar un tipo anotado como una cadena de texto, `data` gestiona la configuración de la herramienta, `tools` ofrece algunas operaciones auxiliares para la transformación de tipos, y `util` contiene funciones útiles varias para trabajar con diferentes tipos de *Erlang*.

Como ya hemos dicho, la entrada principal es a través del módulo `meta`. Una vez configuradas las opciones del comando, se procesa cada fichero de entrada individualmente. Con cada fichero se generan primero unas restricciones con el módulo `analyze`, siguiendo las reglas vistas en la sección 5.2, y después se aplica una transformación con el módulo `reduce`, siguiendo las transformaciones descritas en las secciones 5.5 y 5.6, para eliminar las aplicaciones simbólicas y simplificar el resultado en la medida de lo posible. Para poder gestionar la información necesaria entre las diferentes transformaciones dentro de la reducción, se utiliza una estructura de registro que almacena información útil sobre el estado actual del proceso de transformación. La información de estado que se tiene es la siguiente:

- Un entorno donde se asocian variables de programa con los tipos que se van obteniendo para dichas variables con la transformación.

- Una lista donde quedan registradas las variables de programa que han de ser vigiladas al crear un tipo anotado o un tipo **when** para comprobar que no sean de tipo `none()`.
- La lista de variables de tipo libres que hay fuera del contexto actual de la subexpresión que se está transformando.
- La lista de variables que necesitan ser revaluadas por haber sido su tipo actualizado dentro del entorno del estado actual.
- Un *flag* necesario para la sustitución de variables de tipo, a la hora de aplicar la regla sobre los tipos de los parámetros de un tipo funcional, que indica si las restricciones que se están usando pertenecen al tipo del resultado.

Una vez inicializado el registro del estado actual, usaremos la función para normalizar tipos, siempre que la necesitemos a lo largo de la transformación, y cuyo comportamiento está descrito en las reglas de la sección 5.4. Para realizar la reducción del tipo se ha de recorrer el árbol sintáctico del mismo, actualizando convenientemente el registro del estado actual, para que—al encontrar un par $\langle \tau; C \rangle$ o un tipo **when**—invoque la función que gestiona aquellos tipos que están acompañados por un conjunto de restricciones. También se transformará el tipo cuando se trate de un tipo funcional; tras obtener la modificación del tipo del resultado, tendremos que volver a reconstruir el tipo funcional con la función encargada para tal propósito, dentro de la cual se aplicará la regla de la sustitución de variables de tipo.

Dentro de la función encargada de transformar tipos que están acompañados de restricciones se invoca primero una función para transformar el conjunto de restricciones. Una vez obtenido el nuevo conjunto de restricciones modificado, se transforma el tipo que acompañaba a las restricciones. Con el resultado se comprueba que las variables de tipo asociadas a las variables de una aplicación simbólica no sean de tipo `none()`, para luego reconstruir el tipo anotado o el tipo **when**.

La función que transforma un conjunto de restricciones crea primero un árbol de dependencias entre variables de tipo para resolver las transformaciones en orden de necesidad y detectar aquellas llamadas recursivas que existan. Si tenemos que transformar un grupo de restricciones asociadas a una variable de tipo, tendremos que distinguir si esta representa una función recursiva o no:

- **Caso general:** La función que gestiona este caso va tomando cada variable de tipo de la lista que recibe, para transformar cada una de ellas. Se toma cada restricción—asociada a la variable de tipo indicada—para obtener su simplificación. Una vez obtenida la restricción modificada, se aplican algunas de las reglas de transformación vistas en la sección 5.5.1. Primero se aplica la función encargada de la transformación de las aplicaciones simbólicas, cuyo comportamiento se basa las reglas [CALL-F] y [CALL-A]. Después se aplica la función que limpia de restricciones que no aportan nada al conjunto, cuyo comportamiento se basa en la regla [NOISE-A] de la sección 5.4. Por último, se aplica la función del ínfimo entre restricciones, cuyo comportamiento se basa en las reglas [INFIMUM] e [INFIMUM-X].

- **Caso funciones recursivas:** La función que gestiona este caso inicializa primero los tipos de las variables de tipo que se van a procesar, para luego iniciar un proceso iterativo con un número fijo de pasos, donde se aplica la función del caso general de transformación de conjuntos de restricciones asociadas a una variable de tipo. Con cada aplicación de la transformación de las restricciones, se comprueba si se ha alcanzado el punto fijo y si no se alcanza se aplicará la función de *widening*, que está descrita en la sección 5.6.1.

Capítulo 6

Conclusiones

En este capítulo final cerramos la exposición de este trabajo con una serie de conclusiones en la sección 6.1 y explicamos en la sección 6.2 cuál es el trabajo futuro que se podría abordar en una posterior investigación basada en esta tesis.

6.1. Conclusiones finales

Una de las principales motivaciones de nuestra investigación ha consistido en superar algunas de las limitaciones que *Dialyzer* tiene como herramienta de análisis de tipo estático para *Erlang*, como es la falta de tipos polimórficos efectivos en su sistema de tipos. Para ello, en este trabajo, hemos presentado un conjunto de reglas, vistas en el capítulo 4, para tipar expresiones de un subconjunto importante de *Core Erlang* al que hemos llamado *Mini Erlang*, lenguaje para el que hemos definido una semántica denotacional. Las reglas del sistema de tipos nos permiten derivar especificaciones de tipo polimórficas, que pueden estar sobrecargadas de un modo similar a las vistas en [55] y por lo tanto capturar la semántica de una función de un modo más preciso. Pero para lograr derivar tipos polimórficos no solo ha habido que definir una sintaxis para los tipos que fuera adecuada a nuestras necesidades, sino que también ha sido necesario definir una semántica para poder demostrar que, en efecto, los tipos derivados para una expresión son *success types* de la misma.

El definir la semántica de los tipos polimórficos no se ha limitado a simplemente añadir variables de tipo al sistema de tipos monomórfico del capítulo 3, sino que ha supuesto un importante reto a lo largo de la investigación. Dada la naturaleza de los *success types* el sistema requería una relación de subtipado y, mientras que resulta relativamente fácil de entender la relación de subtipado entre los tipos monomórficos, no resulta tan inmediato expresar cuándo un tipo con variables es subtipo de otro a no ser que se indiquen los conjuntos de valores denotados por cada una de las variables de tipo que aparecen en él. De ahí surge la necesidad de disponer de instanciaciones de tipo. Para poder expresar el subtipado entre tipos con variables fue necesario disponer de las instanciaciones de

tipo, para poder relacionar una variable de tipo α con un conjunto de valores concretos y poder así demostrar si está contenida en otra variable de tipo β. No obstante, las instanciaciones de tipo conllevaron tomar decisiones adicionales para modelar la relación existente entre las múltiples apariciones de una misma variable de tipo α en un tipo τ, ya que esto era importante para poder expresar la relación que existe entre las salidas y las entradas de un tipo que represente a una función. Por ejemplo, si queremos mantener la noción de parametricidad de Walder [120] para la función identidad, cuyo tipo es $\forall\alpha.(\alpha) \to \alpha$, si no hubiéramos definido correctamente la semántica de las variables de tipo y cómo estas se descomponen—entre las diferentes estructuras sintácticas que hemos definido—, el resultado sería que, al coger una instancia π para α con múltiples valores, α podría tomar valores diferentes para la entrada y la salida dentro de dicha instancia, lo que terminaría siendo equivalente al tipo $(\texttt{any()}) \to \texttt{any()}$, que es lo que precisamente queríamos evitar con nuestra investigación. Por ello la semántica para una variable de tipo obliga que, para cada tupla del grafo de una función de tipo $\forall\alpha.(\alpha) \to \alpha$, el conjunto al que está instanciada contenga un único valor. Este paso lógico nos obligó a formular en la semántica una función de descomposición para el caso de listas no vacías y tipos funcionales, ya que estas estructuras pueden manejar instancias con conjuntos de más de un valor para las variables de tipo que hay dentro de ellas. Como consecuencia de tener instancias para variables de tipo, también hubo que adaptar la semántica de los tipos unión para evitar que se instancien variables que no deberían hacerlo, como ocurre en el ejemplo de la sección 4.3 del tipo funcional $\forall\alpha_1,\alpha_2.([\alpha_1]) \to [\alpha_2]$ **when** $\alpha_2 \subseteq \alpha_1$, que recibe una lista y devuelve una lista que depende de la primera. En este ejemplo explicamos que se devolverá una lista vacía si se recibe una, esto es así gracias a que, en la semántica que hemos definido para los tipos unión $\tau \cup \tau'$, se borran en τ las variables de tipo instanciadas en τ' que no existen en τ.

Nuestra semántica de tipos también dispone de restricciones para acotar los conjuntos de valores a los que se pueden instanciar las variables de tipo. Inicialmente se tenía solamente las restricciones de subconjunto para restringir las variables de tipo, pero nos encontramos con que se perdía precisión al mover los tipos asociados a una variable de programa desde los entornos de tipos a las restricciones. Esto motivó la necesidad de crear las restricciones de encaje $\tau \Leftarrow \alpha$, que nos permiten transformar un entorno $[x : \tau]$ en un entorno $[x : \alpha \mid \tau \Leftarrow \alpha]$ sin perder precisión (cosa que no ocurre con $[x : \alpha \mid \alpha \subseteq \tau]$). Esto resulta de gran utilidad para poder definir la operación de ínfimo entre entornos. Una ventaja añadida a este tipo de restricciones, es que las restricciones con la forma $\beta \Leftarrow \alpha$ expresan una igualdad entre variables de tipo, en vez de necesitar tener $\beta \subseteq \alpha$ y $\alpha \subseteq \beta$ en el conjunto de restricciones para hacer lo mismo.

Formalmente, los juicios de tipado derivados con nuestras reglas obtienen, bajo un entorno de tipos dado, un tipo anotado para una expresión e. Dentro de cada par en el tipo anotado obtenido, podemos encontrar un nuevo entorno de tipos con el que expresar las condiciones sobre las variables libres en e que son necesarias para la correcta evaluación de la expresión, además de un conjunto de restricciones para las variables de tipo que hay en el par. Cuando se aplican las reglas a expresiones cerradas, estas derivan *success types*, es decir, sobreaproximaciones de la semántica. Este hecho, reflejado en el colorario 2 es el resultado técnico que expresa que, en efecto, nuestro sistema de tipos respeta el

enfoque de los *success types*.

La sintaxis de los tipos vista en esta tesis implica la existencia de tipos cuantificados universalmente anidados dentro de otros tipos funcionales, similares a los tipos del Sistema F [39]. Aunque la inferencia de tipos en el Sistema F es no computable, en nuestro contexto este problema se vuelve trivial, ya que siempre podemos derivar un tipo para cualquier expresión. De hecho, siempre podremos derivar el tipo `any()`, como se refleja en la proposición 11.

También hemos introducido en este trabajo, en el capítulo 5, un conjunto de reglas con las que poder inferir y transformar tipos para expresiones dadas en *Mini Erlang*. Estos tipos están conectados con el sistema de tipos del capítulo 4, como se demuestra en la corrección de las reglas de inferencia, y por lo tanto también son *success types* de las expresiones analizadas. Como resultado del algoritmo de inferencia hemos implementado una herramienta escrita en *Erlang* con la que inferir tipos sobre ejemplos escritos en *Mini Erlang* (`https://github.com/gorkinovich/meta`).

El trabajo realizado en los capítulos 4 y 5 ha dado como resultado la construcción de un *sistema de tipos flexible* que dispone del *marco teórico sólido* que necesitamos para garantizar que derivamos e inferimos *success types* polimórficos de forma correcta a diferencia de *Dialyzer*, que carece de tipos polimórficos. Para conseguir este marco teórico sólido han hecho falta años de investigación y múltiples evoluciones sobre los sistemas que fuimos desarrollando, empezando por un sistema monomórfico hasta finalizar con la obtención de uno polimórfico con tipos funcionales sobrecargados. Sabemos que los lenguajes de tipado estático tienen una fuerte comunidad detrás de ellos, lo cual ha motivado la aparición de herramientas que incorporan la seguridad de los sistemas de tipado estático, sacrificando parte de la flexibilidad que disponen los lenguajes de tipado dinámico, para desarrollar sistemas que sean críticos. Pero para aquellos usuarios que quieran aprovechar dicha flexibilidad, que no necesiten la seguridad que aportan los sistemas de tipado estático, nuestros *success types polimórficos* ayudarán a que se puedan realizar análisis que permitan al programador emplear la disciplina del tipado dinámico, teniendo en cuenta que al ser una sobreaproximación a la semántica de los programas siempre existirá un espacio intermedio indeterminado donde podrían existir errores sin detectar. Así pues, hemos desarrollado un sistema que, respetando el enfoque de los *success types*, y gracias a la aportación del polimorfismo, es capaz de reflejar la semántica de las expresiones con más precisión que los sistemas de *success types* existentes, y por tanto es capaz de detectar de manera estática más situaciones de error sin incurrir en falsos positivos. Con esto se completan los objetivos definidos al comienzo de la investigación en la sección 1.6.

6.2. Trabajo futuro

Siendo importantes los avances obtenidos en nuestra investigación para cubrir las necesidades que *Dialyzer* no nos ofrece, como es el caso de inferir *success types polimórficos*, nuestro trabajo en su estado actual trabaja solo con un subconjunto reducido, pero suficientemente representativo, del lenguaje

Erlang. Sabemos que *Dialyzer* es una herramienta bien diseñada que sus autores han podido integrar en el ciclo de vida de programas reales escritos con *Erlang*, sin dejar de lado ninguna parte del lenguaje, aunque carece de polimorfismo real. Por ello ofrecemos una alternativa prometedora, que en un futuro próximo podría aspirar a reemplazar o a complementar la función que cumple actualmente *Dialyzer*, cuando permita tratar la totalidad del lenguaje *Erlang*.

6.2.1. Extender *Mini Erlang* al lenguaje *Erlang*

La primera línea de trabajo futuro relevante sobre nuestra investigación sería adaptar el sistema de tipos y el algoritmo de inferencia para que puedan analizar programas escritos en *Erlang*. Para lograr este objetivo el camino pasaría por adaptar los sistemas del lenguaje *Mini Erlang* a *Core Erlang*, dado que el propio compilador de *Erlang* puede convertir cualquier programa escrito en *Erlang* a *Core Erlang*. No es solo que la sintaxis de *Core Erlang* sea más sencilla, sino que la semántica del lenguaje está mucho mejor definida [15], por lo que la corrección sería más fácil de abordar que intentando tomar *Erlang* como lenguaje. Una vez alcanzada esa meta se podría contemplar pasar de *Core Erlang* a *Erlang* para no depender de *Core Erlang* como lenguaje intermedio y evitar así los últimos cambios de la implementación interna de la plataforma *Erlang/OTP*.[1]

Abarcar el lenguaje *Erlang* conlleva incorporar aquellas estructuras de datos que no existen en *Mini Erlang*, así como incluir la representación y la gestión de las excepciones dentro de un programa. También existen en *Core Erlang* las expresiones **do** y **catch**, pero las podemos descartar ya que son azúcar sintáctico. Extender el lenguaje para introducir nuevos tipos y las excepciones implica forzosamente cambios en la semántica y por consiguiente también en el sistema de tipos, así como en la corrección del mismo. Por ello, para entender mejor las exigencias que hay que abordar, vamos a detallar de forma un poco más elaborada algunos de los componentes implicados.

En lo relativo a las estructuras de datos, tendríamos que definir una serie de nuevos tipos cuya semántica sirva como subconjunto a **DVal**, incorporando los valores correspondientes a la semántica de las expresiones que definen dichas estructuras. La primera y más sencilla que habría que incorporar son las *secuencias* de *Core Erlang*. Estas se comportan esencialmente igual que las tuplas, pero su diferencia radica en que son un tipo de tuplas especiales que están optimizadas de cara a la compilación. Este detalle sobre su implementación no es relevante de cara al sistema de tipos, por lo que simplemente hay que replicar el comportamiento que tienen las tuplas en nuestro sistema para poder tratar las secuencias, teniendo en cuenta la sintaxis de cada una de ellas. En el caso de los *bitstrings*, lo que se representa es un bloque o segmento de bits que puede tener un tamaño conocido o no. Una vez se encapsula los valores dentro de un *bitstring*, se pierde la información de tipo de los valores contenidos. Sin embargo, al extraer información de un *bitstring* se puede indicar una serie de etiquetas para definir el tipo del valor que se está extrayendo, por lo que se puede deducir la información de tipo que

[1] En las últimas versiones de *Erlang/OTP*, la expresión **receive** ha sido sustituida por llamadas a primitivas internas del lenguaje junto al uso de la expresión **case**.

tendrá la variable usada para el encaje de patrones con un *bitstring*. De las estructuras de datos disponibles en *Erlang*, la más elaborada son los *maps*, que son diccionarios que conectan una clave con un valor. La semántica que se podría usar para representar estos diccionarios, sería la de un grafo, de forma similar a como representamos en nuestra semántica una función con un parámetro de entrada y un valor de retorno. Una particularidad de los diccionarios es que no siempre vamos a poder conocer las claves que contienen a la hora de otorgarles un tipo. Por ello hay que llevar un registro de las claves que sí se conocen para perder la menor precisión posible al analizar este tipo de estructuras. Como efecto colateral, lo aprendido al incorporar los diccionarios al sistema puede ser de ayuda a la hora de incorporar la orientación a objetos en lenguajes de tipado dinámico como *JavaScript* o *Python*, ya que internamente gestionan las instancias de los objetos como diccionarios. En *Erlang* se pueden definir *registros*, que son azúcar sintáctico para manejar tuplas que tienen una forma determinada, por lo que su semántica es la misma que las tuplas, pero se puede incorporar al sistema de tipos una capa de transformación, que tomando un tipo determinado pueda devolver uno que incorpore aquellos alias correspondientes a definiciones de tipos, como es el caso de los registros.

Como hemos señalado previamente, para afrontar esta transformación de *Mini Erlang* a *Erlang* hay que incorporar la gestión de las excepciones, que podrían conformar un conjunto de valores dentro de **DVal** y por lo tanto representar un conjunto de tipos especiales que gestionar de manera específica. Aunque en *Erlang* no se aliente el uso de las excepciones, por su filosofía *"let it crash"* [9], al ser parte de la sintaxis del lenguaje *Core Erlang*, las excepciones tendrían que incorporarse como parte de una herramienta que aspire a analizar programas *Erlang*. Las excepciones serían tratadas como valores especiales dentro de la semántica, aunque se incluyan en **DVal**, y se incorporaría un tipo `exception()` que tuviera parámetros como ocurre con las listas no vacías, para dar tipo al valor que ha sido lanzado con la expresión **throw**. Sin embargo, para facilitar la incorporación de las excepciones al sistema de tipos, cuando analicemos la expresión **throw** su tipo será `none()`, por lo que el tipo de la excepción se adjuntará al resultado del tipo anotado como una tercera componente, extendiendo así los pares $\langle \tau; \Gamma \rangle$. Por lo tanto la semántica ya no devolverá una tupla de sustituciones con valores, sino una tripleta de sustituciones, valores de la ejecución y valores de error posibles. Por ejemplo, si tuviéramos la expresión **throw** `'invarg'`, el tipo que la representaría podría ser $\langle$`none();[];exception('invarg')`$\rangle$. De este modo, cuando lleguemos a una expresión **try-catch**, podremos ir eliminando de la tercera componente aquellas excepciones que puedan ser capturadas por la expresión. Esto nos permitirá saber qué excepciones puede lanzar cada función en un módulo y ofrecer esta información al usuario final, algo que no ofrece las especificaciones de tipos de *TypEr* en su inferencia. Tengamos en cuenta que lenguajes como *Java* ofrece dicha información, relativa a las excepciones que se pueden lanzar, en la cabecera de un método, aportando información valiosa para la documentación de los proyectos.

Cabría debatir si—de cara a dar este paso de *Mini Erlang* a *Erlang*—se podría tomar *Dialyzer* para extender su sistema, reutilizando así una herramienta tan consolidada dentro de la comunidad. En principio, al extender *Dialyzer*, se podría aprovechar el formato existente que ofrece *TypEr* para expresar especificaciones de tipos para las funciones en *Erlang*. La principal ventaja de incorporar este formato de especificaciones ofrecido por *TypEr*, es el poder recopilar información en programas ya

existentes, así como de la biblioteca estándar de *Erlang*. Pero existe un problema con esta aparente ventaja: tendríamos que decidir cómo re-interpretar la semántica de las variables de tipo que hubiera en dichas especificaciones, ya que nuestro sistema tiene distintos tipos de restricciones, mientras que en *Dialyzer* solo hay restricciones de subconjunto y estas no tienen el mismo significado en nuestro sistema de tipos. Es decir, la semántica que propone *Dialyzer* es incompatible con la de nuestro sistema, ya que el tipo escogido para la función `map`—en el módulo de listas—se asemeja a la firma que tendría `map` en el lenguaje *Haskell*:

```
-spec map(fun((A) -> B), [A]) -> [B].
```

Sin embargo, el tipo que obtenemos en las secciones 4.6.2 y 5.7.1 es:

$$(\texttt{any()},\texttt{[]}) \to \texttt{[]} \sqcup$$
$$\forall \alpha,\alpha',\beta,\beta'.((\alpha) \to \beta,\texttt{nelist(}\alpha'\texttt{,[])}) \to \texttt{nelist(}\beta'\texttt{,[])} \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta$$

Esto ocurre porque, si usáramos una firma como la que usa *Dialyzer* con nuestra semántica, el resultado sería que para poder aplicar la función `map` necesitaríamos que el tipo del parámetro de entrada de la función pasada como argumento fuera exactamente el mismo que el del cuerpo de la lista usada en la aplicación de `map`. Si, por ejemplo, aplicáramos `map` con la lista `[1,2,3]` y la función de incrementar en una unidad un número, cuyo tipo sería `(integer())` → `integer()`, la semántica sería vacía porque `integer()` no es igual a $1 \cup 2 \cup 3$. Por lo tanto tendríamos que modificar toda la semántica propuesta por *Dialyzer* y descartar todo su algoritmo de inferencia para integrarlo con nuestro sistema de tipos, no pudiendo reutilizar la mayor parte de la herramienta como resultado.

Una vez completada esta línea y comprobando los resultados obtenidos sobre un conjunto real de programas, estaríamos en condiciones de poder intentar difundir nuestra herramienta dentro de la comunidad de *Erlang*, para que se pudiera considerar un sistema a integrar dentro de la distribución oficial de *Erlang/OTP*.

6.2.2. Comunicación entre procesos en *Erlang*

La segunda línea de trabajo futuro que podría derivar de nuestra investigación consiste en aplicar el análisis estático de tipos orientado a las entradas y salidas de la comunicación entre nodos de *Erlang*. Nuestros resultados están enfocados a las entradas y salidas de las funciones del programa y las interacciones de estas entre sí, pero *Erlang* también es un lenguaje de computación concurrente y distribuida, por lo que es de interés analizar las interacciones entre nodos para detectar errores que se puedan producir con el paso de mensajes. Al contrario que en la línea anterior, extender el sistema para incorporar el concepto de los nodos supondría grandes cambios para el sistema, porque si bien podemos detectar con relativa facilidad qué entradas o salidas tiene una función (rastreando las llamadas a `erlang:send/2` para las salidas y las cláusulas en los `receive` para las entradas), comprobar quién se comunica con quién puede implicar emular la creación y comunicación entre nodos.

Una posibilidad para modelar este sistema sería extender el tipo `pid()` con un tipo que sobreaproxime los términos que admite como mensaje. Para ello hay que analizar el código que está ejecutando el nodo creado para dicho identificador de proceso (PID). De este modo, la función `erlang:send/2`, al recibir un PID, tendría que comprobar que el término a enviar es compatible con el tipo de los mensajes esperados por el proceso en cuestión. También se podría estudiar la incorporación de los llamados *Session Types* [83, 34, 30] que sirven para formalizar modelos de protocolos de comunicación completos, que involucra las comunicaciones con los procesos de la aplicación, pudiendo ser una herramienta de interés de cara a esta línea de investigación. A pesar de las complicaciones, no cabe duda de que es un campo de notable relevancia en un mundo cada vez más dependiente de software de computación distribuida.

6.2.3. Orientación a objetos y lenguajes imperativos

La tercera línea de trabajo futuro tiene que ver con adaptar nuestra investigación para el análisis estático de tipos sobre lenguajes imperativos tales como *JavaScript* o *Python*. En principio, cambiar del paradigma funcional al imperativo para analizar funciones de un programa no supondría un cambio tan drástico en las bases del sistema. Se tendrían que incorporar nuevas reglas para poder analizar y tipar sentencias de control como el caso de los bucles `while` y `for`, por ejemplo. El cambio sustancial sería la gestión de las variables de programa, ya que ahora podrían cambiar de valor durante la ejecución, teniendo diferentes instancias posibles para cada variable e indicando en cada uso a qué instancia estaríamos referenciando para saber el tipo que tuviera en dicho momento. Por ejemplo, en el siguiente código:

```
var x = 1998;
var y = 1999;
console.log(y - x);
x = "Yôko Kanno";
console.log(y - x);
```

no sería suficiente guardar el tipo final de `x`, sino que tenemos que registrar que hay dos instancias para dicha variable: (a) con el tipo unitario `1998` y (b) con el tipo `string`. En la primera resta `x` utiliza la instancia (a) y en la segunda resta usa (b), donde fallaría la ejecución al intentar restar a un número una cadena de texto. No obstante, no siempre es tan sencilla esta gestión de instancias, porque en el caso de los bucles, donde podríamos desconocer cuántas iteraciones se van a producir, tendríamos que realizar un análisis buscando un punto fijo o aplicar un *widening* si no lo alcanzamos. No obstante, cuando analizamos funciones recursivas en *Mini Erlang*, tampoco sabemos de antemano cuántas iteraciones se realizan, y aun así podemos inferir tipos, aplicando *widening* si es necesario. También para el caso de los lenguajes imperativos tendríamos que incorporar la gestión de excepciones como en la primera línea de trabajo, que se resolvería de forma similar. Sí sería un cambio importante incorporar otros elementos como la orientación a objetos, que pasaría por definir nuevos tipos dentro

del sistema para reflejar las clases del programa e incorporar reglas adicionales para poder tipar los accesos y usos de los miembros de un objeto. Un lenguaje conocido de tipado dinámico con orientación a objetos es *Python*. Por ejemplo, si invocamos un método de un objeto, dependiendo de la clase a la que pertenezca el tipo actual del objeto se tendrá que tomar el tipo de una función u otra. Sin embargo, en el caso de *JavaScript*, su orientación a objetos está basada en prototipos, por lo que las clases son azúcar sintáctico y—al acceder a un miembro del objeto instanciado—hay que comprobar si existe el miembro invocado en la instancia actual del tipo `object`.[2] Independientemente de las particularidades de cada lenguaje, las razones que justifican por qué el polimorfismo ayuda a mejorar el análisis de los programas seguirán siendo de aplicación a estos lenguajes imperativos, como asimismo lo serán los fundamentos teóricos de los *success types* polimórficos estudiados en esta tesis.

[2] Las instancias de `object` en *JavaScript* funcionan como si fueran diccionarios de valores asociados a nombres de variables. Al acceder a un miembro del objeto instanciado, si no lo encuentra, en la tabla de la instancia actual, comprueba si está en la tabla apuntada por el campo `prototype`. Este campo especial llamado `prototype` está disponible en todos los objetos instanciados de *JavaScript*.

Bibliografía

[1] Alexander Aiken. Set Constraints: Results, Applications, and Future Directions. In Alan Borning, editor, *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings*, volume 874 of *Lecture Notes in Computer Science*, pages 326–335. Springer, 1994.

[2] Alexander Aiken. Introduction to Set Constraint-Based Program Analysis. *Sci. Comput. Program.*, 35(2):79–111, 1999.

[3] Alexander Aiken and Edward L. Wimmers. Type Inclusion Constraints and Type Inference. In John Williams, editor, *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, pages 31–41. ACM, 1993.

[4] Joseph Albahari and Ben Albahari. *C# 5.0 in a Nutshell - The Definitive Reference: Covers CLR 4.5 and Asynchronous Programming, 5th Edition.* O'Reilly, 2012.

[5] Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 251–270. ACM, 2014.

[6] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 459–472. ACM, 2011.

[7] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: A step towards reconciling dynamically and statically typed OO languages. In Pascal Costanza and Robert Hirschfeld, editors, *Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, October 22, 2007, Montreal, Quebec, Canada*, pages 53–64. ACM, 2007.

[8] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th Eu-*

ropean Conference, Glasgow, UK, July 25-29, 2005, Proceedings, volume 3586 of *Lecture Notes in Computer Science*, pages 428–452. Springer, 2005.

[9] Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.

[10] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[11] European Computer Manufacturers Association. ECMA-262, 12th edition, ECMAScript 2021 Language Specification, 2021. Retrieved Jun. 31, 2021. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[12] Roberto Barbuti and Roberto Giacobazzi. A Bottom-Up Polymorphic Type Inference in Logic Programming. *Sci. Comput. Program.*, 19(3):281–313, 1992.

[13] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 2014.

[14] Luca Cardelli. Type Systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[15] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0.3 language specification, november 2004.

[16] Robert Cartwright and Mike Fagan. Soft Typing. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 278–292. ACM, 1991.

[17] David Castro, Víctor M. Gulías, Clara Benac Earle, Lars-Åke Fredlund, and Samuel Rivas. A Case Study on Verifying a Supervisor Component Using McErlang. *Electron. Notes Theor. Comput. Sci.*, 271:23–40, 2011.

[18] Francesco Cesarini. Which companies are using Erlang, and why?, 2019. Retrieved Oct. 18, 2019. https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus.html.

[19] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *PACMPL*, 1(OOPSLA):48:1–48:30, 2017.

[20] Natalia Chechina, Kenneth MacKenzie, Simon J. Thompson, Phil Trinder, Olivier Boudeville, Viktoria Fordós, Csaba Hoch, Amir Ghaffari, and Mario Moro Hernandez. Evaluating Scalable Distributed Erlang for Scalability and Reliability. *IEEE Trans. Parallel Distributed Syst.*, 28(8):2244–2257, 2017.

[21] Noam Chomsky and Marcel P. Schützenberger. The algebraic theory of context-free languages. *Studies in Logic and the Foundations of Mathematics*, 35:118–161, 1963.

[22] Maria Christakis and Konstantinos Sagonas. Static Detection of Race Conditions in Erlang. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2010.

[23] Matteo Cimini and Jeremy G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 443–455. ACM, 2016.

[24] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[25] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[26] Alain Colmerauer and Philippe Roussel. The Birth of Prolog. In John A. N. Lee and Jean E. Sammet, editors, *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*, pages 37–52. ACM, 1993.

[27] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[28] Luís Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, UK, 1984.

[29] Luís Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982.

[30] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017.

[31] Philip W. Dart and Justin Zobel. A Regular Type Language for Logic Programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 157–187. The MIT Press, 1992.

[32] Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in MLsub. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 60–72. ACM, 2017.

[33] Clara Benac Earle and Lars-Åke Fredlund. Verification of Timed Erlang Programs Using McErlang. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 251–267. Springer, 2012.

[34] Simon Fowler. An Erlang Implementation of Multiparty Session Actors. In Massimo Bartoletti, Ludovic Henrio, Sophia Knight, and Hugo Torres Vieira, editors, *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*, volume 223 of *EPTCS*, pages 36–50, 2016.

[35] Lars-Åke Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 125–136. ACM, 2007.

[36] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. Static type inference for Ruby. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1859–1866. ACM, 2009.

[37] John P. Gallagher and D. Andre de Waal. Fast and Precise Regular Approximations of Logic Programs. In Pascal Van Hentenryck, editor, *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, June 13-18, 1994*, pages 599–613. MIT Press, 1994.

[38] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 429–442. ACM, 2016.

[39] Jean-Yves Girard. The System F of Variable Types, Fifteen Years Later. *Theor. Comput. Sci.*, 45(2):159–192, 1986.

[40] James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[41] Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *Proc. ACM Program. Lang.*, 2(ICFP):71:1–71:32, 2018.

[42] Spyros Hadjichristodoulou. A Gradual Polymorphic Type System with Subtyping for Prolog. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*, pages 451–457. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[43] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. In Donald Sannella, editor, *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 1994.

[44] Joseph Harrison. Runtime type safety for Erlang/OTP behaviours. In Adrian Francalanza and Viktória Fördós, editors, *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2019, Berlin, Germany, August 18, 2019*, pages 36–47. ACM, 2019.

[45] Fred Hebert. *Learn You Some Erlang for Great Good! A Beginner's Guide*. No Starch Press, 2013.

[46] Nevin Heintze and Joxan Jaffar. A Finite Presentation Theorem for Approximating Logic Programs. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 197–209. ACM Press, 1990.

[47] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–55. ACM, 2007.

[48] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell: A Non-strict, Purely Functional Language. *ACM SIGPLAN Notices*, 27(5):1, 1992.

[49] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, 2003.

[50] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. *Proc. ACM Program. Lang.*, 1(ICFP):40:1–40:29, 2017.

[51] Koen Jacobs, Amin Timany, and Dominique Devriese. Fully abstract from static to gradual. *Proc. ACM Program. Lang.*, 5(POPL):1–30, 2021.

[52] Robert Jakob and Peter Thiemann. Towards Tree Automata-based Success Types. *CoRR*, abs/1306.5061, 2013.

[53] Robert Jakob and Peter Thiemann. A Falsification View of Success Typing. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 234–247. Springer, 2015.

[54] Trevor Jim. What Are Principal Typings and What Are They Good For? In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 42–53. ACM Press, 1996.

[55] Miguel Jimenez, Tobias Lindahl, and Konstantinos Sagonas. A language for specifying type contracts in Erlang and its interaction with success typings. In Simon J. Thompson and Lars-Åke Fredlund, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007*, pages 11–17. ACM, 2007.

[56] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[57] Nico Lehmann and Éric Tanter. Gradual refinement types. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 775–788. ACM, 2017.

[58] Rasmus Lerdorf and Kevin Tatroe. *Programming PHP: creating dynamic web pages*. O'Reilly, 2002.

[59] John Levine, Doug Brown, and Tony Mason. *lex & yacc, 2nd Edition*. O'Reilly, 1992.

[60] Tobias Lindahl and Konstantinos Sagonas. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2004.

[61] Tobias Lindahl and Konstantinos Sagonas. TypEr: a type annotator of Erlang code. In Konstantinos Sagonas and Joe Armstrong, editors, *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, Tallinn, Estonia, September 26-28, 2005*, pages 17–25. ACM, 2005.

[62] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In Annalisa Bossi and Michael J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pages 167–178. ACM, 2006.

[63] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011.

[64] Francisco J. López-Fraguas, Manuel Montenegro, and Gorka Suárez-García. A type derivation system for Erlang (Extended version). Technical report, Number 01/17. Dpto. de Sistemas Informáticos y Computación, 2017. Available at https://federwin.sip.ucm.es/sic/investigacion/publicaciones/informes-tecnicos.

[65] Francisco J. López-Fraguas, Manuel Montenegro, and Gorka Suárez-García. Polymorphic success types for Erlang (Extended version). Technical report, Number 03/18. Dpto. de Sistemas Informáticos y Computación, 2018. Available at https://federwin.sip.ucm.es/sic/investigacion/publicaciones/informes-tecnicos.

[66] Francisco Javier López-Fraguas, Manuel Montenegro, and Juan Rodríguez-Hortalá. Polymorphic Types in Erlang Function Specifications. In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan,*

March 4-6, 2016, Proceedings, volume 9613 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2016.

[67] Francisco Javier López-Fraguas, Manuel Montenegro, and Gorka Suárez-García. Polymorphic success types for Erlang. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 515–533. EasyChair, 2018.

[68] Francisco Javier López-Fraguas, Manuel Montenegro, and Gorka Suárez-García. Deriving overloaded success type schemes in Erlang. *J. Comput. Lang.*, 58:100965, 2020.

[69] Lunjin Lu. Polymorphic Type Analysis in Logic Programs by Abstract Intepretation. *J. Log. Program.*, 36(1):1–54, 1998.

[70] Lunjin Lu. A precise type analysis of logic programs. In Maurizio Gabbrielli and Frank Pfenning, editors, *Proceedings of the 2nd international ACM SIGPLAN conference on on Principles and practice of declarative programming, Montreal, Canada, September 20-23, 2000*, pages 214–225. ACM, 2000.

[71] Lunjin Lu. Improving precision of type analysis using non-discriminative union. *Theory Pract. Log. Program.*, 8(1):33–79, 2008.

[72] Lunjin Lu. Inferring precise polymorphic type dependencies in logic programs. In Sergio Antoy and Elvira Albert, editors, *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 143–151. ACM, 2008.

[73] Lunjin Lu. A Polymorphic Type Dependency Analysis for Logic Programs. *New Gener. Comput.*, 29(4):409–444, 2011.

[74] Simon Marlow and Philip Wadler. A Practical Subtyping System For Erlang. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, pages 136–149. ACM, 1997.

[75] Norman Matloff. *Art of R Programming*. No Starch Press, 2011.

[76] Nicholas D. Matsakis and Felix S. Klock II. The Rust Language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014.

[77] Yukihiro Matsumoto. *Ruby in a nutshell - a desktop quick reference*. O'Reilly, 2002.

[78] John McCarthy. LISP: a programming system for symbolic manipulations. In Eric A. Weiss, editor, *Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery, ACM 1959, Cambridge, Massachusetts, USA, September 1-3, 1959*, pages 1:1–1:4. ACM, 1959.

[79] Robin Milner. A Proposal for Standard ML. In Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr., editors, *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, pages 184–197. ACM, 1984.

[80] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.

[81] Cameron Moy, Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Corpse reviver: sound and efficient gradual typing via contract verification. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021.

[82] Alan Mycroft and Richard A. O'Keefe. A Polymorphic Type System for Prolog. *Artif. Intell.*, 23(3):295–307, 1984.

[83] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In Bharat Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[84] Martin R. Neuhäußer and Thomas Noll. Abstraction and Model Checking of Core Erlang Programs in Maude. In Grit Denker and Carolyn L. Talcott, editors, *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 147–163. Elsevier, 2006.

[85] Max S. New and Amal Ahmed. Graduality from embedding-projection pairs. *Proc. ACM Program. Lang.*, 2(ICFP):73:1–73:30, 2018.

[86] Sven-Olof Nyström. A soft-typing system for Erlang. In Bjarne Däcker and Thomas Arts, editors, *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003*, pages 56–71. ACM, 2003.

[87] Sven-Olof Nyström. A subtyping system for Erlang. In *IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020*, 2020.

[88] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, Fourth Edition*. artima, 2020.

[89] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[90] Tommaso Petrucciani. *Polymorphic set-theoretic types for functional languages. (Types ensemblistes polymorphes pour les langages fonctionnels)*. PhD thesis, Sorbonne Paris Cité, France, 2019.

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

[92] Pawel Pietrzak. *A type-based framework for locating errors in constraint logic programs*. PhD thesis, Linköping University Electronic Press, 2002.

[93] Pawel Pietrzak, Jesús Correas, Germán Puebla, and Manuel V. Hermenegildo. A practical type analysis for verification of modular Prolog programs. In Robert Glück and Oege de Moor, editors, *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, pages 61–70. ACM, 2008.

[94] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 481–494. ACM, 2012.

[95] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM, 2015.

[96] John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[97] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[98] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[99] Konstantinos Sagonas. Experience from developing the Dialyzer: A static analysis tool detecting defects in Erlang applications. In *Proceedings of the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools, Chicago, IL, USA, June 12, 2005*. ACM, 2005.

[100] Konstantinos Sagonas. Using Static Analysis to Detect Type Errors and Concurrency Defects in Erlang Programs. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 13–18. Springer, 2010.

[101] Konstantinos Sagonas and Daniel Luna. Gradual typing of Erlang programs: a wrangler experience. In Soon Tee Teoh and Zoltán Horváth, editors, *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, Victoria, BC, Canada, September 27, 2008*, pages 73–82. ACM, 2008.

[102] Konstantinos Sagonas, Josep Silva, and Salvador Tamarit. Precise explanation of success typing errors. In Elvira Albert and Shin-Cheng Mu, editors, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*, pages 33–42. ACM, 2013.

[103] Neil Savage. Gradual evolution. *Commun. ACM*, 57(10):16–18, 2014.

[104] Felipe Bañados Schwerter, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia. Abstracting gradual typing moving forward: precise and space-efficient. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021.

[105] Bhargav Shivkumar, Enrique Naudon, and Lukasz Ziarek. Putting Gradual Types to Work. In José F. Morales and Dominic A. Orchard, editors, *Practical Aspects of Declarative Languages - 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18-19, 2021, Proceedings*, volume 12548 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2021.

[106] Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the Design Space of Higher-Order Casts. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2009.

[107] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the 2006 ACM Scheme and Functional Programming, Portland, OR, USA, September 17, 2006*, pages 81–92. ACM, 2006.

[108] Jeremy G. Siek and Walid Taha. Gradual Typing for Objects. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2007.

[109] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 274–293. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[110] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, 2012.

[111] Jon Skeet. *C# in Depth, Fourth Edition.* Manning, 2019.

[112] Bjarne Stroustrup. *The C++ Programming Language, First Edition.* Addison-Wesley, 1986.

[113] Gorka Suárez-García. Comprobación de modelos en sistemas concurrentes a partir de su semántica en Maude. Master's thesis, Universidad Complutense de Madrid, 2017.

[114] Norihisa Suzuki. Inferring Types in Smalltalk. In John White, Richard J. Lipton, and Patricia C. Goldberg, editors, *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*, pages 187–199. ACM Press, 1981.

[115] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. A unified semantics for future Erlang. In Scott Lystig Fritchie and Konstantinos Sagonas, editors, *Proceedings of the 9th ACM SIGPLAN workshop on Erlang, Baltimore, Maryland, USA, September 30, 2010*, pages 23–32. ACM, 2010.

[116] Nachiappan Valliappan and John Hughes. Typing the wild in Erlang. In Natalia Chechina and Adrian Francalanza, editors, *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, ICFP 2018, St. Louis, MO, USA, September 23-29, 2018*, pages 49–60. ACM, 2018.

[117] Guido van Rossum. Python Programming Language. In Jeff Chase and Srinivasan Seshan, editors, *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007*. USENIX, 2007.

[118] Claudio Vaucheret and Francisco Bueno. More Precise Yet Efficient Type Inference for Logic Programs. In Manuel V. Hermenegildo and Germán Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2002.

[119] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In Andrew P. Black and Laurence Tratt, editors, *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 45–56. ACM, 2014.

[120] Philip Wadler. Theorems for Free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.

[121] Niklaus Wirth. The Programming Language Pascal. *Acta Informatica*, 1:35–63, 1971.

[122] Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. In Robert R. Kessler, editor, *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*, pages 250–262. ACM, 1994.

[123] Eyal Yardeni and Ehud Shapiro. A Type System for Logic Programs. *J. Log. Program.*, 10(2):125–153, 1991.

Apéndice A

Derivación monomórfica de `map`

Aquí se encuentra la derivación completa para el ejemplo de la sección 3.5.2:

$$\dfrac{\begin{array}{c}\Gamma_0 = [\texttt{'*'} : (\texttt{number}(), \texttt{number}()) \xrightarrow{[]} \texttt{number}()] \\ \Gamma_1 = \Gamma_0 \sqcap [\texttt{Map} : (\texttt{any}(), [\texttt{any}()]) \xrightarrow{\Gamma_0} [\texttt{any}()]; \texttt{Foo} : ([\texttt{any}()]) \xrightarrow{\Gamma_0} [\texttt{any}()]] \\ (1) \quad \Gamma_1 \vdash \textbf{fun}(\texttt{V1}, \texttt{V2}) \rightarrow \cdots : (\texttt{any}(), [\texttt{any}()]) \xrightarrow{\Gamma_1} [\texttt{any}()], \Gamma_1 \\ (2) \quad \Gamma_1 \vdash \textbf{fun}(\texttt{V0}) \rightarrow \cdots : ([\texttt{any}()]) \xrightarrow{\Gamma_1} [\texttt{any}()], \Gamma_1 \\ \Gamma_1 \vdash \texttt{Foo} : ([\texttt{any}()]) \xrightarrow{[]} [\texttt{any}()], \Gamma_1\end{array}}{\Gamma_0 \vdash \textbf{letrec}\ \texttt{Map} = \ldots \texttt{Foo} = \ldots \textbf{in} \cdots : ([\texttt{any}()]) \xrightarrow{\Gamma_0} [\texttt{any}()], \Gamma_0}\ [\text{LETREC}]$$

$$\dfrac{\begin{array}{c}(3) \quad \Gamma_1 \vdash \textbf{case}\ \texttt{V2}\ \textbf{of} \ldots \textbf{end} : [\texttt{any}()], (\Gamma_2 = \Gamma_1 \sqcap [\texttt{V1} : \texttt{any}(); \texttt{V2} : [\texttt{any}()]]) \\ \Gamma_2 \backslash \{\texttt{V1}, \texttt{V2}\} = \Gamma_1 \qquad [\texttt{any}()] \backslash \{\texttt{V1}, \texttt{V2}\} = [\texttt{any}()] \\ (1) \quad \Gamma_1 \vdash \textbf{fun}(\texttt{V1}, \texttt{V2}) \rightarrow \cdots : (\texttt{any}(), [\texttt{any}()]) \xrightarrow{\Gamma_1} [\texttt{any}()], \Gamma_1\end{array}}{}\ [\text{ABS}]$$

$$\dfrac{\begin{array}{c}\Gamma_3 = \Gamma_1 \sqcap [\texttt{V1} : \texttt{any}(); \texttt{V2} : []] \\ \Gamma_4 = \Gamma_1 \sqcap [\texttt{V1} : (\texttt{any}()) \xrightarrow{[]} \texttt{any}(); \texttt{V2} : [\texttt{any}()]; X : \texttt{any}(); XS : [\texttt{any}()]] \\ (4) \quad \Gamma_1 \Vdash_{\{\texttt{V2}\}} [] \ \textbf{when}\ \texttt{'true'} \rightarrow [] : [\texttt{any}()], \Gamma_3 \sqcup \Gamma_4 \\ (5) \quad \Gamma_1 \Vdash_{\{\texttt{V2}\}} [X \mid XS]\ \textbf{when}\ \texttt{'true'} \rightarrow \cdots : [\texttt{any}()], \Gamma_3 \sqcup \Gamma_4\end{array}}{(3) \quad \Gamma_1 \vdash \textbf{case}\ \texttt{V2}\ \textbf{of} \ldots \textbf{end} : [\texttt{any}()], \Gamma_2}\ [\text{CASE}]$$

$$\dfrac{\begin{array}{c}\Gamma_1 \vdash [] : [], \Gamma_1 \qquad \Gamma_1 \sqcap [\texttt{V2} : []] \vdash \texttt{'true'} : \texttt{'true'}, \Gamma_3 \\ \texttt{'true'} \sqcap true \neq \texttt{none}() \qquad \Gamma_3 \vdash [] : [\texttt{any}()], \Gamma_3 \sqcup \Gamma_4\end{array}}{(4) \quad \Gamma_1 \Vdash_{\{\texttt{V2}\}} [] \ \textbf{when}\ \texttt{'true'} \rightarrow [] : [\texttt{any}()], \Gamma_3 \sqcup \Gamma_4}\ [\text{CLS-T}]$$

$$\dfrac{\begin{array}{c}\Gamma_1 \vdash [X \mid XS] : [\texttt{any}() \mid \texttt{any}()], \Gamma_1 \\ (\Gamma_1 \sqcap [\texttt{V2} : [\texttt{any}() \mid \texttt{any}()]] = \Gamma_5) \vdash \texttt{'true'} : \texttt{'true'}, \Gamma_5 \\ \texttt{'true'} \sqcap true \neq \texttt{none}() \qquad \Gamma_5 \vdash \textbf{let}\ \texttt{V3} = \ldots \textbf{in} \cdots : [\texttt{any}()], \Gamma_3 \sqcup \Gamma_4 \quad (6)\end{array}}{(5) \quad \Gamma_1 \Vdash_{\{\texttt{V2}\}} [X \mid XS]\ \textbf{when}\ \texttt{'true'} \rightarrow \cdots : [\texttt{any}()], \Gamma_3 \sqcup \Gamma_4}\ [\text{CLS-T}]$$

$$\frac{\begin{array}{l}(7)\quad \Gamma_5 \vdash \texttt{V1}(X) : \texttt{any}(), (\Gamma_6 = \Gamma_5 \sqcap [\texttt{V1} : (\texttt{any}()) \xrightarrow{[]} \texttt{any}(); X : \texttt{any}()]) \\ (8)\quad \Gamma_6 \sqcap [\texttt{V3} : \texttt{any}()] \vdash \textbf{let}\ \texttt{V4} = \ldots \textbf{in} \cdots : [\texttt{any}()], \Gamma_4 \sqcap [\texttt{V3} : \texttt{any}()]\end{array}}{(6)\quad \Gamma_5 \vdash \textbf{let}\ \texttt{V3} = \ldots \textbf{in} \cdots : [\texttt{any}()], \Gamma_4}\ [\text{LET}]$$

$$\frac{\Gamma_5(\texttt{V1}) \sqcap ((\texttt{any}()) \xrightarrow{[]} \texttt{any}()) = (\texttt{any}()) \xrightarrow{[]} \texttt{any}()}{(7)\quad \Gamma_5 \vdash \texttt{V1}(X) : \texttt{any}(), \Gamma_6}\ [\text{APP1}]$$

$$\frac{\begin{array}{l}(9)\quad \Gamma_6 \sqcap [\texttt{V3} : \texttt{any}()] \vdash \texttt{Map}(\texttt{V1}, XS) : [\texttt{any}()], (\Gamma_7 = \Gamma_6 \sqcap \\ \qquad [\texttt{V3} : \texttt{any}(); \texttt{V1} : \texttt{any}(); XS : [\texttt{any}()]] = \Gamma_4 \sqcap [\texttt{V3} : \texttt{any}()]) \\ (10)\quad (\Gamma_7 \sqcap [\texttt{V4} : [\texttt{any}()]] = \Gamma_8) \vdash [\texttt{V3} \mid \texttt{V4}] : [\texttt{any}()], \Gamma_8\end{array}}{(8)\quad \Gamma_6 \sqcap [\texttt{V3} : \texttt{any}()] \vdash \textbf{let}\ \texttt{V4} = \ldots \textbf{in} \cdots : [\texttt{any}()], \Gamma_4 \sqcap [\texttt{V3} : \texttt{any}()]}\ [\text{LET}]$$

$$\frac{\Gamma_5(\texttt{Map}) \sqcap ((\texttt{any}(), \texttt{any}()) \xrightarrow{[]} \texttt{any}()) = (\texttt{any}(), [\texttt{any}()]) \xrightarrow{\Gamma_0} [\texttt{any}()]}{(9)\quad \Gamma_6 \sqcap [\texttt{V3} : \texttt{any}()] \vdash \texttt{Map}(\texttt{V1}, XS) : [\texttt{any}()], \Gamma_7}\ [\text{APP1}]$$

$$\frac{\Gamma_8 \vdash \texttt{V3} : \texttt{any}(), \Gamma_8 \qquad \Gamma_8 \vdash \texttt{V4} : [\texttt{any}()], \Gamma_8}{(10)\quad \Gamma_8 \vdash [\texttt{V3} \mid \texttt{V4}] : [\texttt{any}()], \Gamma_8}\ [\text{LIST}]$$

$$\frac{\begin{array}{l}(11)\quad \Gamma_1 \vdash \textbf{let}\ \texttt{V5} = \ldots \textbf{in} \cdots : [\texttt{any}()], (\Gamma_9 = \Gamma_1 \sqcap [\texttt{V0} : [\texttt{any}()]]) \\ \qquad \Gamma_9 \backslash \{\texttt{V0}\} = \Gamma_1 \qquad [\texttt{any}()] \backslash \{\texttt{V0}\} = [\texttt{any}()]\end{array}}{(2)\quad \Gamma_1 \vdash \textbf{fun}(\texttt{V0}) \rightarrow \cdots : ([\texttt{any}()]) \xrightarrow{\Gamma_1} [\texttt{any}()], \Gamma_1}\ [\text{ABS}]$$

$$\frac{\begin{array}{l}(12)\quad \Gamma_1 \vdash \textbf{fun}(\texttt{V6}) \rightarrow \cdots : (\texttt{number}()) \xrightarrow{\Gamma_1} \texttt{number}(), \Gamma_1 \\ (13)\quad (\Gamma_1 \sqcap [\texttt{V5} : (\texttt{number}()) \xrightarrow{\Gamma_1} \texttt{number}()] = \Gamma_{10}) \vdash \texttt{Map}(\texttt{V5}, \texttt{V0}) : \\ \qquad [\texttt{any}()], (\Gamma_{11} = \Gamma_9 \sqcap [\texttt{V5} : (\texttt{number}()) \xrightarrow{\Gamma_1} \texttt{number}()])\end{array}}{(11)\quad \Gamma_1 \vdash \textbf{let}\ \texttt{V5} = \ldots \textbf{in} \cdots : [\texttt{any}()], \Gamma_9}\ [\text{LET}]$$

$$\frac{\begin{array}{l}(14)\quad \Gamma_1 \vdash \textbf{let}\ \texttt{V7} = \ldots \textbf{in} \cdots : \texttt{number}(), (\Gamma_{12} = \Gamma_1 \sqcap [\texttt{V6} : \texttt{number}()]) \\ \qquad \Gamma_{12} \backslash \{\texttt{V6}\} = \Gamma_1 \qquad \texttt{number}() \backslash \{\texttt{V6}\} = \texttt{number}()\end{array}}{(12)\quad \Gamma_1 \vdash \textbf{fun}(\texttt{V6}) \rightarrow \cdots : (\texttt{number}()) \xrightarrow{\Gamma_1} \texttt{number}(), \Gamma_1}\ [\text{ABS}]$$

$$\frac{\begin{array}{l}\Gamma_1 \vdash \texttt{2} : \texttt{2}, \Gamma_1 \qquad \Gamma_{13} = \Gamma_1 \sqcap [\texttt{V7} : \texttt{number}()] \\ (15)\quad \Gamma_{13} \vdash \texttt{'*'}(\texttt{V6}, \texttt{V7}) : \texttt{number}(), (\Gamma_{14} = \Gamma_{12} \sqcap [\texttt{V7} : \texttt{number}()])\end{array}}{(14)\quad \Gamma_1 \vdash \textbf{let}\ \texttt{V7} = \ldots \textbf{in} \cdots : \texttt{number}(), \Gamma_{12}}\ [\text{LET}]$$

$$\frac{\Gamma_{13}(\texttt{'*'}) \sqcap ((\texttt{any}(), \texttt{any}()) \xrightarrow{[]} \texttt{any}()) = (\texttt{number}(), \texttt{number}()) \xrightarrow{[]} \texttt{number}()}{(15)\quad \Gamma_{13} \vdash \texttt{'*'}(\texttt{V6}, \texttt{V7}) : \texttt{number}(), \Gamma_{14}}\ [\text{APP1}]$$

$$\frac{\Gamma_{10}(\texttt{Map}) \sqcap ((\texttt{any}(), \texttt{any}()) \xrightarrow{[]} \texttt{any}()) = (\texttt{any}(), [\texttt{any}()]) \xrightarrow{\Gamma_0} [\texttt{any}()]}{(13)\quad \Gamma_{10} \vdash \texttt{Map}(\texttt{V5}, \texttt{V0}) : [\texttt{any}()], \Gamma_{11}}\ [\text{APP1}]$$

Apéndice B

Operaciones con success types polimórficos

B.1. Relación de subtipado entre tipos *ground*

La siguiente operación que vamos a mostrar es una aproximación a la relación de subtipado entre tipos *ground* dentro de nuestros tipos polimórficos, es decir, aquellos que no contienen variables de tipo en su interior, ni libres, ni cerradas. Dados dos tipos *ground* τ y τ', usaremos la notación $\tau \dot{\subseteq} \tau'$ para representar esta aproximación a la relación de subtipado entre tipos *ground*, que definiremos de la siguiente manera:

1. $\tau \dot{\subseteq} \tau = \textbf{true}$
2. $\tau \dot{\subseteq} \texttt{any()} = \textbf{true}$
3. $\texttt{none()} \dot{\subseteq} \tau = \textbf{true}$
4. $c \dot{\subseteq} B = \begin{cases} \textbf{true} & c \in \mathscr{B}[\![B]\!] \\ \textbf{false} & c \notin \mathscr{B}[\![B]\!] \end{cases}$
5. $B \dot{\subseteq} B' = \mathscr{B}[\![B]\!] \subseteq \mathscr{B}[\![B']\!]$
6. $\{\overline{\tau_i}^n\} \dot{\subseteq} \{\overline{\tau'_i}^n\} = \bigwedge_{i=1}^n \tau_i \dot{\subseteq} \tau'_i$
7. $\begin{aligned} \texttt{nelist}(\tau_1,\tau_2) \dot{\subseteq} \texttt{nelist}(\tau_3,\tau_4) &= \left(\tau_1 \dot{\subseteq} \tau_3 \wedge \tau_2 \dot{\subseteq} \tau_4\right) \\ &\vee \left(\tau_1 \dot{\subseteq} \tau_3 \wedge \tau_2 \dot{\subseteq} \texttt{nelist}(\tau_3,\tau_4)\right) \end{aligned}$
8. $\left(\overline{\tau_i}^n\right) \to \tau \dot{\subseteq} \left(\overline{\tau'_i}^n\right) \to \tau' = \bigwedge_{i=1}^n \tau_i \dot{\subseteq} \tau'_i \wedge \tau \dot{\subseteq} \tau'$

9. $\tau \dot{\subseteq} (\tau' \textbf{ when } C) = \begin{cases} \tau \dot{\subseteq} \tau' & success(C) \\ \textbf{false} & \neg success(C) \end{cases}$

10. $(\tau \textbf{ when } C) \dot{\subseteq} \tau' = \begin{cases} \tau \dot{\subseteq} \tau' & success(C) \\ \textbf{true} & \neg success(C) \end{cases}$

11. $(\tau \cup \tau') \dot{\subseteq} \tau'' = \tau \dot{\subseteq} \tau'' \wedge \tau' \dot{\subseteq} \tau''$

12. $\bigsqcup_{i=1}^{n} \sigma_i \dot{\subseteq} \tau = \bigwedge_{i=1}^{n} \sigma_i \dot{\subseteq} \tau$

13. $\tau \dot{\subseteq} (\tau' \cup \tau'') = \tau \dot{\subseteq} \tau' \vee \tau \dot{\subseteq} \tau''$

14. $\tau \dot{\subseteq} \bigsqcup_{i=1}^{n} \sigma_i = \bigvee_{i=1}^{n} \tau \dot{\subseteq} \sigma_i$

15. Cualquier otro caso: **false**

Como se puede apreciar en la operación, tenemos casos donde hay tipos con una condición **when**. En el hipotético caso de que encontráramos un conjunto de restricciones asociadas a un tipo, solo se podrían tener restricciones de la forma $c \subseteq \tau$, donde c es un literal y τ un tipo *ground*, por lo que se podría comprobar si se cumplen las restricciones o no. Para ello tenemos una función *success* que recibe un conjunto $C = \{\overline{\varphi_i}^n\}$, que se define del siguiente modo:

$$\begin{aligned} success(\{\overline{\varphi_i}^n\}) &= \bigwedge_{i=1}^{n} success(\varphi_i) \\ success(c \subseteq \tau) &= c \dot{\subseteq} \tau \end{aligned}$$

De esta manera, si tenemos un τ **when** C y se cumple $success(C)$, tendremos τ como tipo final. Pero si no se cumple $success(C)$, el tipo final sería `none()`. Por ello al hacer la comprobación de subtipado, cuando nos encontramos con un tipo condicional, lo que estaremos haciendo como paso intermedio es ver si lo sustituimos por el tipo en sí o por `none()`.

Por último, indiquemos la propiedad que esta aproximación a la relación de subtipado entre tipos *ground* deberá cumplir:

Proposición 34. *Para cada par de tipos* ground τ_1 *y* τ_2*, es cierto que* $\tau_1 \dot{\subseteq} \tau_2$ *si y solo si* $\mathcal{T}[\![\tau_1]\!] \subseteq \mathcal{T}[\![\tau_2]\!]$.

B.2. Relación de subtipado entre tipos polimórficos

Dados dos tipos τ y τ', tales que $vars(\tau) \cap vars(\tau') = \emptyset$, usaremos la notación $\tau \overset{\circ}{\subseteq} \tau'$ para representar una aproximación a la relación de subtipado entre tipos. En caso de haber colisión entre variables de tipo, se puede renombrar uno de los dos tipos para hacer la operación. La operación está dividida en tres fases:

1. Transformamos τ y τ' en sus respectivos tipos *ground*, τ_g y τ'_g, para comprobar con $\tau_g \dot{\subseteq} \tau'_g$ que se cumple que el primero es subtipo del segundo.

2. Colapsamos los cuerpos de los tipos `nelist` que tengamos anidados dentro de cada uno de los tipos de entrada, unificando los mismos patrones en los diferentes fragmentos de cada tipo.

3. Tomamos τ y τ', para descomponerlo en una serie de relaciones de conexión—entre fragmentos de cada tipo—para comprobar que se mantiene el polimorfismo del operando derecho en el izquierdo.

B.2.1. Colapsando las listas no vacías

Necesitaremos para el algoritmo una función total $\Upsilon : \mathbf{TypeVar} \to \mathscr{P}(\mathbf{TypeVar})$. Usamos la notación $[]$ para denotar aquella función donde $\Upsilon(\alpha) = \emptyset$ para toda $\alpha \in \mathbf{TypeVar}$. Con $\Upsilon\left[\overline{\alpha_i : A_i}^n\right]$ denotamos una función Υ' tal que:

$$\Upsilon'(\alpha) = \begin{cases} A_i & \text{si } \alpha = \alpha_i \\ \Upsilon(\alpha) & \text{e.o.c.} \end{cases}$$

La notación $\left[\overline{\alpha_i : A_i}^n\right]$ es un abuso de notación para indicar $[]\left[\overline{\alpha_i : A_i}^n\right]$. Con esta función vamos a registrar las variables de tipo que vamos a usar para sustituir a un conjunto de ellas, dentro de los cuerpos de las listas no vacías.

Hemos de tener en cuenta que estamos trabajando con esquemas de tipos funcionales, por lo que todas las restricciones que generemos en el proceso irán en el **when** del esquema de tipo funcional actual en el que hayamos entrado. Entonces nuestro Υ de partida será $[]$ e iremos recorriendo el árbol sintáctico abstracto del tipo de entrada, para buscar exactamente este patrón:

$$\texttt{nelist}(\alpha_1, \cdots \texttt{nelist}(\alpha_n, \tau) \cdots) \qquad \text{donde } \tau \text{ no es un tipo } \texttt{nelist}$$

Es decir, buscamos una serie de listas no vacías anidadas cuyos cuerpos sean variables de tipo exclusivamente. Así que teniendo un conjunto $A = \{\overline{\alpha_i}^n\}$, si existe una α tal que $\Upsilon(\alpha) = A$, entonces sustituimos el tipo con $\texttt{nelist}(\alpha, \tau)$. Si no existe en Υ el conjunto A, entonces tomaremos una variable fresca δ, para construir una nueva función Υ' tal que $\Upsilon\,[\delta : A]$, y con ello sustituir el tipo con $\texttt{nelist}(\delta, \tau)$.

Cuando se vaya a salir del último **when** del esquema de tipo funcional que hayamos entrado, tendremos por un lado la Υ que se recibió al entrar en el tipo funcional y por el otro una Υ' que será la última función modificada, que Υ' será igual a $\Upsilon\left[\overline{\alpha_i : A_i}^n\right]$. Con esta información hemos de crear las siguientes restricciones:

$$\bigcup_{i=1}^{n} \{\alpha \subseteq \alpha_i \mid \alpha \in A_i\}$$

Estas restricciones las añadiremos a las existentes en el **when** y con ellos se finaliza el colapso de las listas no vacías.

Veamos ahora, brevemente, para qué necesitamos hacer esto con un ejemplo. Supongamos que queremos comparar los siguientes tipos:

$$\sigma_1 = \forall \alpha_1.(\texttt{nelist(}\alpha_1\texttt{,[])}) \to \texttt{nelist(}\alpha_1\texttt{,[])}$$
$$\sigma_2 = \forall \beta_1, \beta_2.\big(\texttt{nelist(}\beta_1\texttt{,nelist(}\beta_2\texttt{,[]))}\big) \to \texttt{nelist(}\beta_1\texttt{,nelist(}\beta_2\texttt{,[])}$$

donde la comparación es $\sigma_2 \overset{\circ}{\subseteq} \sigma_1$. Supongamos que hemos descompuesto los tipos para encontrar las conexiones, obteniendo que $\beta_1 \leftharpoondown \alpha_1$ y $\beta_2 \leftharpoondown \alpha_1$, que significa que α_1 tiene que estar conectada con β_1 y β_2. En esta situación no podemos garantizar que β_1 y β_2 vaya a tener la misma instanciación que α_1, con lo que daríamos como falso que esté contenido σ_2 en σ_1, cosa que sabemos que no es verdad. Por ello necesitamos colapsar β_1 y β_2 en una δ en todos aquellos puntos donde aparezca ese patrón.

B.2.2. Descomposición del tipo

La descomposición de los tipos—en la aproximación a la relación de subtipado—estará representada por la función *get_formula*$\big(\tau \overset{\circ}{\subseteq} \tau'\big)$, donde τ y τ' son los tipos de entrada. El resultado de la función es una disyunción de relaciones, que tendrán la siguiente sintaxis:

$$Q ::= Q \vee Q' \mid \big(P; C; C'\big) \qquad\qquad P ::= P \wedge P' \mid \tau \leftharpoondown \tau' \mid \textbf{true} \mid \textbf{false}$$

Tenemos Q que es una disyunción de tripletas, donde P es una conjunción de relaciones que buscan comprobar si se mantiene la conexión del polimorfismo de derecha a izquierda, luego C son las restricciones encontradas en el τ de entrada, de la aproximación a la relación de subtipado, y C' las encontradas en el τ' de entrada. Los elementos que conforman P son valores booleanos por un lado y la relaciones de conexión $\tau \leftharpoondown \tau'$, que indica que τ' ha de estar conectado con τ en lo relativo al polimorfismo. Para generar las relaciones de conexión tenemos que usar las siguientes reglas:

$$\frac{}{\tau \overset{\circ}{\subseteq} \texttt{any()} \vdash (\textbf{true}; \emptyset; \emptyset)}\ [\text{ANY}] \qquad \frac{}{\texttt{none()} \overset{\circ}{\subseteq} \tau \vdash (\textbf{true}; \emptyset; \emptyset)}\ [\text{NONE}]$$

$$\frac{\tau_1 \overset{\circ}{\subseteq} \tau_3 \vdash Q_1 \qquad \tau_2 \overset{\circ}{\subseteq} \tau_4 \vdash Q_2 \qquad \tau_2 \overset{\circ}{\subseteq} \texttt{nelist(}\tau_3\texttt{,}\tau_4\texttt{)} \vdash Q_3}{\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)} \overset{\circ}{\subseteq} \texttt{nelist(}\tau_3\texttt{,}\tau_4\texttt{)} \vdash (Q_1 \odot Q_2) \vee (Q_1 \odot Q_3)}\ [\text{NELIST}]$$

$$\frac{\forall i \in \{1..n\}.\tau_i \overset{\circ}{\subseteq} \tau'_i \vdash Q_i}{\{\overline{\tau_i}^n\} \overset{\circ}{\subseteq} \{\overline{\tau'_i}^n\} \vdash \bigodot_{i=1}^n Q_i}\ [\text{TUPLE}] \qquad \frac{\forall i \in \{1..n\}.\tau_i \overset{\circ}{\subseteq} \tau'_i \vdash Q_i \qquad \tau \overset{\circ}{\subseteq} \tau' \vdash Q}{\big(\overline{\tau_i}^n\big) \to \tau \overset{\circ}{\subseteq} \big(\overline{\tau'_i}^n\big) \to \tau' \vdash \bigodot_{i=1}^n Q_i \odot Q}\ [\text{FUN}]$$

$$\frac{\tau \overset{\circ}{\subseteq} \tau' \vdash Q}{\forall \overline{\alpha_i}.\tau \overset{\circ}{\subseteq} \tau' \vdash Q}\ [\text{SCHEME-L}] \qquad \frac{\tau \overset{\circ}{\subseteq} \tau' \vdash \bigvee_{i=1}^n \big(P_i; C_i; C'_i\big)}{(\tau\ \textbf{when}\ C) \overset{\circ}{\subseteq} \tau' \vdash \bigvee_{i=1}^n \big(P_i; C_i \cup C; C'_i\big)}\ [\text{WHEN-L}]$$

$$\frac{\tau \overset{\circ}{\subseteq} \tau' \vdash Q}{\tau \overset{\circ}{\subseteq} \forall \overline{\alpha_i}.\tau' \vdash Q}\ [\text{SCHEME-R}] \qquad \frac{\tau \overset{\circ}{\subseteq} \tau' \vdash \bigvee_{i=1}^{n} \left(P_i; C_i; C_i'\right)}{\tau \overset{\circ}{\subseteq} \left(\tau' \textbf{ when } C\right) \vdash \bigvee_{i=1}^{n} \left(P_i; C_i; C_i' \cup C\right)}\ [\text{WHEN-R}]$$

$$\frac{\tau \overset{\circ}{\subseteq} \tau'' \vdash Q \qquad \tau' \overset{\circ}{\subseteq} \tau'' \vdash Q'}{\left(\tau \cup \tau'\right) \overset{\circ}{\subseteq} \tau'' \vdash Q \odot Q'}\ [\text{UNION-L}] \qquad \frac{\forall i \in \{1..n\}.\sigma_i \overset{\circ}{\subseteq} \tau \vdash Q_i}{\bigsqcup_{i=1}^{n} \sigma_i \overset{\circ}{\subseteq} \tau \vdash \bigodot_{i=1}^{n} Q_i}\ [\text{SEQ-L}]$$

$$\frac{\tau \overset{\circ}{\subseteq} \tau' \vdash Q \qquad \tau \overset{\circ}{\subseteq} \tau'' \vdash Q'}{\tau \overset{\circ}{\subseteq} \left(\tau' \cup \tau''\right) \vdash Q \vee Q'}\ [\text{UNION-R}] \qquad \frac{\forall i \in \{1..n\}.\tau \overset{\circ}{\subseteq} \sigma_i \vdash Q_i}{\tau \overset{\circ}{\subseteq} \bigsqcup_{i=1}^{n} \sigma_i \vdash \bigvee_{i=1}^{n} Q_i}\ [\text{SEQ-R}]$$

$$\frac{}{\alpha \overset{\circ}{\subseteq} \tau \vdash (\alpha \longleftarrow \tau; \emptyset; \emptyset)}\ [\text{ALPHA-L}] \qquad \frac{}{\tau \overset{\circ}{\subseteq} \alpha \vdash (\tau \longleftarrow \alpha; \emptyset; \emptyset)}\ [\text{ALPHA-R}]$$

$$\frac{vars(\tau) = \emptyset \qquad vars\left(\tau'\right) = \emptyset}{\tau \overset{\circ}{\subseteq} \tau' \vdash \left(\tau \overset{\bullet}{\subseteq} \tau'; \emptyset; \emptyset\right)}\ [\text{MONO}] \qquad \frac{}{\tau \overset{\circ}{\subseteq} \tau' \vdash (\textbf{false}; \emptyset; \emptyset)}\ [\text{FAIL}]$$

Estas reglas tienen un orden de preferencia, para poder ser aplicadas a la hora de generar las relaciones de conexión, que es el siguiente:

1. MONO
2. ANY, NONE
3. NELIST, TUPLE, FUN, SCHEME-L, SCHEME-R, WHEN-L, WHEN-R
4. UNION-L, SEQ-L
5. UNION-R, SEQ-R
6. ALPHA-L, ALPHA-R
7. FAIL

Por último, en las reglas vemos que estamos utilizando el operador $\odot$, que se trata de una operación conmutativa, y que vamos a definir de la siguiente manera:

$$Q_L \odot Q_R = \begin{cases} \left(P_L \wedge P_R; C_L \cup C_R; C_L' \cup C_R'\right) & \textit{if } Q_L = \left(P_L; C_L; C_L'\right) \wedge Q_R = \left(P_R; C_R; C_R'\right) \\ \left(Q_L \odot Q_R\right) \vee \left(Q_L' \odot Q_R\right) & \textit{if } Q_L = Q_L \vee Q_L' \\ \left(Q_L \odot Q_R\right) \vee \left(Q_L \odot Q_R'\right) & \textit{if } Q_R = Q_R \vee Q_R' \end{cases}$$

B.2.3. Comprobación de las relaciones

Después de la descomposición, tenemos que resolver la Q obtenida con la función $solve\left(\tau \overset{\circ}{\subseteq} \tau'; Q\right)$, donde τ y τ' son los tipos de entrada iniciales de la aproximación a la relación de subtipado. La función

solve devolverá **true** o **false** como resultado de la operación y la definimos de la siguiente manera:

$$solve\left(\tau \overset{\circ}{\subseteq} \tau'; Q \vee Q'\right) = solve\left(\tau \overset{\circ}{\subseteq} \tau'; Q\right) \vee solve\left(\tau \overset{\circ}{\subseteq} \tau'; Q'\right)$$
$$solve\left(\tau \overset{\circ}{\subseteq} \tau'; \left(P; C; C'\right)\right) = is\_connected\left(\tau \overset{\circ}{\subseteq} \tau'; P; C; C'\right)$$

Luego con la función *is_connected*, para cada tripleta $\left(P; C; C'\right)$ dentro de Q, tendremos que comprobar si se respetan las conexiones entre variables de tipo. Para ello, tendremos que transformar las relaciones en P, con una serie de reglas que vamos a definir a continuación, hasta encontrar un estado en el que podamos hacer la comprobación final. Esta transformación finalizará cuando lleguemos al valor **true** o **false**, obteniendo así el resultado de la función. Las reglas para transformar P son:

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \textbf{true}; C; C' \Longrightarrow \textbf{true}} \text{ [END]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \textbf{true} \wedge P; C; C' \Longrightarrow P; C; C'} \text{ [TRUE]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \textbf{false} \wedge P; C; C' \Longrightarrow \textbf{false}} \text{ [FALSE]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash P; C \cup \{\varphi\}; C' \Longrightarrow P; C; C'} \text{ [DELETE-L]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash P; C; C' \cup \{\varphi\} \Longrightarrow P; C; C'} \text{ [DELETE-R]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash P; C \cup \{\alpha \Leftarrow \beta\}; C' \Longrightarrow P; C \cup \{\beta \Leftarrow \alpha\}; C'} \text{ [TURN-L]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash P; C; C' \cup \{\alpha \Leftarrow \beta\} \Longrightarrow P; C; C' \cup \{\beta \Leftarrow \alpha\}} \text{ [TURN-R]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \bigwedge_{i=1}^{n} \tau_i \leftarrowtail \tau_i'; C \cup \{\tau \Leftarrow \alpha\}; C' \Longrightarrow \bigwedge_{i=1}^{n} \tau_i\,[\alpha/\tau] \leftarrowtail \tau_i'; C \cup \{\tau \Leftarrow \alpha\}; C'} \text{ [REPLACE-LP]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \bigwedge_{i=1}^{n} \tau_i \leftarrowtail \tau_i'; C; C' \cup \{\tau \Leftarrow \alpha\} \Longrightarrow \bigwedge_{i=1}^{n} \tau_i \leftarrowtail \tau_i'\,[\alpha/\tau]; C; C' \cup \{\tau \Leftarrow \alpha\}} \text{ [REPLACE-RP]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash P; C \cup \{\beta \Leftarrow \alpha\}; C' \Longrightarrow P; C\left[\alpha/\beta\right] \cup \{\beta \Leftarrow \alpha\}; C'} \text{ [REPLACE-LC]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash P; C; C' \cup \{\beta \Leftarrow \alpha\} \Longrightarrow P; C; C'\left[\alpha/\beta\right] \cup \{\beta \Leftarrow \alpha\}} \text{ [REPLACE-RC]}$$

$$\frac{fvs\left(\tau'\right) = \emptyset}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \alpha \leftarrowtail \tau' \wedge P; C; C' \Longrightarrow P; C; C'} \text{ [REMOVE-1]}$$

$$\frac{\beta \notin ftv(P) \cup ftv\left(C'\right)}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \tau \leftarrowtail \beta \wedge P; C; C' \Longrightarrow P; C; C'} \text{ [REMOVE-2]}$$

$$\frac{\tau_1 \neq \tau_2 \qquad \left(fvs(\tau_1) \cup fvs(\tau_2)\right) \cap fvs(C) = \emptyset \qquad ftv\left(\tau'\right) \neq \emptyset}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \tau_1 \leftarrowtail \tau' \wedge \tau_2 \leftarrowtail \tau' \wedge P; C; C' \Longrightarrow \textbf{false}} \text{ [MANY]}$$

$$\frac{}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \alpha \leftharpoondown \tau' \wedge \alpha \leftharpoondown \tau' \wedge P; C; C' \Longrightarrow \alpha \leftharpoondown \tau' \wedge P; C; C'} \text{ [SAME]}$$

$$\frac{\beta \notin ftv(P) \qquad P' = not\left(is\_plural(\tau_0; \alpha) \wedge is\_singular\left(\tau_0'; \beta\right)\right)}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \alpha \leftharpoondown \beta \wedge P; C; C' \Longrightarrow P' \wedge P; C; C'} \text{ [CHECK]}$$

$$\frac{\tau_1 \neq \alpha_1 \qquad \tau_2 \neq \alpha_2 \qquad P' = solve\left(\tau_0 \overset{\circ}{\subseteq} \tau_0'; (P; C; C') \odot get\_formula\left(\tau_1 \overset{\circ}{\subseteq} \tau_2\right)\right)}{\tau_0 \overset{\circ}{\subseteq} \tau_0' \vdash \tau_1 \leftharpoondown \tau_2 \wedge P; C; C' \Longrightarrow P'} \text{ [EXPAND]}$$

Vistas las reglas de transformación, tenemos que elegir una estrategia a seguir para aplicarlas y poder llegar al resultado final que nos diga si es cierta la inclusión, o si por en contrario no podemos garantizar que sea cierta.[1] Siempre que lleguemos a un valor, **true** o **false**, el algoritmo terminará. Si no se pudiera aplicar ninguna regla, por llegar a un punto muerto, devolveremos como resultado **false**. A continuación vamos a enumerar los pasos para aplicar las reglas.

1. Aplicamos [REMOVE-1] para quitar todas las relaciones de conexión que son inútiles para la comprobación, ya que en el lado derecho hay un tipo monomórfico y eso no nos sirve para comprobar el polimorfismo.

2. Aplicamos [REMOVE-2] para quitar aquellas relaciones de conexión únicas, cuya variable de tipo en la derecha no existe en P, ni en C', con lo que no queda ninguna transformación que poder hacer con ella y por consiguiente es cierta la conexión.

3. Aplicamos [DELETE-L] y [DELETE-R], eliminando toda aquella restricción que no sea de encaje, ya que no sirven para hacer sustituciones y poder así transformar los tipos que tenemos en las relaciones. También eliminaremos aquellas restricciones de encaje $\tau \Leftarrow \alpha$ donde τ sea un tipo monomórfico, porque esta información no nos aportará nada útil de cara a la comprobación.

4. Aplicamos [DELETE-L] y [DELETE-R], en una segunda pasada, para eliminar aquellas restricciones de encaje $\alpha \Leftarrow \alpha$ donde alguna de las dos variables de tipo no vuelve a aparecer en P, ni tampoco en su respectivo C o C'. Esta situación lo que quiere decirnos es que estaríamos, dentro del grafo de igualdades entre variables de tipo, en una rama que está suelta y no es útil hacer ninguna sustitución con ella.

5. Aplicamos [SAME] todas las veces que podamos, para eliminar relaciones duplicadas, ya que el objetivo es reducir a una sola relación $\alpha \leftharpoondown \beta$ las que tengamos en P.

6. Aplicamos [TURN-L] y [TURN-R], para toda aquella restricción de encaje $\beta \Leftarrow \alpha$ en el que β aparezca un número menor de veces que α en P, junto a su respectivo C o C'.

[1] Con este algoritmo, si el resultado es **false** no quiere decir necesariamente que sea falsa la inclusión entre los tipos, más bien que no sabemos si es cierta.

7. Si tenemos una restricción de encaje $\beta \Leftarrow \alpha$, aplicaremos las reglas [REPLACE-LC] y [REPLACE-LP] si esta restricción estaba en C, o [REPLACE-RC] y [REPLACE-RP] si la restricción estaba en C'. Después de hacer la sustitución saltaremos al punto (4) de la estrategia.

8. Si tenemos en C una restricción de encaje $\tau \Leftarrow \alpha$, donde τ no es una variable de tipo, siempre que α no aparezca más veces en C, podremos aplicar la regla [REPLACE-LP].

9. Si tenemos en C' una restricción de encaje $\tau \Leftarrow \alpha$, donde τ no es una variable de tipo, siempre que α no aparezca más veces en C', podremos aplicar la regla [REPLACE-RP] si en P solo hay una relación de conexión con α en el lado derecho. Esta última condición es importante, porque si hiciéramos la sustitución estaríamos dando como cierta una conexión que no existe.

10. Aplicamos [EXPAND], cuando exista alguna relación donde a ambos lados de la relación de conexión se encuentra un tipo que no sea una variable de tipo. Esto lo querremos hacer cuando después de realizar algunas sustituciones, nos encontremos que, por ejemplo, a ambos lados de la relación hay una tupla, con lo que tendremos que descomponer la relación para entrar en más detalle. Aplicando esta regla, se obtiene directamente un resultado, porque después de descomponer la comparación invocamos recursivamente la función *solve* y lo que nos dé será el resultado.

11. Aplicamos [CHECK], sobre aquellas relaciones únicas $\alpha \leftharpoondown \beta$, para comprobar si están bien conectadas de derecha a izquierda. Para ello tenemos que comprobar que no estamos en una situación donde α es plural y β es singular. Sabemos que una variable de tipo es singular si está en una posición instanciable y no está dentro del cuerpo de una lista no vacía o de un tipo funcional. Con el resultado de la condición podremos aplicar las reglas [TRUE] o [FALSE].

12. Aplicamos [MANY] si se dan las circunstancias, que tendría que ser que τ' contenga variables de tipo, que τ_1 y τ_2 no sean el mismo tipo, además estos tipos no deben contener variables de tipo que existan en el conjunto de restricciones C. En caso de poder aplicarse esta regla, el algoritmo termina devolviendo **false**.

13. Aplicamos [END] si P contiene únicamente el valor **true**, ya que hemos realizado todas las comprobaciones anteriores con éxito.