Obteniendo eficiencia y legibilidad en programas generados automáticamente

Esteban de la Canal

Trabajo de Grado

Director: M.Sc. Pablo E. Martínez López
Codirector: Prof. Lic. Gabriel Baum
Facultad de Informática
Universidad Nacional de La Plata

Índice general

Agradecimientos

Luego de escribir esta tesis puedo darme cuenta de que, más allá de todo el trabajo técnico, esta es la sección más difícil de hacer. Hay mucha gente a la que me gustaría agradecer por mi formación como profesional y voy a intentar hacerlo aquí en una sola carilla de papel.

En primer lugar le quiero agradecer al LIFIA en particular a Gustavo y a Gabriel que, aunque con puntos de vista diferentes, han hecho *muchísimo* por mi educación.

Quiero agradecer también a todos aquellos con quienes compartí cursos o escuelas de verano, tanto asistentes como profesores, gracias a los que me fui volviendo cada vez más loco por lo que hago.

Estoy muy agradecido con el director de mi tesis, el Sr. Martínez López, por haberme guiado con mucha calidad tanto en este trabajo como en toda mi formación académica. Agradezco también a mi compañero de trabajo Pablo E. por haberme hecho descubrir la semilla docente que hay dentro mío y hacerla crecer con unos parámetros docentes envidiables. No debo olvidar de mencionar a mi amigo Fidel, con quien hemos compartido desde charlas filosóficas hasta los más fervientes juegos de tablero. Gracias FF!.

A todos mis compañeros de facultad que me ayudaron a recorrer estos años como estudiante. A aquellos que me hicieron dar cuenta que existe gente como uno y que no se está lo tan loco que se cree estar.

Gracias a Abel, por todas las locuras compartidas durante la carrera, esas redes, ACM, la música, en fin... gracias por haberme dejado encontrar un amigo.

A Hernán. Decir cualquier cosa sobre él seria olvidar cientos de otras. Simplemente gracias H por haber crecido conmigo.

A mi vieja que, sin lugar a dudas, es quien me hizo lo que soy. A mi tío que siempre tuvo y tiene una inexplicable fe ciega en mi.

Finalmente quiero agradecerle a quien hace que todas estas cosas tengan sentido, dándome fuerzas en el día a día y haciéndome sentir que juntos es la única manera de caminar esta vida. A mi Sil.

Capítulo 1

Introducción

No tengo ningún talento especial. Sólo soy apasionadamente curioso.

Albert Einstein

¿Alguna vez ha visto código generado automáticamente? Los generadores automáticos de código ya tienen suficientes problemas para lograr corrección como para estar fijándose en la calidad del código generado. La mayoría de estos generadores dejan para fases posteriores la no grata tarea de dar los últimos retoques al trabajo. Estos "retoques" dependen de la tarea específica del generador de código y de cómo es resuelta.

En esta tesis se ataca uno de los problemas demorados para etapas posteriores en la generación de código mediante la técnica de *Especialización de tipos* de [Hughes, 1996]. Los principios de este mecanismo evolucionaron a partir de un método más general llamado *Especialización de programas*. En las secciones siguientes se explican brevemente cada uno de ellos y finalmente se detalla la etapa a la cual se aboca esta tesis, denominada *Arity Raising*.

1.1 Especialización de programas

La especialización de programas [Jones et al., 1993] es una técnica de generación automática de código que consiste en la producción de código "especializado", a partir de un programa escrito de manera general y un conjunto de restricciones sobre éste. El programa generado debe entonces comportarse de la misma manera que el original cuando se lo ejecuta bajo esas restricciones

Uno de los enfoques más exitosos para esta tarea es la que se realiza mediante la estrategia de evaluación parcial [Jones et al., 1993]. En este caso las restricciones se basan en fijar los valores de algunas variables de entrada y realizar las computaciones que dependan de éstas, obteniendo una versión "especializada" para los valores de entrada dados.

El clásico ejemplo de *Especialización de programas* mediante el uso de evaluación parcial es el siguiente: supongamos que tenemos una función que compute el valor x^n

```
power n x =  if n == 1
             then x
             else x * power (n-1) x
```

Bajo la restricción de que el parámetro `n` siempre será `3`, es posible computar todas las acciones relacionadas con este `n` y así, automáticamente obtener un código como

```
power3 x = x * (x * x)
```

Se ha obtenido un programa que, bajo la restricción supuesta, realiza el mismo trabajo que el original, pero de una manera mucho más eficiente que el anterior pues no hay llamados recursivos ni comparaciones.

1.2 Especialización de tipos

La *Especialización de tipos* [Hughes, 1996] es un enfoque a la *Especialización de programas* en la cual no sólo es especializado el cuerpo de una expresión, sino también su tipo. Si observamos un programa tipado, por ejemplo de tipo `Int`, entonces una versión especializada de éste puede retornar valores de sólo una versión "especializada"de su tipo. Por ejemplo si tenemos el siguiente código

```
id :: Int ->Int
id x = x
```

Si se aplica la técnica de evaluación parcial de programas tradicional, con la variable `x = 42`, se tiene una versión especializada como

```
id42 :: Int
id42 = 42
```

Pero, si se ha modificado el código para que trabaje bajo nuevas restricciones entonces ¿por qué no aplicar el mismo razonamiento a los tipos? En ese caso se podría analizar el tipo del programa residual (i.e. el programa especializado) bajo las mismas restricciones que se utilizaron para el código. De esta manera el tipo del programa anterior puede ser restringido a un tipo que sólo contenga un valor, cuyo significado será el del número 42.

En general, cada vez que aparezca un valor constante es posible especializar su tipo a uno más específico que sólo pueda contenter ese valor. Al realizar esta operación la constante deja de tener sentido pues tendrá un único valor posible. Es así entonces que se puede remplazar por una constante nula que no posea información, ya que su significado será dado por la pertenencia al tipo de datos introducido. En el ejemplo anterior, si se desea especializar la constante `42` restringiendo su tipo, puede especializarse como:

```
id42 :: 4̂2
id42 = ()
```

Donde `()`, la tupla 0-aria, representa la constante nula y $\hat{42}$ denota el tipo de datos que sólo puede poseer un valor cuyo significado será el número 42. En otros

trabajos la constante `()` se denota con el símbolo especial $\bullet$, pero a los efectos de este trabajo es más ortogonal tratarlo como una tupla.

La eliminación de algunas construcciones lleva a otro problema que no está en el campo de la evaluación parcial, que es decidir cuáles son los valores que se van a especializar. Está claro que el especializador, *a priori*, no puede decidir cuáles son los deseos de quien quiere especializar el código, por lo que es éste quien debe señalar qué debe ser especializado y qué no. Para esto se introduce el concepto de construcciones estáticas y dinámicas. Quien utilice el especializador debe *marcar* como *estáticas* las construcciones que desea sean eliminadas, mientras que las construcciones que deben permanecer en el código generado se deben marcar como *dinámicas*. De aquí en adelante se verán marcadas con el superíndice ^{S} las construcciones estáticas y con ^{D}, o simplemente sin anotación, a las dinámicas.

Un problema con el que se enfrenta la especialización de tipos es que al modificar el tipo de las funciones, si una de estas funciones es utilizada a lo largo del programa, las aplicaciones se ven obligadamente modificadas. Veamos el siguiente ejemplo:

$$\begin{array}{l} let\ f\ x\ =\ x \\ in\ (f\ 42^S, f\ 8^S) \end{array}$$

¡Aquí la función f, de tipo $Int^S \to Int^S$, debería ser especializada con *dos* tipos diferentes! Esto es uno para el número 42 y otro para el 8. Para resolver esta situación se utiliza un recurso conocido con el nombre de *polivarianza* y en la especialización se refleja esto con nuevas construcciones que hacen que se generen muchas funciones para f de acuerdo a lo que se necesite. Las distintas versiones de f son introducidas todas dentro de una tupla y en aquellos lugares que haya que utilizar alguna de ellas se usará una proyección sobre esta tupla. Este enfoque produce una gran cantidad de tuplas, generadas por el especializador, que no escribió el programador y que se deben eliminar de alguna manera.

1.3 Arity Raising

La introducción de tuplas para resolver tanto la especialización de una constante como los casos de *polivarianza*, motivan la necesidad de una etapa de posprocesamiento que elimine del código estas construcciones introducidas, no por el programador, sino por el especializador. Permitir la inserción de estructuras auxiliares a la especialización hace que el código residual no sea sólo una versión especializada del original. Las tuplas introducidas, incluyendo la tupla `()`, dan como resultado que el código residual sea abstruso e ineficiente. Si bien es cierto que un código generado no tiene por qué tener la particularidad de ser legible, pues no necesariamente tiene que ser mantenido por un programador, la legibilidad es importante para obtener al menos una versión de código entendible por un humano.

Por otra parte, desde el punto de vista de la eficiencia, en lenguajes como Haskell [Jones et al., 1999] o ML [Milner et al., 1997], la construcción y destrucción de las nuevas tuplas al ser pasadas como parámetros introducen un *overhead* de tiempo innecesario y que no estaba en el programa original. Técnicas ya conocidas, tales como la llamada *Variable Spliting*, se han aplicado para resolver este problema.

Arity Raising es el algoritmo que realiza el trabajo de eliminar del código generado las tuplas que son introducidas por el especializador. El diseño del algoritmo va un paso más alla, independizándose de quién generó el código a tratar, y se dedica a eliminar *todas* las tuplas que se señalen como *estáticas*. Para ello utiliza diferentes estrategias de acuerdo a dónde se encuentren estas tuplas; por ejemplo utiliza *Variable Spliting* si una tupla estática aparece pasada como parámetro.

Para ilustrar el funcionamiento de *Arity Raising* veamos informalmente algunos ejemplos de su utilización.

Ejemplo 1.1. Supongamos que se quiere eliminar la tupla del siguiente código:

```
let t = (1,2,3)
in  first t + second t + third t
```

Luego del proceso de *Arity Raising* se espera que la expresión anterior se transforme en:

```
let a = 1; b = 2; c = 3
in  a + b + c
```

Ejemplo 1.2. Otro ejemplo en el que se utiliza una tupla como argumento de función.

```
let f x = fst x
in  f (2,4)
```

Sería deseable que el resultado de remover esa tupla sea el código:

```
let f x y = x
in  f 2 4
```

Ejemplo 1.3. Este último ejemplo ilustra como se eliminan las constantes `()`. Veamos el siguiente código:

```
let f x = 4; g = ()
in  f () + f g
```

En este caso la aplicación del algoritmo debería devolver el siguiente código:

```
let f = 4
in  f + f
```

1.4 Contribución de esta tesis

Hasta el momento las técnicas utilizadas para realizar *Arity Raising* sólo trabajan con programas monomórficos, o si lo hacen con programas polimórficos, lo hacen de una forma monovariante [Jones et al., 1993]. Por otra parte trabajos recientes han incluido algunas de las características de alto orden [Hannan and Hicks, 1998] y el concepto de *co-Arity Raising* [Thiemann, 2000].

El aporte de esta tesis se enfoca en los dos puntos antes mencionados. En primer término se presenta un algoritmo de *Arity Raising* que presenta la caracteristica de manipular, tanto en el código origen como en el destino, código polimórfico. En segundo lugar, el algoritmo presentado hace uso completo del concepto de alto orden. Además, en esta presentación no hay ningún tipo de restricción para las anotaciones (*estáticas* o *dinámicas*) de entrada, pudiendo estar éstas independientemente en cualquier tupla del código; vale aclarar esto pues en otros trabajos las anotaciones deben seguir reglas particulares como por ejemplo no poder anotar una tupla como dinámica si se encuentra dentro de una tupla estática.

Finalmente se provee una demostración de que el sistema propuesto es correcto. La corrección demostrada está basada en una noción de simulación para lo cual fue necesario dar semántica a los lenguajes utilizados.

1.5 Síntesis

En el capítulo 2 se presenta el lenguaje sobre el que se realizará *Arity Raising* junto con una transformación de éste a un lenguaje de trabajo adecuado para el algoritmo propuesto. En el capítulo 3 se explica en detalle el algoritmo de *Arity Raising* propuesto. En el capítulo 4 se define la semántica de los lenguajes objeto y destino, se plantea la noción de corrección y se demuestra que el algoritmo del capítulo anterior cumple con esta noción. Luego, en el capítulo 5 se presenta la implementación del algoritmo realizado en el lenguaje funcional Haskell [Jones et al., 1999] y se comentan las particularidades observadas en la implementación. Como cierre del trabajo se delinean, en el capítulo 6, las distintas posibilidades para la continuación del presente trabajo y se finaliza, en el capítulo 7, con las conclusiones y un resumen del aporte de esta tesis. Se adjunta además, un apéndice con las demostraciones completas de todo lo postulado durante el desarrollo del trabajo.

Capítulo 2

Lenguaje fuente

Ante todo existió el Caos

Hesíodo

Antes de comenzar con la tarea de remoción de tuplas es necesario preparar el terreno. En este capítulo se define el lenguaje sobre el que trabajará el algoritmo de *Arity Raising*, que denominaremos *lenguaje objeto*. Este lenguaje es el lenguaje fuente enriquecido con construcciones necesarias para aplicar el algoritmo descripto en el capítulo siguiente. Primero se define el lenguaje fuente, luego se plantea el lenguaje objeto y finalmente se presenta la transformación necesaria para obtenerlo.

2.1 Lenguaje fuente

Si bien el lenguaje residual de un especializador, que es el lenguaje fuente para la etapa de *Arity Raising*, posee una gran variedad de construcciones, para el estudio de este trabajo la mayoría de éstas son irrelevantes. Por ello hemos elegido el mínimo conjunto posible que permita construir sobre él todas las herramientas de un lenguaje convencional. El lenguaje fuente que se utilizará es una versión extendida del lambda cálculo para el manejo apropiado de tuplas y de polimorfismo. El lenguaje posee las tres construcciones básicas del lambda cálculo (variables, aplicaciones y abstracciones), las tuplas con sus respectivas proyecciones y una construcción `let` para la introducción de polimorfismo.

Además de definir las tuplas y las proyecciones es necesario incorporar un agregado. Esto es debido a que la intención no es remover todas la tuplas sino aquellas construcciones que haya creado el especializador y no las que el programador definió en el código original. Esto genera la necesidad de distinguir entre ambos tipos de tuplas. Para eso se introduce un concepto derivado de los especializadores: existen las tuplas estáticas, que son las que hay que remover, y las tuplas dinámicas, que deben permanecer en el código.

Ahora se tienen dos tipos de tuplas. Pero, ¿no bastaría con poseer un solo tipo de proyecciones? El siguiente ejemplo ilustra la necesidad de tener ambas proyecciones.

Ejemplo 2.1. Sea $\pi_{n,i}$ la proyección de la componente i-ésima de una tupla de

tamaño n. Observemos el siguiente fragmento de código:

$$\lambda v.\pi_{n,i}\ v$$

En este caso, si bien es posible inferir que v es una tupla, es imposible saber si ésta es dinámica o estática.

A continuación se define formalmente el lenguaje fuente con las características mencionadas anteriormente.

Definición 2.2. Sea v a una *variable del lenguaje fuente*. Un *término* del lenguaje fuente, denotado e, queda definido por la siguiente gramática:

$$e := \ v \mid e\ e \mid \lambda v.e \mid \texttt{let}\ v\ \texttt{=}\ e\ \texttt{in}\ e \mid (e)_n^S \mid (e)_n^D \mid \pi_{i,n}^S\ e \mid \pi_{i,n}^D\ e$$

donde $(e)_n^S$ y $(e)_n^D$ denotan las tuplas de n componentes estáticas y dinámicas respectivamente. Se utilizará la notación $e.i$ para denotar la componente i-ésima de la tupla $(e)_n$

Notar que las proyecciones de las tuplas, estáticas o dinámicas, no están definidas como expresiones por sí mismas, sino que en su definición ya están aplicadas a una expresión particular. Esta decisión facilita ampliamente el desarrollo del trabajo y no introduce ninguna restricción. Pueden definirse posteriormente las proyecciones usuales de la manera siguiente:

$$\texttt{fst}^D = \lambda x.\pi_{1,2}^D\ x$$

Para continuar es necesario definir el tipo que tendrán los términos anteriores. El sistema de tipos presentado es un sistema *Hindley-Milner* tradicional [Damas and Milner, 1982] modificado con dos tipos de tuplas.

Definición 2.3. Sea τ un *tipo base* y σ un *esquema de tipo* definidos de la siguiente manera:

$$\begin{array}{lll} \sigma & := & \forall t.\sigma \mid \tau \\ \tau, \eta & := & \iota \mid t \mid \tau \to \eta \mid (\tau)_n^S \mid (\tau)_n^D \end{array}$$

donde ι denota el conjunto de los tipos base del lenguaje fuente.

2.2 Polimorfismo

El siguiente ejemplo ilustra el funcionamiento de *Arity Raising* sobre una función *monomórfica* que toma una tupla estática como parámetro y devuelve un valor constante.

Ejemplo 2.4. Supongamos que c es un término de tipo ι, el siguiente código

$$\lambda v.c : (\iota_1, \iota_2)^S \to \iota$$

daría como resultado

$$\lambda v_1.\lambda v_2.c : \iota_1 \to \iota_2 \to \iota$$

Por otra parte, ¿qué sucedería si tuviéramos el mismo código pero con tipo diferente? Veamos el siguiente ejemplo

Ejemplo 2.5. Supongamos que c es un término de tipo ι, el siguiente código

$$\lambda v.c : \iota' \to \iota$$

daría como resultado

$$\lambda v.c : \iota' \to \iota$$

A partir de estos ejemplos el lector puede inferir que la expresión sola no basta para la correcta aplicación de *Arity Raising*, sino que el resultado de ésta depende del tipo de todas sus sub-expresiones y no sólo del tipo de la expresión. Al aplicar un algoritmo de *Arity Raising* la entrada no es solamente la expresión y su tipo, sino que se necesita todo el árbol de inferencia de ese tipo.

El polimorfismo utilizado en este lenguaje se denomina polimorfismo *let-bounded*, pues una función es polimórfica sólo en el ambiente en el cual está definido bajo un `let`. Este tipo de polimorfismo es suficiente para las ambiciones de este trabajo, ya que lenguajes como Haskell [Jones et al., 1999] utilizan este mecanismo.

Como vimos anteriormente, el tipo de una expresión tiene un impacto directo en el resultado del procesamiento; pero ¿qué sucede con las expresiones polimórficas? Veamos el siguiente código, siguiendo con la línea de los ejemplos anteriores

$$\lambda v.c : \forall t.t \to \iota$$

¿Que debería dar como resultado? Claramente el código resultante depende del tipo que tome la variable t.

2.3 Lenguaje objeto

El *lenguaje objeto* nace con la intención de simplificar los problemas descritos en la sección anterior. Por un lado captura el árbol de inferencia de tipos utilizando marcas dentro de la expresión y por otro hace explícito el polimorfismo a la manera de [Reynolds, 1998].

Capturar el árbol de inferencia de tipos utilizando marcas puede hacerse utilizando una técnica conocida como *Tipado explícito* [Reynolds, 1998]. Para poder mantener el tipo de cada sub-expresión basta con dejar en la expresión aquellos tipos que se pierdan al momento de aplicar una regla de inferencia. El ejemplo clásico de esto sucede en una aplicación; si tenemos dos expresiones como

$$f : \tau \to \eta$$
$$e : \tau$$

¿Cuál es el tipo de $f\ e$? Obviamente es η; pero en este caso hemos perdido el tipo de e. Por ello las aplicaciones en el nuevo lenguaje poseen una marca del tipo de e. Situaciones similares ocurren con las proyecciones, pues en el tipo de la expresión no aparecen los tipos de las componentes no proyectadas, y con las expresiones definidas en un `let`.

Por otro lado, la utilización de polimorfismo explícito se hace en las definiciones de un `let`, mediante un binder (Λ) que abstrae las variables de tipo libres en una expresión. Estas variables de tipos aparecerán libres en las marcas que posea esta expresión. Luego, en la utilización de una de estas variables es necesario, mediante la aplicación del Λ, instanciar el tipo de la expresión polimórfica adecuadamente.

Para definir el lenguaje objeto, veamos antes un poco de notación.

Notación 2.6. Denotaremos con $\vec{x}$ a la secuencia $x_1, ..., x_n$. Haremos abuso de notación, por ejemplo en los esquemas de tipos, notando $\forall \vec{t}.\tau$, al esquema $\forall t_1.\forall t_2....\forall t_n.\tau$.

Definimos entonces el lenguaje objeto sobre el que se realizará la remoción de tuplas.

Definición 2.7. Sean v las *variables del lenguaje objeto*. Un término del lenguaje objeto, denotado e, está definido por la siguiente gramática:

$$e := \ v\ \vec{\tau} \mid e\ e_\tau \mid \lambda v.e \mid \texttt{let}\ v_\sigma\ \texttt{=}\ \Lambda\vec{t}.e\ \texttt{in}\ e \mid (e)^S_n \mid (e)^D_n \mid \pi^S_{i,n}\ e_\tau \mid \pi^D_{i,n}\ e_\tau$$

Los tipos de estos términos tienen la misma estructura que los tipos del lenguaje fuente, pero de todas maneras son definidos a continuación por completitud.

Definición 2.8. Sea τ un *tipo base* del lenguaje objeto y σ un *esquema de tipo* del lenguaje objeto, definidos por la siguiente gramática.

$$\begin{array}{lll} \sigma & := & \forall t.\sigma \mid \tau \\ \tau, \eta & := & \iota \mid t \mid \tau \rightarrow \eta \mid (\tau)^S_n \mid (\tau)^D_n \end{array}$$

En la definición del lenguaje objeto, los tipos denotados en subíndices son las marcas de tipado explícito del término. Cada uno de estos subíndices indican que el término asociado posee ese tipo. Estas marcas junto con el tipo global de un término determinan completamente el tipo de todas las expresiones involucradas en él.

La expresión $v\ \vec{\tau}$ simboliza una aplicación de tipos, la cual es utilizada en [Reynolds, 1998] para instanciar las marcas del polimorfismo explícito. Aquí, como permitimos sólo una visión sumamente restringida de polimorfismo, la aplicación de tipos sólo puede hacerse en variables y serán resueltas al momento que la variable tome el valor que denota. Para simplificar toda la maquinaria semántica (Capítulo 4), definiremos esta aplicación embebida en la definición de sustitución de una variable por un término.

A continuación se definirán formalmente las nociones explicadas intuitivamente en los párrafos anteriores.

Definición 2.9. Definimos $\tau[\vec{t}/\vec{\tau'}]$ como la sustitución simultánea de t_i por τ'_i en τ.

La diferencia entre una definición de sustitución simultánea respecto de sustituir una tras otra ciertas variables puede verse con el siguiente ejemplo

Ejemplo 2.10. Sea $\tau = t_1$, y la sustitución $S = [t_1, t_2/t_2, \tau']$. Aplicar esta sustitución en forma simultánea produce

$$t_1[t_1, t_2/t_2, \tau'] = t_2$$

Pero si se sustituye de una en una las variables, el resultado es

$$t_1[t_1/t_2][t_2/\tau'] = t_2[t_2/\tau'] = \tau'$$

En general, en el resto del trabajo no se presentará la situación dada anteriormente. La siguiente proposición formaliza esa situación

Propiedad 2.11. *Si $t_2 \notin FV(\tau_1)$, entonces vale que*

$$\tau[t_1/\tau_1][t_2/\tau_2] = \tau[t_1, t_2/\tau_1, \tau_2]$$

Al existir tipado explícito, los términos poseen tipos embebidos en ellos, por lo que es posible extender la noción de sustitución de variables libres de tipos, a términos.

Notación 2.12. Sea $S = [t/\tau]$, notaremos con Se a la expresión $e[t/\tau]$.

Definición 2.13. Sea $S = [t/\eta]$ tal que $t \notin \vec{s}$. Se define la sustitución de tipos sobre términos de la siguiente manera:

$S(v\ \vec{\tau})$	$=$	$v\ (S\vec{\tau})$	$S(e)^S_n$	$=$	$(Se)^S_n$
$S(e\ e_\tau)$	$=$	$Se\ Se_{S\tau}$	$S(e)^D_n$	$=$	$(Se)^D_n$
$S(\lambda v.e)$	$=$	$(\lambda v.Se)$	$S(\pi^S_{i,n}\ e_\tau)$	$=$	$\pi^S_{i,n}\ Se_{S\tau}$
$S(\pi^D_{i,n}\ e_\tau)$	$=$	$\pi^D_{i,n}\ Se_{S\tau}$			
$S(\texttt{let}\ v_\sigma\ \texttt{=}\ \Lambda\vec{s}.e_1\ \texttt{in}\ e_2)$	$=$	$\texttt{let}\ v_{S\sigma}\ \texttt{=}$	$\Lambda\vec{s}.Se_1\ \texttt{in}\ Se_2$		

Esta definición puede extenderse de manera natural, utilizando la definición 2.9, para obtener una sustitución simultánea de tipos sobre términos.

Ahora es posible dar una definición de sustitución de variables por términos, en el lenguaje objeto, embebiendo en ésta la aplicación de tipos.

Definición 2.14. Sea $e' = \Lambda\vec{t}.e''$; se define la sustitución de una variable v por e' en un término e, denotado $e[v/e']$, como

$$\begin{array}{lcll}
(v\ \vec{\tau})[v/e'] & = & e''[\vec{t}/\vec{\tau}] & \\
(v'\ \vec{\tau})[v/e'] & = & v'\ \vec{\tau} & , \text{si } v' \neq v \\
(e\ e_\tau)[v/e'] & = & e[v/e']\ e[v/e']_\tau & \\
(\lambda v.e)[v/e'] & = & \lambda v.e & \\
(\lambda v'.e)[v/e'] & = & \lambda v'.(e[v/e']) & , \text{si } v' \neq v \\
(e)^S_n[v/e'] & = & (e[v/e'])^S_n & \\
(e)^D_n[v/e'] & = & (e[v/e'])^D_n & \\
(\pi^S_{i,n}\ e_\tau)[v/e'] & = & \pi^S_{i,n}\ (e[v/e']_\tau) & \\
(\pi^D_{i,n}\ e_\tau)[v/e'] & = & \pi^D_{i,n}\ (e[v/e']_\tau) & \\
(\texttt{let}\ v_\sigma\ \texttt{=}\ \Lambda\vec{t'}.e\ \texttt{in}\ e)[v/e'] & = & \texttt{let}\ v_\sigma\ \texttt{=}\ \Lambda\vec{t'}.e[v/e']\ \texttt{in}\ e & \\
(\texttt{let}\ v'_\sigma\ \texttt{=}\ \Lambda\vec{t'}.e\ \texttt{in}\ e)[v/e'] & = & \texttt{let}\ v'_\sigma\ \texttt{=}\ \Lambda\vec{t'}.e[v/e']\ \texttt{in}\ e[v/e'] & , \text{si } v' \neq v
\end{array}$$

2.4 Obtención del lenguaje objeto

Para la obtención del lenguaje objeto se utiliza un algoritmo estándar de inferencia de tipos [Damas and Milner, 1982] modificado para generar una nueva expresión perteneciente al lenguaje objeto. Para ello es necesario contar antes con una modificación de la relación estándar para instanciar tipos.

Definición 2.15. Definimos la relación $\leq_{\vec{\eta}}$ de la siguiente manera. Sea $\sigma = \forall\vec{t}.\tau'$,

$$\tau \leq_{\vec{\eta}} \sigma \text{ sii } \tau = \tau'[\vec{t}/\vec{\eta}]$$

Notar que la definición de esta relación, a la que llamaremos *instanciación*, no permite instanciar sólo algunas de las variables abstraídas por un esquema de tipo sino que es necesario hacerlo con todas las variables al mismo tiempo.

Definición 2.16. Definimos $A \vdash e \rightsquigarrow e' : \tau$, como la relación especificada con las reglas de la figura 2.1, que debe leerse como 'A partir de A, la expresión e del lenguaje fuente se convierte en la expresión e' del lenguaje objeto cuyo tipo es τ'.

Veamos ahora, sin entrar en demasiados detalles técnicos, cómo se obtienen algunas expresiones del lenguaje objeto realizando la transformación de la definición anterior.

Ejemplo 2.17. Supongamos tener una función polimórfica definida en un `let` utilizada de la siguiente manera:

$$\texttt{let}\ f\ \texttt{=}\ (\lambda p.\pi^S_{1,2}\ p)\ \texttt{in}\ f\ (1,2)^S_2$$

$$\text{(MG-VAR)}\quad \frac{(v:\sigma)\in A \quad \tau \leq_{\vec{\eta}} \sigma}{A \vdash v \rightsquigarrow v\ \vec{\eta} : \tau}$$

$$\text{(MG-APP)}\quad \frac{A \vdash e \rightsquigarrow e' : \tau \to \eta \quad A \vdash f \rightsquigarrow f' : \tau}{A \vdash e\ f \rightsquigarrow e'\ f'_\tau : \eta}$$

$$\text{(MG-LAM)}\quad \frac{A_v, v:\tau \vdash e \rightsquigarrow e' : \eta}{A \vdash \lambda v.e \rightsquigarrow \lambda v.e' : \tau \to \eta}$$

$$\text{(MG-LET)}\quad \frac{A \vdash e \rightsquigarrow e' : \tau \quad A_v, v:\sigma \vdash f \rightsquigarrow f' : \eta \quad \sigma = \forall \vec{t}.\tau = Gen(A,\tau)}{A \vdash \texttt{let}\ v\ \texttt{=}\ e\ \texttt{in}\ f \rightsquigarrow \texttt{let}\ v_\sigma\ \texttt{=}\ \Lambda \vec{t}.e'\ \texttt{in}\ f' : \eta}$$

$$\text{(MG-STUPLE)}\quad \frac{A \vdash e.i \rightsquigarrow e'.i : \tau.i}{A \vdash (e)^S_n \rightsquigarrow (e')^S_n : (\tau)^S_n}$$

$$\text{(MG-DTUPLE)}\quad \frac{A \vdash e.i \rightsquigarrow e'.i : \tau.i}{A \vdash (e)^D_n \rightsquigarrow (e')^D_n : (\tau)^D_n}$$

$$\text{(MG-SPROJ)}\quad \frac{A \vdash e \rightsquigarrow e' : (\tau)^S_n}{A \vdash \pi^S_{i,n} e \rightsquigarrow \pi^S_{i,n} e'_{(\tau)^S_n} : \tau.i}$$

$$\text{(MG-DPROJ)}\quad \frac{A \vdash e \rightsquigarrow e' : (\tau)^D_n}{A \vdash \pi^D_{i,n} e \rightsquigarrow \pi^D_{i,n} e'_{(\tau)^D_n} : \tau.i}$$

Figura 2.1: Reglas para la construcción del lenguaje objeto

Los números 1 y 2 se utilizan aquí como constantes del tipo ι, que será llamado *Int* sólo para hacer más intuitivo el ejemplo. Suponemos además que estas constantes no se ven afectadas por la conversión. Es decir que

$$\vdash 1 \rightsquigarrow 1 : Int \qquad \vdash 2 \rightsquigarrow 2 : Int$$

Luego, aplicando la regla (MG-STUPLE), vale que

$$\vdash (1,2)^S_2 \rightsquigarrow (1,2)^S_2 : (Int, Int)^S_2$$

Utilizando la regla (MG-VAR) y sabiendo que $(Int, Int)^S_2 \to Int \leq (a,b)^S_2 \to a$ tenemos

$$(f : \forall a.\forall b.(a,b)^S_2 \to a) \vdash f \rightsquigarrow f\ [Int, Int] : (Int, Int)^S_2 \to Int$$

Utilizando los dos últimos resultados y la regla (MG-APP) tenemos que

$$(f : \forall a.\forall b.(a,b)^S_2 \to a) \vdash f\ (1,2)^S_2 \rightsquigarrow f\ [Int, Int]\ (1,2)^S_{2\,(Int,Int)^S_2} : Int \tag{2.1}$$

Por otro lado con la regla (MG-VAR) obtenemos

$$(p : (a,b)^S_2) \vdash p \rightsquigarrow p : (a,b)^S_2$$

Aplicando la regla (MG-SPROJ) tenemos

$$(p : (a, b)_2^S) \vdash \pi_{1,2}^S\ p \rightsquigarrow \pi_{1,2}^S\ p_{(a,b)_2^S} : a$$

Y utilizando la regla (MG-LAM)

$$\vdash \lambda p.\pi_{1,2}^S\ p \rightsquigarrow \lambda p.\pi_{1,2}^S\ p_{(a,b)_2^S} : (a, b)_2^S \to a$$

Utilizando este último resultado junto con (2.1), aplicando la regla (MG-LET), obtenemos finalmente

$$\begin{array}{ll} \vdash & \texttt{let}\ f = (\lambda p.\pi_{1,2}^S\ p)\ \texttt{in}\ f\ (1, 2)_2^S \rightsquigarrow \\ & \texttt{let}\ f_{\forall a.\forall b.(a,b)_2^S \to a} = \Lambda a.\Lambda b.(\lambda p.\pi_{1,2}^S\ p_{(a,b)_2^S})\ \texttt{in}\ f\ [Int, Int]\ (1, 2)_{2\,(Int,Int)_2^S}^S : Int \end{array}$$

Capítulo 3

Arity Raising

¿Te das cuenta que no eres un simple artesano?.
Te has convertido en un verdadero artista.

Sir Gerald Martin
El robot humano
Isaac Asimov

En este capítulo se describe en detalle cuál es la tarea que se tiene que realizar y se plantea un algoritmo que permite realizar la remoción de tuplas de un lenguaje polimórfico. Se presenta primero el lenguaje destino, en el que se expresarán los resultados de la remoción de las tuplas y luego se especifica el algoritmo.

3.1 Descripción

El objetivo principal de la etapa de *Arity Raising* es la eliminación de todas las tuplas estáticas en el código de entrada. Esta remoción se puede dividir en dos problemas claramente diferenciables. El primero es determinar la forma en que se debe remover una tupla estática que está siendo pasada como parámetro a una función; el segundo es eliminar aquellas tuplas que están siendo devueltas como resultado en una función. Analicemos estos casos por separado.

3.1.1 Tuplas en parámetros de funciones

La remoción de las tuplas que aparecen como parámetro es relativamente más sencilla que cuando éstas se encuentran como resultado. Supongamos el caso en que una función reciba un par estático como parámetro: si bien es necesario eliminar la tupla, las componentes de ésta deberán seguir apareciendo en el código, puesto que tenemos por objetivo obtener código que preserve el mismo significado que el anterior. El primer enfoque para eliminar este par pero mantener sus componentes es cambiar el parámetro de la función por dos nuevos parámetros que recibirán las componentes por separado. Esta decisión involucra dos tareas a resolver. La primera es modificar cada llamado a la función para que en lugar de recibir pares, ésta comience a recibir las componentes separadas y la segunda es modificar el cuerpo de la función para que no trabaje más con un par como parámetro sino que manipule las componentes del par independientemente. Visto desde otro punto de vista se puede ver este procesamiento como una *currificación*

[Jones et al., 1999] automática de todas las funciones que reciban tuplas estáticas. Para ilustrar el problema a resolver veamos el siguiente ejemplo.

Ejemplo 3.1. Supongamos tener las siguientes expresiones en algún lugar de nuestro código en el lenguaje objeto. Por un lado una definición de función

$$f = \lambda x.\pi^S_{1,2}\ x$$

y por otro un llamado a ésta

$$f\ (e_1, e_2)^S_{(t_1,t_2)^S}$$

¿Cuál sería un resultado intuitivo al eliminar el par estático? En primer lugar es necesario modificar el cuerpo de la función para que, en lugar del par, reciba las componentes por separado. Esto genera la necesidad de modificar todo el cuerpo de la función para reflejar estos cambios. Desearíamos entonces obtener una nueva definición de f tal como

$$f = \lambda x.\lambda y.x$$

En segundo lugar se debe modificar el llamado a la función, eliminando la tupla y dejando sus componentes, obteniendo como resultado

$$f\ e_1\ e_2$$

Esta técnica de eliminación de tuplas es ampliamente conocida y recibe el nombre de *variable splitting* [Romanenko, 1990], y tiene como resultado elevar la aridad de las funciones que posean tuplas estáticas como parámetros; este es el hecho que le da nombre a la etapa completa.

3.1.2 Tuplas como resultado de funciones

Una función que devuelve una tupla estática debería ser modificada de manera que preserve el mismo significado, pero sin la utilización de la tupla. Un enfoque para solucionar este problema es el de poseer un algoritmo *polivariante*. Al devolver una tupla, la función está devolviendo varios valores simultáneamente; es decir que ésta podría separar sus tareas en varias funciones diferentes, donde cada una de ellas devuelva el valor de cada componente de la tupla. Esta característica, la de devolver varios códigos destino en lugar de uno solo, es la que le da al algoritmo la clasificación de *polivariante*. Para realizar esta tarea es necesario modificar la función, introduciendo nuevas funciones y reestructurar los llamados a ésta para que utilicen las nuevas funciones adecuadamente. La eliminación de las tuplas como resultado de funciones es una tarea más difícil que la anterior porque, sin entrar en detalles, es necesario tener la capacidad de manipular varias expresiones obtenidas de transformar una sola, para luego volver introducirlas en el código en los lugares adecuados. Para aclarar esta situación veamos el siguiente ejemplo.

Ejemplo 3.2. Supongamos tener en nuestro código una definición de función tal como

$$f = \lambda x.\lambda y.(x + y, x - y)^S$$

y luego unos llamados a ésta como los siguientes

$$\pi^S_{1,2}\ (f\ 1\ 2) + \pi^S_{2,2}\ (f\ 3\ 4)$$

La definición de la función podría separarse en dos nuevas funciones, donde cada una realiza las tareas por separado. Así, se obtendría

$$f_1 = \lambda x.\lambda y.x + y$$
$$f_2 = \lambda x.\lambda y.x - y$$

Y luego los llamados a la función deberían reemplazarse por

$$f_1\ 1\ 2 + f_2\ 3\ 4$$

Como se puede ver en el ejemplo, no sólo es necesario cambiar los llamados a la función sino que deben cambiarse las expresiones en los que éstos están envueltos. Ahí es donde radica una de las más grandes dificultades.

3.2 Lenguaje Destino

Luego de aplicar la etapa de *Arity Raising* obtenemos expresiones en un lenguaje que posee diferentes características que el lenguaje objeto. Este lenguaje, que de aquí en adelante llamaremos *Lenguaje Destino*, no posee dos tipos de tuplas sino que tiene únicamente el tipo que representa el residuo de las tuplas dinámicas del lenguaje objeto. Definamos formalmente el lenguaje.

Definición 3.3. Sea v una *variable del lenguaje destino*; denotaremos con e a un *término del lenguaje destino* definido por la siguiente gramática.

$$e := \ v \mid e\ e \mid \lambda v.e \mid \texttt{let}\ v_1\ \texttt{=}\ e_1; ...; v_n\ \texttt{=}\ e_n\ \texttt{in}\ e \mid (e)_n \mid \pi_{i,n}\ e \mid \$e \mid \texttt{force}\ e$$

Las tuplas y sus componentes del lenguaje destino se denotan de la misma manera que las del lenguaje objeto, salvo que sin las marcas de estáticas o dinámicas. Este lenguaje posee una diferencia con el lenguaje objeto en la definición del `let`. Aquí, esta construcción permite la declaración de varias variables simultáneamente, pero posee la restricción de que una definición de una variable no puede hacer referencia a otra variable definida en el mismo `let`.

Más allá de estas diferencias de forma, este lenguaje posee una diferencia importante con el anterior en las construcciones \$ y `force`. Si bien en el capítulo 4 se define formalmente la semántica de éstos, haremos una descripción informal de cada uno y a lo largo de este capítulo se motivará su introducción en este lenguaje.

La construcción \$$e$ [Okasaki, 1998] denota una ejecución suspendida de la expresión e, pudiéndose acceder solamente utilizando la operación `force`. En lenguajes que carecen de *laziness* la introducción de operadores como \$ son frecuentes para poder poseer evaluación *lazy*. En el marco de este trabajo es utilizada para preservar *laziness* en aquellos lugares en los que la remoción de tuplas eliminarían esta particularidad.

El lector atento habrá notado que hasta el momento se están utilizando las mismas categorías sintácticas para describir tanto expresiones de los lenguajes fuente u objeto como del lenguaje destino. Se distinguirán unas de otras de acuerdo al contexto en que se estén utilizando y se aclarará en aquellos lugares que esta inferencia no sea posible.

Definición 3.4. Sean τ un *tipo del lenguaje destino*, y σ un *esquema de tipos del lenguaje destino*, definidos por la siguiente gramática:

$$\begin{aligned} \sigma &:= \forall t.\sigma \mid \tau \\ \tau &:= \iota \mid t \mid \tau \to \tau \mid (\tau)_n \mid \texttt{susp}\ \tau \end{aligned}$$

Los tipos del lenguaje destino, además del tipo para las tuplas residuales, poseen un tipo especial para identificar las expresiones en suspensión. En la figura 3.1 pueden observarse las reglas (HMD-SUSP) y (HMD-FORCE) donde se introduce y elimina este tipo. Observar además que se asume el mismo conjunto de tipos base (ι) para el lenguaje fuente y para el lenguaje destino. Esta decisión no afecta de ninguna manera el desarrollo del trabajo.

Definición 3.5. Para expresar la inferencia de tipos usaremos un juicio de la forma $A \vdash e : \sigma$. Las reglas que permiten formular juicios válidos se dan en la figura 3.1.

3.3 *Arity Raising* de Tipos

Antes de comenzar con la especificación del algoritmo en sí, es necesario echar un vistazo a los tipos de los términos. La primer incógnita a resolver es la siguiente: ¿El tipo del código resultante depende del código fuente o solamente de su tipo?. Veamos algunos ejemplos:

Ejemplo 3.6. Supongamos tener la siguiente expresión, junto con su tipo, en el lenguaje objeto

$$(\lambda x.\lambda y.\pi^S_{1,2}\ y) : (\iota, (\iota, \iota)^S)^D \to (\iota, \iota)^S \to \iota$$

La eliminación de la primer tupla estática es absorbida por la tupla dinámica, que incrementa su aridad. La segunda tupla impacta en la función currificándola. La expresión resultante y su tipo es entonces

$$(\lambda x.\lambda y_1.\lambda y_2.y_1) : (\iota, \iota, \iota) \to \iota \to \iota \to \iota$$

$$\text{(HMD-VAR)}\ \frac{(v:\sigma)\in A}{A\vdash v:\sigma}$$

$$\text{(HMD-APP)}\ \frac{A\vdash e:\tau\to\eta \quad A\vdash f:\tau}{A\vdash e\,f:\eta}$$

$$\text{(HMD-LAM)}\ \frac{A_v, v:\tau\vdash e:\eta}{A\vdash \lambda v.e:\tau\to\eta}$$

$$\text{(HMD-TUPLE)}\ \frac{A\vdash e.i:\tau.i}{A\vdash (e)_n:(\tau)_n}$$

$$\text{(HMD-PROJ)}\ \frac{A\vdash e:(\tau)_n}{A\vdash \pi_{i,n}e:\tau.i}$$

$$\text{(HMD-SUSP)}\ \frac{A\vdash e:\tau}{A\vdash \$e:\texttt{susp}\ \tau}$$

$$\text{(HMD-FORCE)}\ \frac{A\vdash e:\texttt{susp}\ \tau}{A\vdash \texttt{force}\ e:\tau}$$

$$\text{(HMD-GEN)}\ \frac{A\vdash e:\sigma}{A\vdash e:\forall v.\sigma}\ (v\notin A)$$

$$\text{(HMD-INST)}\ \frac{A\vdash e:\forall v.\sigma}{A\vdash e:\sigma[v/\tau]}$$

$$\text{(HMD-LET)}\ \frac{A\vdash e_i:\sigma_i \quad A_{v_1,\ldots,v_n}, v_1:\sigma_1,\ldots,v_n:\sigma_n\vdash f:\eta}{A\vdash \texttt{let}\ v_1\ \texttt{=}\ e_1;\ldots;v_n\ \texttt{=}\ e_n\ \texttt{in}\ f\colon \eta}$$

Figura 3.1: Reglas para la inferencia de tipos en el lenguaje destino

En este caso el tipo de la función resultante no depende en ningún aspecto del código que poseía el tipo original. Veamos ahora un ejemplo un poco más complicado:

Ejemplo 3.7. Supongamos tener una expresión como la siguiente

$$\lambda x.\lambda y.(x,y)^S : \iota\to\iota\to(\iota,\iota)^S$$

Aquí la función devuelve una tupla estática. La eliminación de ésta supondría la generación de alguna forma de obtener dos valores como resultado. Para resolver esto se generan dos expresiones resultantes:

$$\lambda x.\lambda y.x : \iota\to\iota\to\iota,\ \lambda x.\lambda y.y : \iota\to\iota\to\iota$$

El caso anterior nos lleva a pensar en la necesidad de poseer un algoritmo de *Arity Raising polivariante* que para un mismo fragmento de código genere varios

programas como resultado. En el capítulo 4 veremos que la semántica dada a cada uno de los programas residuales concuerda con la de las componentes de la tupla original. A partir de aquí debemos considerar entonces que el algoritmo que vamos a definir no genera un único código residual, sino una secuencia de ellos. Sin embargo, al igual que en el caso anterior, los tipos del resultado no dependen del código original sino solamente de su tipo. A continuación definimos formalmente conceptos necesarios para el tratamiento de estas secuencias.

Notación 3.8. Se denotará a la secuencia de términos en el lenguaje destino $e_1, ..., e_n$, como $[e_i]_{i:1..n}$.

Definición 3.9. Se define la concatenación de dos secuencias $[e_i]_{i:1..n}$ y $[f_j]_{j:1..m}$, y se la denota $[e_i]_{i:1..n} +\!\!+ [f_j]_{j:1..m}$, a la secuencia $e_1, ..., e_n, f_1, ..., f_m$.

El siguiente paso a dar es resolver la aparición de variables de tipos. Las expresiones polimórficas se definen sólo mediante la utilización de una construcción `let`. De esta manera, si tenemos un término como

$$\texttt{let } v \texttt{ = } e \texttt{ in } f$$

la variable v, a la que se hace referencia desde f, es la versión polimórfica de la expresión e. Cada aparición de esta variable en f asigna un nuevo tipo a las variables del tipo de e; y como hemos visto esta asignación es determinante para la aplicación del algoritmo. En el resto del capítulo veremos cómo se resuelve este problema en lo referente al código. Observando sólo los tipos de la expresión polimórfica, que es la que posee variables de tipos, podemos observar entonces que el tipo resultante depende de cuántos tipos genere una asignación determinada a una variable, pero que no es importante cuáles son esos tipos. En el capítulo 4 se demuestra esta particularidad.

Los párrafos anteriores pretenden justificar de manera intuitiva que el tipo de la expresión residual depende solamente del tipo de la expresión origen más un contexto que determine la cantidad de tipos que generará cada variable de tipo libre. La cantidad de tipos generados por *Arity Raising* a partir de un tipo dado se denominará *tuplicidad* del tipo. Más adelante definiremos formalmente dicho número.

Definimos entonces la relación que especifica la conversión de un tipo del lenguaje objeto en varios del lenguaje destino.

Definición 3.10. Sea Δ un conjunto que asocia cada variable de tipo con una tuplicidad; definimos la relación de *Arity Raising* sobre tipos, denotada $\Delta \vdash \sigma \xrightarrow{tar} [\sigma_i]_{i:1..n}$, según las reglas de la figura 3.2

Veamos en detalle cada una de las reglas.

$$\text{(TAR-T)} \quad \Delta \vdash \iota \xrightarrow{tar} [\iota]$$

$$\text{(TAR-T)}\quad \Delta \vdash \iota \xrightarrow{tar} [\iota]$$

$$\text{(TAR-V)}\quad \frac{(t,n) \in \Delta}{\Delta \vdash t \xrightarrow{tar} [t_i]_{i:1..n}}$$

$$\text{(TAR-FUN)}\quad \frac{\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n>0} \quad \Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..m}}{\Delta \vdash \tau \to \eta \xrightarrow{tar} [\tau'_1 \to ... \to \tau'_n \to \eta'_i]_{i:1..m}}$$

$$\text{(TAR-SUSP)}\quad \frac{\Delta \vdash \tau \xrightarrow{tar} [\,] \quad \Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..n}}{\Delta \vdash \tau \to \eta \xrightarrow{tar} [\texttt{susp}\ \eta'_i]_{i:1..n}}$$

$$\text{(TAR-DTUPLE)}\quad \frac{\Delta \vdash \tau.i \xrightarrow{tar} [\tau'.i_j]_{j:1..m_i}}{\Delta \vdash (\tau)^D_n \xrightarrow{tar} [(\tau'.1_1, ..., \tau'.1_{m_1}, ..., \tau'.n_{m_n})_{m_1+\cdots+m_n}]}$$

$$\text{(TAR-STUPLE)}\quad \frac{\Delta \vdash \tau.i \xrightarrow{tar} [\tau'.i_j]_{j:1..m_i}}{\Delta \vdash (\tau)^S_n \xrightarrow{tar} [\tau'.1_j]_{j:1..m_1} \Vdash ... \Vdash [\tau'.n_j]_{j:1..m_n}}$$

$$\text{(TAR-SCHM)}\quad \frac{\Delta, (t,n) \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..m}}{\Delta \vdash \forall t.\sigma \xrightarrow{tar} [\forall t_1 \ldots \forall t_n.\sigma'_i]_{i:1..m}}$$

Figura 3.2: Reglas del "Type Arity Raising"

En la regla para los tipos base, al no haber tuplas involucradas, se genera como resultado una copia del mismo elemento.

$$\text{(TAR-V)}\quad \frac{(t,n) \in \Delta}{\Delta \vdash t \xrightarrow{tar} [t_i]_{i:1..n}}$$

El caso de una variable de tipo no puede ser resuelto localmente, ya que se estarían tomando decisiones arbitrarias. Es aquí donde surge la necesidad de incluir un contexto que indique la *tuplicidad* de la variable; este valor indica cuántas variables se deben generar como resultado (n en la regla). Observar que las variables generadas *no* son variables frescas (nuevas) sino que son el resultado de agregarle subíndices a la variable original. Este hecho es fundamental pues los *binders* que capturan estas variables serán tratados de la misma manera logrando así generar nuevos *bindings* para las nuevas variables.

$$\text{(TAR-FUN)}\quad \frac{\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n>0} \quad \Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..m}}{\Delta \vdash \tau \to \eta \xrightarrow{tar} [\tau'_1 \to ... \to \tau'_n \to \eta'_i]_{i:1..m}}$$

Para decidir el resultado de procesar el tipo de una función, primero se observa cuál es el resultado de procesar el argumento y el resultado de la función. Si el argumento genera una secuencia de uno o más tipos como resultado, entonces es posible pensar en utilizar un mecanismo similar a la currificación y con ello

se resuelve cómo tratar al argumento. El problema surge cuando el resultado de la función se transforma en una secuencia de tipos, lo que se puede interpretar como que la función está devolviendo una secuencia de valores. Como resultado se generan entonces los tipos para cada función que devuelva un solo valor de la secuencia.

$$\text{(TAR-SUSP)} \quad \frac{\Delta \vdash \tau \xrightarrow{tar} [\,] \quad \Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..n}}{\Delta \vdash \tau \to \eta \xrightarrow{tar} [\texttt{susp}\ \eta'_i]_{i:1..n}}$$

En el caso anterior asumimos que el tipo del parámetro genera uno o más tipos como resultado, pero es posible también que el parámetro no genere ningún tipo como resultado. En este caso la eliminación completa de la función no sería correcta pues no sólo se están eliminando los argumentos, cosa que es deseable, sino que se está perdiendo la naturaleza de no evaluación del resultado de la función. Para esto se introduce el tipo `susp` η'_i que representa los valores de tipo η'_i cuya evaluación está suspendida. (Ver semántica de la suspensión en el capítulo 4)

$$\text{(TAR-DTUPLE)} \quad \frac{\Delta \vdash \tau.i \xrightarrow{tar} [\tau'.i_j]_{j:1..m_i}}{\Delta \vdash (\tau)_n^D \xrightarrow{tar} [(\tau'.1_1, ..., \tau'.1_{m_1}, ..., \tau'.n_{m_n})_{m_1+\cdots+m_n}]}$$

El resultado de procesar el tipo de una tupla dinámica es una sola tupla donde sus componentes son la concatenación de todos las secuencias que se obtienen al procesar los tipos de la tupla original.

$$\text{(TAR-STUPLE)} \quad \frac{\Delta \vdash \tau.i \xrightarrow{tar} [\tau'.i_j]_{j:1..m_i}}{\Delta \vdash (\tau)_n^S \xrightarrow{tar} [\tau'.1_j]_{j:1..m_1} \mathbin{\#} ... \mathbin{\#} [\tau'.n_j]_{j:1..m_n}}$$

Una tupla estática es similar al caso anterior, salvo por la diferencia de que los tipos resultantes no son englobados en una tupla sino que el resultado es la secuencia completa.

$$\text{(TAR-SCHM)} \quad \frac{\Delta, (t, n) \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..m}}{\Delta \vdash \forall t.\sigma \xrightarrow{tar} [\forall t_1 \ldots \forall t_n.\sigma'_i]_{i:1..m}}$$

La regla para los esquemas de tipo especifica lo siguiente: Si el resultado de procesar un esquema de tipo σ, asumiendo que la *tuplicidad* para la variable de tipo t es n, da como resultado una secuencia de m esquemas de tipo σ'_i entonces el resultado de procesar la generalización de la variable t en σ es la misma secuencia anterior pero donde se generalizan las n variables generadas por t.

Si observamos la relación definida por las reglas anteriores a excepción de la regla (TAR-SCHM) se puede ver que se está definiendo una relación funcional, es decir que dado un tipo τ se puede obtener su secuencia de tipos asociada. La

incorporación de la regla (TAR-SCHM) hace que, dado que la elección del número n puede ser arbitraria, se puedan obtener diferentes secuencias para un mismo esquema tipo σ. Es posible entonces, dejando de lado los esquemas de tipo, definir una función para los tipos τ como la siguiente:

Definición 3.11. Se define la *tuplicidad* de un tipo τ, relativa a un conjunto de tuplicidades Δ, como el valor devuelto por la función $\rho(\Delta, \tau)$, definida como $\rho(\Delta, \tau) = n$ *si* $\Delta \vdash \tau \xrightarrow{tar} [\tau']_{i:1..n}$

La utilización de este algoritmo sobre un ambiente polimórfico hace necesario entender el comportamiento que posee el *Arity Raising* de tipos con respecto a las sustituciones de tipos. Supongamos que tenemos una transformación de un tipo τ en varios tipos τ'_i, en la cual se supone que la tuplicidad de una variable t es n. ¿Qué sucede con el resultado si se sustituye en τ la variable t por algún tipo, llamémoslo η, que posea la misma tuplicidad? El resultado es el esperado: es el mismo que sustituir cada una de las variables provenientes de t, en cada τ'_i, por el resultado de transformar η. El lema siguiente formaliza lo antedicho.

Lema 3.12. *Sea* η *tal que* $\Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..m}$. *Si* $(t, m) \in \Delta$ *y* $\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n}$, *entonces* $\Delta \vdash \tau[t/\eta] \xrightarrow{tar} [\tau'_i[\vec{t}/\vec{\eta}]]_{i:1..n}$.

Una operación útil sobre un conjunto de *tuplicidades* es aquella que agrega la tuplicidad para una variable de acuerdo a la tuplicidad de un tipo dado.

Definición 3.13. Definimos la operación $\Delta; (t, \tau)$ definida como

$$\Delta; (t, \tau) = \Delta, (t, \rho(\Delta, \tau))$$

Utilizar esta operación para agregar tuplicidades a un conjunto conlleva una idea de orden, puesto que la tuplicidad nueva es calculada en base a las anteriores. Si bien es cierto que agregar la tuplicidad para una variable que no está libre en un tipo dado, hace que la tuplicidad de éste no cambie, en el caso general esto no es cierto. La siguiente propiedad formaliza esta situación.

Propiedad 3.14. *Sea* Δ *tal que* $s, t \notin \Delta$, *entonces vale que*

$$\Delta; (s, \tau); (t, \eta) = \Delta; (t, \eta[s/\tau]); (s, \tau)$$

A partir de aquí será útil la aplicación de la función de tuplicidad ρ sobre una secuencia de tipos. Definimos entonces la extensión de la función.

Definición 3.15. Sea $\eta_1, ..., \eta_n$ una secuencia de tipos denotada $\vec{\eta}$. Definimos la extensión sobre secuencias de la función ρ de la siguiente forma

$$\rho(\Delta, \vec{\eta}) = \rho(\Delta, \eta_1), \ldots, \rho(\Delta, \eta_n)$$

A continuación se ven un par de ejemplos sobre la aplicación de las reglas. En el primero se puede observar cuál es el resultado de procesar un tipo monomórfico y en el segundo se utiliza un tipo polimórfico y se muestran varios resultados para éste.

Ejemplo 3.16. Supongamos tener el siguiente tipo monomórfico:

$$(\iota, (\iota, \iota)^S)^S \to \iota$$

En primer lugar, utilizando la regla (TAR-T) obtenemos

$$\vdash \iota \xrightarrow{tar} [\iota]$$

Aplicando la regla (TAR-STUPLE) sobre ese resultado podemos afirmar

$$\vdash (\iota, \iota)^S \xrightarrow{tar} [\iota, \iota]$$

Volviendo a aplicar (TAR-STUPLE) sobre los dos resultados anteriores

$$\vdash (\iota, (\iota, \iota)^S)^S \xrightarrow{tar} [\iota, \iota, \iota]$$

Finalmente utilizando la regla (TAR-FUN) sobre este último resultado obtenemos

$$\vdash (\iota, (\iota, \iota)^S)^S \to \iota \xrightarrow{tar} [\iota \to \iota \to \iota \to \iota]$$

Se puede observar que al eliminar las tuplas estáticas de los argumentos de la función se ha obtenido la versión currificada del tipo anterior.

Ejemplo 3.17. Veamos un ahora un tipo de datos polimórfico. Tomemos por ejemplo

$$\forall a.\forall b.a \to b$$

Aquí, la elección de las tuplicidades de a y b es fundamental, pues es la que decide el resultado final. Seleccionemos entonces el siguiente conjunto de tuplicidades, y lo denotaremos con Δ:

$$\Delta = \{(a, 2), (b, 1)\}$$

Utilizando entonces la regla (TAR-V) tenemos que

$$\Delta \vdash a \xrightarrow{tar} [a_1, a_2]$$
$$\Delta \vdash b \xrightarrow{tar} [b_1]$$

Utilizando ahora la regla (TAR-FUN) para unir estos resultados obtenemos

$$\Delta \vdash a \to b \xrightarrow{tar} [a_1 \to a_2 \to b_1]$$

Finalmente, aplicando dos veces la regla (TAR-SCHM) se puede concluir con

$$\Delta \vdash \forall a.\forall b.a \to b \xrightarrow{tar} [\forall a_1.\forall a_2.\forall b_1.a_1 \to a_2 \to b_1]$$

Sin embargo, con una elección diferente para Δ, se pueden obtener resultados radicalmente diferentes. Tomemos por ejemplo:

$$\{(a, 0), (b, 2)\}$$

Comenzando por la regla (TAR-V) tenemos

$$\begin{array}{l} \Delta \vdash a \xrightarrow{tar} [] \\ \Delta \vdash b \xrightarrow{tar} [b_1, b_2] \end{array}$$

Aplicando aquí la regla (TAR-SUSP) se obtiene

$$\Delta \vdash a \to b \xrightarrow{tar} [\texttt{susp}\ b_1, \texttt{susp}\ b_2]$$

Finalmente utilizando la regla (TAR-SCHM), y eliminando los cuantificadores innecesarios, tenemos

$$\Delta \vdash \forall a.\forall b.a \to b \xrightarrow{tar} [\forall b_1.\texttt{susp}\ b_1, \forall b_2.\texttt{susp}\ b_2]$$

3.4 Arity Raising Polimórfico

Como se ha visto anteriormente la introducción de polimorfismo en el proceso de *Arity Raising* trae aparejados resultados no triviales. Volvamos momentáneamente a un ejemplo del capítulo 2 en el cual se estudiaba cuál era el resultado que debería tener el siguiente código

$$\lambda v.c : \forall t.t \to \iota$$

Como hemos visto, el resultado depende del valor que tome la variable t. En la sección anterior se expuso que la aparición de expresiones polimórficas en el lenguaje objeto está definida por una expresión `let`. Es decir que para hacer referencia a una expresión polimórfica es necesario utilizar una variable que la represente. Luego, el alcance de esa variable, que es el cuerpo del `let`, es el único lugar donde puede aparecer ésta. Entonces, si conocemos todas las apariciones de la variable polimórfica, en las cuales se conoce su tipo ya instanciado, ¿no es posible utilizar dicha información para saber todas las posibles asignaciones de tipos para t? La técnica de generar todas las versiones monomórficas de una definición en un `let`, por vías de inspeccionar el cuerpo de éste para ver cuales son los lugares donde se utiliza, es ampliamente conocida y se denomina *monomorfización*.

Aplicar monomorfización para resolver el problema que se presenta al introducir polimorfismo en *Arity Raising* está lejos de ser una buena solución. Supongamos tener la siguiente definición

$$\texttt{let } f_{\forall t.\iota} \texttt{ = } \lambda x.c \texttt{ in } e$$

Donde c es una constante de tipo ι y en e aparecen innumerables referencias a f, pero todas ellas con tipos base diferentes. Aplicar monomorfización llevaría a tener una definición distinta de f para cada tipo al cual se aplica, independientemente de si el código para cada una de éstas es distinto o no. ¿Cómo es posible entonces identificar cuál es un punto coherente para aplicar monomorfización? ¿Cuáles son aquellos tipos que, al ser vistos como el tipo de la función polimórfica, dan el mismo código como resultado? La respuesta a estas preguntas es *tuplicidad*.

A manera de primer ejemplo veamos qué sucede con los tipos base. Como hemos visto en la sección anterior todos los tipos base tienen la misma tuplicidad, en particular, 1. Para generar código para todas las referencias a f que instancien la variable t con un tipo de tuplicidad 1, bastaría con poseer un único código **polimórfico** para todas ellas.

Veamos otro ejemplo en el cual dos instanciaciones para tipos completamente diferentes, pero cuya tuplicidad es la misma, generarían el mismo código. Supongamos que en e aparecen dos llamados a f:

$$f\ (c, c)^S$$
$$f\ (c, \lambda x.\pi^S_{1,2}\ x)^S$$

En el primer caso la variable t de f se instancia con el tipo $(\iota, \iota)^S$, que, como vimos en la sección anterior, posee como resultado $[\iota, \iota]$. Es decir que su tuplicidad es 2.

En el segundo caso la variable t se instancia con $(\iota, (\iota, \iota)^S \to \iota)^S$, cuyo resultado es $[\iota, \iota \to \iota \to \iota]$, de tuplicidad también 2.

En ambos casos bastaría con generar una versión código polimórfica de f de la siguiente manera:

$$\texttt{let } f_{\forall t_1.\forall t_2.\iota} \texttt{ = } \lambda x_1.\lambda x_2.c \texttt{ in } e'$$

y convertir los llamados a f en

$$f\ c\ c$$
$$f\ c\ (\lambda x_1.\lambda x_2.x_1)$$

Los ejemplos anteriores muestran de una manera informal que no es necesario realizar una monomorfización completa sino que basta con generar diferentes códigos para aquellas referencias a las expresiones polimórficas que instancien el tipo polimorfico con tipos de diferente *tuplicidad*.

El desarrollo de un algoritmo que realice la tarea descripta involucrará entonces un método para recolectar todas las instanciaciones que se realizan sobre una expresión polimórfica. Definamos primero este concepto.

A continuación se explicará el sistema de *Arity Raising* mediante un conjunto de reglas. Estas reglas especifican el algoritmo de acuerdo a la estructura sintáctica del término y a su tipo. Desde el punto de vista algorítmico *Arity Raising* produce, a partir de:

- un código fuente,
- su árbol de derivación,
- un juicio para determinar la *tuplicidad* de las variables de tipos libres en el código,

la salida:

- una secuencia de programas,
- un conjunto de instanciaciones.

Definiremos entonces la relación que caracteriza al algoritmo.

Definición 3.18. Sea Δ un conjunto de *tuplicidades* de variables de tipo, Γ un conjunto de instanciaciones, e un término del lenguaje objeto de tipo τ; definimos la relación $\Delta \mid \Gamma \vdash e : \tau \rightarrowtail [e'_i]_{i:1..n}$ con la que se especifica el algoritmo de *Arity Raising* mediante las reglas que se presentan en la figura 3.3.

Veamos detenidamente cada una de las reglas que especifican el algoritmo.

$$\text{(AR-VAR)} \quad \Delta \mid (v, \vec{\tau}) \vdash v\ \vec{\tau} : \eta \rightarrowtail [v_{\rho(\Delta,\vec{\tau}),i}]_{i:1..\rho(\Delta,\eta)}$$

Cada variable de tipo η genera como resultado tantas variables como tuplicidad tiene este tipo. Observar que las variables generadas *no* son frescas (nuevas) sino que es la misma variable más un subíndice.

Como se explicó antes, ya que es posible que la variable sea una referencia a una expresión polimórfica, aquí es donde se comienza a utilizar el concepto de monomorfización. La misma variable puede generar distintas variables de acuerdo a las tuplicidades de $\vec{\tau}$; si la tuplicidad de la instanciación, es decir de $\vec{\tau}$, es igual a la de otra instanciación, digamos $\vec{\tau'}$, $v\ \vec{\tau}$ y $v\ \vec{\tau'}$ generarán la misma secuencia de variables en el lenguaje destino. Esto se vé en la regla en que cada variable generada está marcada con $\rho(\Delta, \vec{\tau})$, es decir una secuencia de *tuplicidades*, y esta marca es la misma que se obtendría con $\rho(\Delta, \vec{\tau'})$; generando así la misma secuencia de variables.

Es importante notar qué sucede con las variables monomórficas. En ese caso la instanciación $\vec{\tau}$ será una secuencia vacía pues no habrá variables de tipo que

instanciar. Luego, para las variables monomórficas se generará siempre la misma secuencia de variables del lenguaje objeto ya que éstas no estarán marcadas por los valores de las tuplicidades.

Por último el conjunto de instanciaciones es simplemente $(v, \vec{\tau})$, pues es la única instanciación que hay en la expresión.

$$\text{(AR-AP)} \quad \frac{\Delta \mid \Gamma \vdash e : \tau \to \eta \rightarrowtail [e'_i]_{i:1..n} \qquad \Delta \mid \Gamma' \vdash f : \tau \rightarrowtail [f'_j]_{j:1..m>0}}{\Delta \mid \Gamma, \Gamma' \vdash e\ f_\tau : \eta \rightarrowtail [e'_i\ f'_1 ... f'_m]_{i:1..n}}$$

En una aplicación el resultado depende de los resultados de la expresión de tipo funcional y del argumento. Al igual que sucedió con los tipos en la sección anterior es necesario separar una aplicación en dos casos. El primero, el que trata esta regla, es cuando el argumento de la función genera una secuencia de una o más expresiones. En este caso, como la función tiene como argumento una expresión de tipo τ y aunque genera una secuencia de funciones como resultado, todas poseerán los mismos tipos para sus argumentos (ver regla (TAR-FUN)); por lo tanto, es posible aplicar a cada una de las funciones resultantes de procesar e la misma secuencia de argumentos.

$$\text{(AR-APS)} \quad \frac{\Delta \mid \Gamma \vdash e : \tau \to \eta \rightarrowtail [e'_i]_{i:1..n} \qquad \Delta \mid \Gamma' \vdash f : \tau \rightarrowtail []}{\Delta \mid \Gamma \vdash e\ f_\tau : \eta \rightarrowtail [\texttt{force}\ e'_i]_{i:1..n}}$$

El segundo caso es cuando el argumento genera una secuencia vacía. Aquí la función e generará como resultado una secuencia de valores de tipo `susp` η'_i. Como la aplicación fuerza a que se evalúe el cuerpo de la función y aquí se están eliminando todas las aplicaciones, es necesario introducir un nuevo operador, llamado `force`, que fuerce la evaluación de los valores que se encuentran en suspensión. En el capítulo 4 se da una semántica precisa para esta operación y se explica con más detalle.

Notar que existe una diferencia extra entre esta regla y la anterior. Los conjunto de instanciaciones capturados por el procesamiento de las premisas, denotados con Γ y Γ', no aparecen de la misma manera en la conclusión de cada regla. En la primera aparecen ambos conjuntos mientras que en la segunda sólo aparece Γ. Esto es simplemente porque en esta última regla las variables resultantes capturadas por Γ' no aparecerán en la expresión e', y si bien el código será equivalente es posible detectarlo en esta etapa.

Otra particularidad que vale la pena notar es la necesidad de las marcas de tipos. La especificación del algoritmo en reglas guiadas por la estructura sintáctica de la expresión tiene como objetivo una traducción sencilla a una implementación. En esta regla puede verse entonces que si el algoritmo necesita como entrada el tipo de la expresión, el llamado recursivo para procesar el argumento no podría ser tan inmediato de resolver si no existiera la marca de su tipo (τ) embebida en la expresión original.

$$\text{(AR-LAM)}\quad \frac{\Delta \mid \Gamma \vdash e : \eta \rightarrowtail [e'_i]_i \quad \Delta \mid \Gamma' \vdash v : \tau \rightarrowtail [v'_j]_{j:1..n>0}}{\Delta \mid \Gamma_v \vdash \lambda v.e : \tau \rightarrow \eta \rightarrowtail [(\lambda v'_1 ... v'_n . e'_i)]_i}$$

De la misma manera que sucedía con las aplicaciones, el procesamiento de una función se divide en dos casos. El primero de ellos es cuando el tipo del argumento genera al menos un tipo como resultado y el otro es cuando se obtiene una secuencia vacía de tipos para los argumentos.

En el primer caso, como vimos en la regla (TAR-FUN), se resuelve currificando la función para los diferentes argumentos y generando una secuencia de funciones para cada una de las expresiones obtenidas al procesar el cuerpo de la función.

$$\text{(AR-LAMS)}\quad \frac{\Delta \mid \Gamma \vdash e : \eta \rightarrowtail [e'_i]_i \quad \Delta \mid \Gamma' \vdash v : \tau \rightarrowtail []}{\Delta \mid \Gamma_v \vdash \lambda v.e : \tau \rightarrow \eta \rightarrowtail [\$e'_i]_i}$$

En el segundo caso, al no haber ninguna variable generada por la variable de la función, es cuando se eliminan por completo los argumentos de la función. Eliminar simplemente la función nos trae además un problema semántico. El cuerpo de una función no es evaluado hasta que ésta no se aplica (ver semántica en el capítulo 4). Para seguir manteniendo esta situación se introduce el operador \$ que suspende la evaluación de una función hasta que se le aplica la operación `force` [Okasaki, 1998].

En estas últimas reglas, las instanciaciones que se obtienen para la variable v carecen de sentido, pues son variables monomórficas. Es por esto que no se utiliza esta información y luego se descarta del conjunto de instanciaciones, denotado Γ_v.

$$\text{(AR-STUPLE)}\quad \frac{\Delta \mid \Gamma \vdash e.i : \tau.i \rightarrowtail [e.i'_j]_{j:1..m_i}}{\Delta \mid \Gamma \vdash (e)^S_n : (\tau)^S_n \rightarrowtail [e.1'_j]_{j:1..m_1} \Vdash ... \Vdash [e.n'_j]_{j:1..m_n}}$$

La eliminación de una tupla estática consiste en la concatenación de todas las secuencias obtenidas de procesar cada una de las componentes de la tupla. El conjunto de instanciaciones de una tupla estática es la unión de todos los conjuntos obtenidos al procesar sus componentes.

$$\text{(AR-DTUPLE)}\quad \frac{\Delta \mid \Gamma \vdash e.i : \tau.i \rightarrowtail [e.i'_j]_{j:1..m_i}}{\Delta \mid \Gamma \vdash (e)^D_n : (\tau)^D_n \rightarrowtail [(e.1'_1, ..., e.1'_{m_1}, ..., e.n'_{m_n})_{m_1+...+m_n}]}$$

Las tuplas dinámicas son las que generan como resultado las únicas tuplas que posee el lenguaje destino. Sin embargo no es posible dejar la tupla en el estado en que se encuentra sino que es necesario procesar las componentes de ésta. Como resultado se obtiene una tupla de diferente aridad a la original donde las componentes son la concatenación de las secuencias obtenidas al procesar todas las componentes de la tupla dinámica.

$$\text{(AR-SPROJ)}\quad \frac{\Delta \mid \Gamma \vdash e : (\tau)_n^S \rightarrowtail [e'_i]_i \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \mid \Gamma \vdash \pi^S_{p,n}\ e_{(\tau)_n^S} : \tau.p \rightarrowtail [e'_{k+j}]_{j:1..\rho(\Delta,\tau.p)}}$$

La aplicación de una proyección estática sobre una tupla estática también debe ser eliminada. Si se desea proyectar la componente p-ésima de una tupla estática se obtiene como resultado la secuencia obtenida al procesar la p-ésima componente de la tupla estática.

Si bien es cierto que, al momento de evaluar esta expresión, el resultado de la computación de e será una tupla, sintácticamente hablando e no es una tupla. De modo que para obtener la secuencia correspondiente a la p-ésima componente de la tupla estática es necesario procesar toda la expresión e y extraer de la secuencia que resulte de ésta sólo la subsecuencia necesaria.

$$\text{(AR-DPROJ)}\quad \frac{\Delta \mid \Gamma \vdash e : (\tau)_n^D \rightarrowtail [e'] \quad m = \sum_s \rho(\Delta, \tau.s) \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \mid \Gamma \vdash \pi^D_{p,n}\ e_{(\tau)_n^D} : \tau.p \rightarrowtail [\pi_{k+j,m} e']_{j:1..\rho(\Delta,\tau.p)}}$$

Las proyecciones dinámicas sufren una suerte diferente. Éstas deben permanecer en la expresión resultante. El problema que aquí se enfrenta es que la tupla dinámica cambia de aridad al ser procesada. Si se está intentando proyectar la componente p-ésima de una tupla estática hay dos cuestiones a resolver. La primera es determinar la nueva posición en la tupla resultado y la segunda es decidir qué sucede si la componente p genera más de un valor en la tupla del lenguaje destino. El primero de estos problemas se resuelve calculando la tuplicidad de los tipos de las componentes anteriores a p.

El segundo tiene un tratamiento particular. Si la componente que se desea proyectar genera una secuencia de valores, entonces el resultado de la proyección de ésta generará una secuencia de proyecciones. Cada una de éstas proyectan cada elemento de la secuencia generada por la p-ésima componente.

$$\text{(AR-LET)}\quad \frac{\Delta \mid \Gamma \vdash f : \eta \rightarrowtail [f'_t]_t \quad (\forall \vec{\tau}_{i:1..m} \in \Gamma(v))(\Delta; (\vec{t}, \vec{\tau_i}) \mid \Gamma'_i \vdash e : \tau \rightarrowtail [e'_i]_{j:1..n_i})}{\begin{array}{l}\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash \texttt{let}\ v_{\forall \vec{t}.\tau} = \Lambda \vec{t}.e\ \texttt{in}\ f : \eta \rightarrowtail [\,\texttt{let}\ v_{\rho(\Delta,\vec{\tau_1}),1} = e'_{1_1} \quad \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v_{\rho(\Delta,\vec{\tau_1}),n_1} = e'_{1_{n_1}} \quad \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v_{\rho(\Delta,\vec{\tau_m}),n_m} = e'_{m_{n_m}}\ \texttt{in}\ f'_t\ ]_t\end{array}}$$

La regla para el tratamiento de un `let` es la más complicada de todas. Para explicar su funcionamiento es necesario enfocarla desde una visión algorítmica. Para el procesamiento de un let, primero se obtiene la secuencia de expresiones correspondientes a f; esto genera como resultado el conjunto de instanciaciones de todas las variables libres de f, es decir Γ. A partir de aquí se pueden obtener todas las instanciaciones para la variable polimórfica v, denotadas con $\Gamma(v)$. Para cada instanciación con diferente *tuplicidad*, se genera entonces una secuencia de

expresiones polimórficas. El resultado de procesar un `let` es entonces un nuevo `let` con todas las versiones polimórficas para cada instanciación que posea una *tuplicidad* diferente.

Los nombres de las variables utilizados son los que se obtendrán al procesar cada variable ya que, como las tuplicidades involucradas son las que se obtienen luego de procesar el cuerpo del `let`, se asegura que las variables queden ligadas a la definición correspondiente así como también que no hay variables definidas que no vayan a poseer su correspondiente utilización.

Para sumarizar todas las reglas explicadas anteriormente, estas se pueden observar en su totalidad en la figura 3.3.

$$\text{(AR-VAR)}\quad \Delta \mid (v,\vec{\tau}) \vdash v\ \vec{\tau} : \eta \rightarrowtail [v_{\rho(\Delta,\vec{\tau}),i}]_{i:1..\rho(\Delta,\eta)}$$

$$\text{(AR-AP)}\quad \frac{\Delta \mid \Gamma \vdash e : \tau \to \eta \rightarrowtail [e'_i]_{i:1..n} \quad \Delta \mid \Gamma' \vdash f : \tau \rightarrowtail [f'_j]_{j:1..m>0}}{\Delta \mid \Gamma, \Gamma' \vdash e\ f_\tau : \eta \rightarrowtail [e'_i\ f'_1 ... f'_m]_{i:1..n}}$$

$$\text{(AR-APS)}\quad \frac{\Delta \mid \Gamma \vdash e : \tau \to \eta \rightarrowtail [e'_i]_{i:1..n} \quad \Delta \mid \Gamma' \vdash f : \tau \rightarrowtail []}{\Delta \mid \Gamma \vdash e\ f_\tau : \eta \rightarrowtail [\texttt{force}\ e'_i]_{i:1..n}}$$

$$\text{(AR-LAM)}\quad \frac{\Delta \mid \Gamma \vdash e : \eta \rightarrowtail [e'_i]_i \quad \Delta \mid \Gamma' \vdash v : \tau \rightarrowtail [v'_j]_{j:1..n>0}}{\Delta \mid \Gamma_v \vdash \lambda v.e : \tau \to \eta \rightarrowtail [(\lambda v'_1 ... v'_n . e'_i)]_i}$$

$$\text{(AR-LAMS)}\quad \frac{\Delta \mid \Gamma \vdash e : \eta \rightarrowtail [e'_i]_i \quad \Delta \mid \Gamma' \vdash v : \tau \rightarrowtail []}{\Delta \mid \Gamma_v \vdash \lambda v.e : \tau \to \eta \rightarrowtail [\$e'_i]_i}$$

$$\text{(AR-SPROJ)}\quad \frac{\Delta \mid \Gamma \vdash e : (\tau)^S_n \rightarrowtail [e'_i]_i \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \mid \Gamma \vdash \pi^S_{p,n}\ e_{(\tau)^S_n} : \tau.p \rightarrowtail [e'_{k+j}]_{j:1..\rho(\Delta,\tau.p)}}$$

$$\text{(AR-DPROJ)}\quad \frac{\Delta \mid \Gamma \vdash e : (\tau)^D_n \rightarrowtail [e'] \quad m = \sum_{s} \rho(\Delta, \tau.s) \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \mid \Gamma \vdash \pi^D_{p,n}\ e_{(\tau)^D_n} : \tau.p \rightarrowtail [\pi_{k+j,m} e']_{j:1..\rho(\Delta,\tau.p)}}$$

$$\text{(AR-STUPLE)}\quad \frac{\Delta \mid \Gamma \vdash e.i : \tau.i \rightarrowtail [e.i'_j]_{j:1..m_i}}{\Delta \mid \Gamma \vdash (e)^S_n : (\tau)^S_n \rightarrowtail [e.1'_j]_{j:1..m_1} \parallel ... \parallel [e.n'_j]_{j:1..m_n}}$$

$$\text{(AR-DTUPLE)}\quad \frac{\Delta \mid \Gamma \vdash e.i : \tau.i \rightarrowtail [e.i'_j]_{j:1..m_i}}{\Delta \mid \Gamma \vdash (e)^D_n : (\tau)^D_n \rightarrowtail [(e.1'_1, ..., e.1'_{m_1}, ..., e.n'_{m_n})_{m_1+...+m_n}]}$$

$$\text{(AR-LET)}\quad \frac{\Delta \mid \Gamma \vdash f : \eta \rightarrowtail [f'_t]_t \quad (\forall \vec{\tau}_{i:1..m} \in \Gamma(v))(\Delta; (\vec{t}, \vec{\tau}_i) \mid \Gamma'_i \vdash e : \tau \rightarrowtail [e'_i]_{j:1..n_i})}{\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash \begin{array}{l}\texttt{let}\ v_{\forall \vec{t}.\tau} = \Lambda \vec{t}.e \\ \texttt{in}\ f\end{array} : \eta \rightarrowtail [\begin{array}{ll}\texttt{let}\ v_{\rho(\Delta,\vec{\tau_1}),1} & = e'_{1_1} \quad ... \\ & v_{\rho(\Delta,\vec{\tau_1}),n_1} = e'_{1_{n_1}} \quad ... \\ & v_{\rho(\Delta,\vec{\tau_m}),n_m} = e'_{m_{n_m}}\ \texttt{in}\ f'_t\ ]_t\end{array}}$$

Figura 3.3: Reglas de especificación del algoritmo de *Arity Raising*

3.5 Ejemplos

En esta sección se mostrarán algunos ejemplos del funcionamiento del algoritmo de *Arity Raising*. Comencemos con algunos ejemplos sencillos en forma detallada para luego avanzar sobre ejemplos más complicados de una manera más intuitiva. Veamos primero como trabaja el algoritmo sobre algunas proyecciones.

Ejemplo 3.19. En este ejemplo se muestra como son eliminadas las proyecciones estáticas "entrando" en la tupla estática para quedarse con la componente necesaria. La siguiente función toma un par estático cuya segunda componente es otro par estático y devuelve la primer componente de este último.

$$\lambda p.\pi^S_{1,2}\ (\pi^S_{2,2}\ p)$$

Veamos primero la expresión correspondiente en el lenguaje objeto:

$$\lambda p.\pi^S_{1,2}\ (\pi^S_{2,2}\ p_{(\iota,(\iota,\iota)^S_n)^S_n})_{(\iota,\iota)^S_n} : (\iota,(\iota,\iota)^S_n)^S_n \to \iota$$

Como $\vdash (\iota,(\iota,\iota)^S_n)^S_n \xrightarrow{tar} [\iota,\iota,\iota]$ entonces $\rho(\Delta,(\iota,(\iota,\iota)^S_n)^S_n) = 3$ para cualquier Δ. Luego utilizando la regla (AR-VAR) tenemos

$$\Delta \mid\vdash p : (\iota,(\iota,\iota)^S_n)^S_n \rightarrowtail [p_1,p_2,p_3]$$

(notar que aquí se ha omitido $\vec{\tau}$ pues como p es una variable monomórfica esta secuencia es vacía). Luego, aplicando (AR-SPROJ), obtenemos la siguiente expresión.

$$\Delta \mid\vdash \pi^S_{2,2}\ p_{(\iota,(\iota,\iota)^S_n)^S_n} : (\iota,\iota)^S_n \rightarrowtail [p_2,p_3]$$

Aplicando nuevamente (AR-SPROJ) obtenemos:

$$\Delta \mid\vdash \pi^S_{1,2}\ (\pi^S_{2,2}\ p_{(\iota,(\iota,\iota)^S_n)^S_n})_{(\iota,\iota)^S_n} : (\iota,(\iota,\iota)^S_n)^S_n \rightarrowtail [p_2]$$

Finalmente aplicando (AR-LAM) se obtiene:

$$\Delta \mid\vdash \lambda p.\pi^S_{1,2}\ (\pi^S_{2,2}\ p_{(\iota,(\iota,\iota)^S_n)^S_n})_{(\iota,\iota)^S_n} : (\iota,(\iota,\iota)^S_n)^S_n \rightarrowtail [\lambda p_1.\lambda p_2.\lambda p_3.\ p_2]$$

Resumiendo, la expresión obtenida luego de *Arity Raising* es:

$$\lambda p_1.\lambda p_2.\lambda p_3.\ p_2$$

En los ejemplos subsiguientes se omitirá todo el desarrollo detallado de las aplicaciones de las reglas haciendo énfasis solamente en aquellos pasos que sean más relevantes.

Ejemplo 3.20. El tratamiento de las proyecciones dinámicas que manejan tuplas que pueden poseer tuplas estáticas en su interior requiere cierto reacomodamiento de los índices utilizados. En el siguiente ejemplo se ilustra tal situación. Tenemos una expresión como:

$$\lambda x.\pi^S_{1,2}\ (\pi^D_{2,2}\ (x, x))$$

Aquí podemos observar que x es un par estático pero, al ser eliminado, la proyección dinámica debería cambiar sus índices. Luego de aplicar *Arity Raising* se puede verificar tal situación:

$$\lambda x_1.\lambda x_2.(\pi^D_{3,4}\ (x_1, x_2, x_1, x_2))$$

La proyección dinámica, que proyectaba la segunda componente de un par, fue cambiada para proyectar la tercer componente de una tupla de cuatro elementos, obteniendo así el mismo resultado que antes.

Veamos unos ejemplos más complicados cuyo objetivo es observar cómo es eliminada la tupla vacía de una expresión.

Ejemplo 3.21. Una situación muy común con la que debe enfrentarse el algoritmo de *Arity Raising*, debido a que algunas de las salidas del procesamiento de *type specialization* tienen esa forma, es la siguiente. En este ejemplo la constante 2 puede verse como un elemento del tipo ι.

$$\texttt{let}\ f\ \texttt{=}\ ()^S, \lambda x.x)^S\ \texttt{in}\ (\pi^S_{2,2}\ f)\ 2$$

Aplicando *Arity Raising* se obtiene una expresión en la que se vé cómo se ha eliminado la tupla vacía.

$$\texttt{let}\ f\ \texttt{=}\ \lambda x.x\ \texttt{in}\ f\ 2$$

La definición del par f se ha comprimido en una única definición de función y en cada utilización de esta componente se deja únicamente el llamado a la función.

Ejemplo 3.22. Veamos ahora otro ejemplo de eliminación de tuplas vacías. Aquí tenemos una función polimórfica que es utilizadas con dos *tuplicidades* diferentes. En el primer caso se la utiliza contra un par estático de elementos de tipo ι y por la definición de la función retornará uno de los elementos. En la segunda utilización la función es instanciada con un tipo que dará como resultado una tupla vacía.

$$\texttt{let}\ f\ \texttt{=}\ \lambda p.\pi^S_{2,2}\ p\ \texttt{in}\ (f\ (1, 2)^S, f\ (3, ()^S)^S)^S$$

El segundo llamado a la función f es eliminado, ya que el tipo del resultado de esa aplicación es una tupla estática vacía, y la definición de la función es restringida solamente a la tuplicidad utilizada.

$$\texttt{let}\ f\ \texttt{=}\ \lambda p_1.\lambda p_2.p_2\ \texttt{in}\ f\ 1\ 2$$

Ejemplo 3.23. Otro ejemplo, específicamente de polimorfismo, es el siguiente. Aquí una función polimórfica f es utilizada en dos lugares con tuplicidades diferentes.

$$\texttt{let } f \texttt{ = } \lambda x,1 \texttt{ in } (f\ 2, f\ (3,4)^S)$$

La definición de la función se extiende generando dos funciones, una para cada tuplicidad detectada. El algoritmo de *Arity Raising* produce entonces:

$$\begin{array}{lll} \texttt{let} & f_1 & \texttt{= } \lambda x.1 \\ & f_2 & \texttt{= } \lambda x_1.\lambda x_2.1 \texttt{ in} \quad (f_1\ 2, f_2\ 3\ 4) \end{array}$$

Capítulo 4

Corrección

¡Alguien que me explique cómo congeniar próceres con ravioles!

Susana Clotilde Chirusi (Susanita), Mafalda
Joaquín Salvador Lavado (Quino)

Hasta el momento se ha visto toda la manipulación que se realiza sobre el código y la conversión que se aplica sobre los tipos, pero para que el trabajo tenga sentido falta ver que todo lo que se ha planteado en los capítulos anteriores es, desde algún punto de vista, correcto.

En particular en este capítulo se muestra en primer lugar que la transformación de código y la de tipos son correctas en el sentido que las expresiones generadas poseen los tipos generados. En segundo lugar se define la semántica para los lenguajes objeto y destino, y se muestra que el algoritmo propuesto preserva la semántica del lenguaje objeto

4.1 Corrección de tipos

En esta sección veremos que la secuencia de expresiones obtenidas al aplicar el algoritmo de *Arity Raising* sobre una expresión particular, posee los tipos de la secuencia que se obtiene al procesar el tipo de la expresión original.

Antes de enunciar el teorema formalmente necesitamos definir y plantear una serie de propiedades que hablan sobre el comportamiento del *Arity Raising* sobre tipos.

En primer lugar definimos la noción de *compatibilidad*. En el capítulo anterior vimos que la regla (TAR-SCHM) era la que eliminaba la posibilidad de que la relación fuera funcional. Sin embargo podemos restringir el uso de esta regla para un caso particular, lo que lleva a que la aplicación de esta regla sea de carácter funcional. Lo primero que hay que observar es que la posibilidad de elegir diferentes resultados, al aplicar la regla, es consecuencia directa de poder decidir en ese momento la *tuplicidad* de las variables cuantificadas. Fijando esta condición podemos tener una regla funcional. Si bien es posible fijar para cada variable una *tuplicidad*, una mejor aproximación es, dada una instanciación $\vec{\eta}$, fijar cada variable cuantificada con la *tuplicidad* del tipo que le asigna la instanciación. Definimos formalmente este concepto.

Definición 4.1. Sea $\vec{\eta}$ una instanciación de un esquema de tipo $\sigma = \forall \vec{t}.\tau$. Una derivación $\Delta \vdash \forall \vec{t}.\tau \xrightarrow{tar} [\sigma'_i]_{i:1..n}$ se dice *compatible* con $\vec{\eta}$, si vale que

$$\Delta; (\vec{t}, \vec{\eta}) \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n}$$

donde $\sigma'_i = \forall \vec{t'}.\tau'_i$.

A partir de aquí podemos, utilizando la definición anterior, dar un corolario del lema 3.12, que plantea la relación que existe entre un tipo polimórfico y sus instanciaciones contra los resultados de aplicar *Arity Raising* sobre ellos. Este corolario es muy importante ya que muestra que si poseemos una instancia de un tipo polimórfico, aplicar *Arity Raising* sobre ambos (respetando compatibilidad) genera una secuencia de tipos pólimorficos y una secuencia de tipos que son instancia de éstos.

Corolario 4.2. *Sea* $\tau \leq_{\vec{\eta}} \sigma$ *entonces si* $\Delta \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..n}$ *es compatible con* $\vec{\eta}$, $\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n}$ *y* $\tau'_i \leq \sigma'_i$

Para continuar con el objetivo de esta sección, que es mostrar corrección de tipos, veamos lo que necesitamos para dar un juicio sobre los tipos de las expresiones del lenguaje destino. En la figura 3.1 se muestra el sistema de tipos que debemos utilizar. El primer paso que hay que dar es encontrar el contexto (la asignación de variables libres de la expresión a tipos) adecuado. Es claro que el contexto a utilizar dependerá del contexto original con que se realiza el tipado de la expresión en el lenguaje objeto. En la siguiente definición se presenta el conjunto que se utilizará para asignar tipo a las variables libres de una expresión en el lenguaje destino. Intuitivamente, este conjunto está formado "aplicando" *Arity Raising* al contexto utilizado al tipar la expresión objeto.

Definición 4.3. Sea A una asignación de variables a tipos, Γ una colección de instanciaciones y Δ una asignación de aridades. Entonces se define $A \times \Gamma_\Delta$ de la siguiente manera:
$v_{\rho(\Delta,\vec{\eta}),i} : \sigma'_i \in A \times \Gamma_\Delta$, si y sólo si, $v : \sigma \in A$, $(v, \vec{\eta}) \in \Gamma$, $\Delta \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..n}$ y esta última es compatible con $\vec{\eta}$.

Es importante observar que por la definición del conjunto, ya que se hace de una manera constructiva, es posible afirmar *siempre* la existencia del mismo.

Una vez obtenido este conjunto debemos observar que cumple con las características necesarias para ser un contexto para el sistema de tipos. En primer lugar veremos el comportamiento que presenta este conjunto al agrandarse su conjunto de instanciaciones. Incluir una instanciación en el conjunto Γ incluirá, potencialmente, nuevas asignaciones de variables a tipos en el resultado, pero lo que es importante aquí es que estas nuevas asignaciones no sean incoherentes con las anteriores, es decir que no asignen un tipo diferente para una misma variable. La siguiente propiedad establece este hecho.

Propiedad 4.4. *Sea A una asignación de tipos, Γ,Γ' colecciones de instanciaciones y Δ una asignación de tuplicidades. Luego, se cumple que $A \times \Gamma_\Delta \subseteq A \times (\Gamma, \Gamma')_\Delta$ y este último está bien formado.*

Un último aspecto a tener en cuenta sobre este conjunto es el hecho de que el mismo sólo hace un juicio importante con respecto a las *tuplicidades* de las variables libres de la expresión, sin que el tipo particular que ellas tomen sea demasiado importante. Los tipos que sí son relevantes son los que se encuentran en el conjunto original A, pero los que se encuentran en Γ son utilizados solamente por su *tuplicidad*. Este hecho se demuestra en la siguiente propiedad.

Propiedad 4.5. *Sea τ tal que $\rho(\Delta, \tau) = n$. Si $t \notin TV(A)$ y $t \notin \Delta$, entonces $A \times \Gamma_{\Delta,(t,n)} = A \times \Gamma[t/\tau]_\Delta$*

Corolario 4.6. *Si $t \notin TV(A)$ y $t \notin TV(\Gamma)$ entonces $(\forall n)(A \times \Gamma_\Delta = A \times \Gamma_{\Delta,(t,n)})$*

Para finalizar esta sección se enuncia el teorema que establece la corrección de tipos del sistema de *Arity Raising* planteado en el capítulo anterior. Básicamente el teorema dice que dada una expresión e de tipo τ, las secuencias de expresiones y tipos obtenidas son correctas en el sentido que los tipos corresponden a las expresiones de acuerdo al sistema de tipos dado para el lenguaje destino.

Teorema 4.7. *Sea A tal que $A \vdash e \rightsquigarrow e' : \tau$, entonces $(\forall \Gamma)(\forall \Delta)(\Delta \mid \Gamma \vdash e' : \tau \rightarrowtail [e''_i]_{i:1..n} \Rightarrow \Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n} \wedge (\forall i : 1..n)(A \times \Gamma_\Delta \vdash e''_i : \tau'_i))$.*

Hemos recorrido la mitad del camino para tener corrección, viendo que las expresiones obtenidas están "bien" construidas. Falta además tener la certeza de que estas nuevas expresiones realizan el mismo trabajo que las anteriores.

4.2 Semántica Operacional

Antes de continuar con la corrección operacional es necesario dar semántica a los lenguajes objeto y destino. Si queremos demostrar que las expresiones "hacen" lo mismo, primero tenemos que definir formalmente qué es lo que significa una expresión. En esta sección se definen entonces, la semántica operacional del lenguaje objeto y del lenguaje destino.

4.2.1 Lenguaje Objeto

El lenguaje objeto es un lambda cálculo tradicional extendido con construcciones estándar; es por ello que la semántica definida a continuación es ampliamente conocida.

$$\text{(SO-LAM)}\ \frac{}{\lambda v.e \Downarrow_o \lambda v.e}$$

$$\text{(SO-DTUPLE)}\ \frac{}{(e)_n^D \Downarrow_o (e)_n^D}$$

$$\text{(SO-STUPLE)}\ \frac{(e.i \Downarrow_o f.i)_{i:1..n}}{(e)_n^S \Downarrow_o (f)_n^S}$$

$$\text{(SO-APP)}\ \frac{e \Downarrow_o \lambda v.e' \quad e'[v/f] \Downarrow_o f'}{e\ f_\tau \Downarrow_o f'}$$

$$\text{(SO-DPROJ)}\ \frac{e \Downarrow_o (e')_n^D \quad e'.i \Downarrow_o f}{\pi_{i,n}^D e \Downarrow_o f}$$

$$\text{(SO-SPROJ)}\ \frac{e \Downarrow_o (e')_n^S}{\pi_{i,n}^S e \Downarrow_o e'.i}$$

$$\text{(SO-LET)}\ \frac{f[v/e] \Downarrow_o f'}{\texttt{let}\ v_\sigma\ \texttt{=}\ e\ \texttt{in}\ f \Downarrow_o f'}$$

Figura 4.1: Semántica operacional del lenguaje objeto

Definición 4.8. Sean e y e' dos expresiones del lenguaje objeto, se define la *semántica operacional* del lenguaje objeto, denotada con $e \Downarrow_o e'$, con la relación descrita por las reglas de la figura 4.1.

La semántica definida no posee ninguna restricción o característica particular, lo que era uno de los objetivos del trabajo. En particular es una semántica tradicional que no reduce bajo una expresión λ.

En los trabajos previos hechos de *Arity Raising*, la semántica dada a todas las tuplas era estricta. Es decir que al encontrarse con una tupla, se reducen todas sus componentes antes de continuar con la evaluación del programa. En este trabajo sólo las tuplas estáticas han sido definidas con reducción estricta, mientras que las tuplas dinámicas se tratan en forma *lazy*. Esta decisión está fundamentada en la elección de eliminar todas las tuplas estáticas ya que su eliminación puede forzar a evaluar componentes que de otra manera jamás serían evaluados.

Otro aspecto a marcar es que la aplicación de funciones, tanto la explícita como la embebida dentro de un `let`, utiliza el mecanismo de *call by name*. Al ser el lenguaje definido un lenguaje funcional esta es una elección tradicional para un lenguaje con evaluación *lazy* [Jones et al., 1999].

$$\text{(SD-LAM)}\ \frac{}{\lambda v.e \Downarrow_d \lambda v.e}$$

$$\text{(SD-SUSP)}\ \frac{}{\$e \Downarrow_d \$e}$$

$$\text{(SD-TUPLE)}\ \frac{}{(e)_n \Downarrow_d (e)_n}$$

$$\text{(SD-APP)}\ \frac{e \Downarrow_d \lambda v.e' \quad e'[v/f] \Downarrow_d f'}{e\ f \Downarrow_d f'}$$

$$\text{(SD-PROJ)}\ \frac{e \Downarrow_d (e')_n \quad e.i' \Downarrow_d f}{\pi_{i,n} e \Downarrow_d f}$$

$$\text{(SD-LET)}\ \frac{f[\vec{v}/\vec{e}] \Downarrow_d f'}{\texttt{let}\ v_1 = e_1; ...; v_n = e_n\ \texttt{in}\ f \Downarrow_d f'}$$

$$\text{(SD-FORCE)}\ \frac{e \Downarrow_d \$e' \quad e' \Downarrow_d f}{\texttt{force}\ e \Downarrow_d f}$$

Figura 4.2: Semántica operacional del lenguaje destino

4.2.2 Lenguaje Destino

El lenguaje destino posee además de las construcciones estándar, dos construcciones nuevas: las operaciones \$ y `force`. Estas dos operaciones, definidas por [Okasaki, 1998], son mecanismos para la introducción de evaluación *lazy* y en este trabajo son utilizadas para marcar explícitamente algunas expresiones que carecerían de este comportamiento.

El operador \$ suspende la evaluación de una expresión hasta que ésta es forzada. De esta manera la ejecución de una expresión $\$e$, para cualquier e, ha llegado a su fin. Dicho de una manera más formal, la expresión se encuentra en forma normal. La regla (SD-SUSP) muestra que la ejecución de una expresión de tal forma ha concluido. Para forzar la ejecución de una expresión suspendida con \$, se utiliza el operador `force`. Al aplicar este operador sobre una expresión en suspensión, esta finalmente se evalúa. La regla (SD-FORCE) muestra el comportamiento que posee la construcción `force`.

Definimos entonces la relación de semántica para el lenguaje destino

Definición 4.9. Sea e una expresión del lenguaje destino, se define la *semántica operacional* del lenguaje destino, denotada con $e \Downarrow_d e'$, con la relación definida por las reglas de la figura 4.2

La semántica definida para el lenguaje destino, si bien es estándar en la no reducción bajo lambda, provee además un esquema *lazy* para el manejo de las tuplas. Por otra parte presenta un manejo especial de suspensiones.

La necesidad de introducir este operador estuvo motivada en el procesamiento usual de una función. Si bien se ha explicado que lo que se realiza con una función es *currificar* ésta según la aridad de la tupla recibida como parámetro, esto lleva a que si una función recibe una tupla 0-aria se eliminen todos sus parámetros, perdiendo además su característica de no reducción, situación que no es deseada a la hora de comparar las semánticas.

4.3 Corrección Operacional

Ahora que se ha definido la semántica de los lenguajes hay que ver como se comporta el algoritmo de *Arity Raising* con respecto a estas definiciones. El primer paso a dar es analizar el comportamiento del algoritmo propuesto con respecto a las sustituciones. Como se puede ver en la definición de las semánticas, éstas están fuertemente ligadas a la definición de sustitución de variables por términos. En particular la semántica del lenguaje objeto depende además de la sustitución de tipos por ser éste un lenguaje con tipado explícito.

Veamos primero qué sucede con la sustitución de tipos. El siguiente lema afirma que sustituir una variable de tipo libre en el término y en el tipo a procesar es irrelevante siempre y cuando se haya asumido la *tuplicidad* correcta para la variable en cuestión.

Lema 4.10. *Si* $\Delta; (t, \eta) \mid \Gamma \vdash e : \tau \rightarrowtail [e'_i]_{i:1..n}$ *entonces* $\Delta \mid \Gamma[t/\eta] \vdash e[t/\eta] : \tau[t/\eta] \rightarrowtail [e'_i]_{i:1..n}$

La sustitución de variables por términos es, obviamente, mucho más complicada. Sin embargo, en esencia, el teorema siguiente establece que sustituir una variable v por un término e antes de procesar es equivalente a sustituir, la secuencia variables generadas por v por la secuencia de términos resultante de procesar el término e, luego de procesar.

La parte complicada del teorema radica en el tratamiento del polimorfismo. Una misma variable polimórfica puede, en distintos lugares del código, generar secuencias diferentes de variables. Es por ello que la generación de la secuencia de términos a sustituir debe ser particular para cada instanciación con diferente tuplicidad.

Lema 4.11. *Sea* $e' = \Lambda \vec{t}\,.e$ *tal que* $v \notin FV(e)$, $v \in FV(f)$ *y* $(\forall \vec{\tau}_{i:1..m} \in \Gamma(v))$ $(\Delta; (\vec{t}\,, \vec{\tau}_i) \mid \Gamma'_i \vdash e : \eta \rightarrowtail [e'_{i_j}]_{j:1..n_i})$, *si* $\Delta \mid \Gamma \vdash f : \tau \rightarrowtail [f'_i]_{i:1..n}$ *entonces*

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash f[v/e'] : \tau \rightarrowtail [f'_i[v_{\rho(\vec{\Delta},\vec{\tau_1})}/\vec{e'_1}]...[v_{\rho(\vec{\Delta},\vec{\tau_m})}/\vec{e'_m}]]_{i:1..n}$$

Ahora que el comportamiento del algoritmo con respecto a las sustituciones ha sido demostrado, podemos avanzar con el teorema más importante de este capítulo y de la tesis en sí. El siguiente teorema establece que si es posible evaluar

una expresión en el lenguaje objeto, aplicar *Arity Raising* sobre esta expresión y luego ejecutar cada expresión obtenida en la secuencia, es equivalente a aplicar *Arity Raising luego* de evaluar la expresión original.

Dicho de otra manera, si el programa original luego de ser ejecutado produce *un* valor (carente de tuplas estáticas) entonces el programa obtenido luego de hacer *Arity Raising* producirá el *mismo* valor.

Teorema 4.12. *Si* $\Delta \mid \Gamma \vdash e : \tau \rightarrowtail [e'_i]_{i:1..n}$ *y* $e \Downarrow_o f$ *entonces existen términos* f_i *tales que* $e'_i \Downarrow_d f_i$ *y* $\Delta \mid \Gamma' \vdash f : \tau \rightarrowtail [f_i]_{i:1..n}$

Este teorema utiliza la noción de *simulación* [Milner, 1999] y puede entenderse intuitivamente como que el programa antes de ser procesado puede ser *simulado* por el programa obtenido luego de hacer *Arity Raising*. Ello garantiza que es seguro reemplazar el código anterior por el generado por el algoritmo.

Capítulo 5

Prototipo

Esa computadora sólo parece adaptarse bien a usted.[...]Trevize sonrió. Colocó las manos encima de los soportes y sintió la unión mental.

Los límites de la Fundación.
Isaac Asimov.

En este capítulo se muestra una implementación del algoritmo de *Arity Raising* propuesto. Para realizar este desarrollo se utilizó el lenguaje funcional Haskell [Jones et al., 1999]. La elección de este lenguaje está basada en varios puntos. En primer lugar la implementación de un sistema de reglas en este lenguaje es prácticamente directa por ser un lenguaje cuyo código fuente se asemeja a una especificación. Por otra parte, la sencillez para la construcción de parsers de lenguajes y la facilidad de manejo de estructuras de datos son esenciales para obtener un prototipo del algoritmo.

La implementación tuvo varios aspectos diferentes: primero se realizó la codificación de un parser para el lenguaje fuente, luego se implementó un algoritmo estándar de inferencia de tipos para luego ser modificado con las particularidades vistas en los capítulos anteriores y finalmente se procedió con la implementación del algoritmo de *Arity Raising*

5.1 Parsing

Se ha implementado un parser para un lenguaje similar al descripto en el capítulo 2. La diferencia principal radica en el hecho de que el parser implementado soporta constantes numéricas que no figuraban en la especificación original. Esta incorporación es una implementación de constantes del tipo ι definido en los tipos del lenguaje fuente. De aquí en adelante se interpretará el tipo ι como el tipo `Int`, cuyos elementos son las constantes numéricas.

A continuación se muestra la gramática utilizada para realizar el módulo de *parsing*. Se ha simplificado la presentación de la gramática, eliminando algunos aspectos técnicos de la misma y algunas ramas de símbolos no terminales no apropiadas para este caso.

TypedExp	:= *Expression TypeSignature*
Expression	:= *App* | *Binder*
App	:= *App PrimExpression* | *PrimExpression*
PrimExpression	:= *Proj Expression*
	| *Identifier*
	| *Integer*
	| `()` *Annot*
	| `(` *TypedExp* `)`
	| `(` *TypedExp*`,` ... `,` *TypedExp* `)` *Annot*
Binder	:= `\` *Identifier* `->` *Expression*
	| `let` *Decls* `in` *Expression*
Annot	:= `^S` | `^D`
Proj	:= `fst` *Annot*
	| `snd` *Annot*
	| `Proj_`*Integer*`_`*Integer Annot*
Decls	:= *Decl*`;` ... `;` *Decl*
Decl	:= *Identifier* `=` *Expression*
TypeSignature	:= `::` *TypeExp*
	| ϵ
TypeExp	:= *TypeFact* `->` *TypeExp*
	| *TypeFact*
TypeFact	:= `T_`*Integer*
	| `()` *Annot*
	| `(`*TypeExp*`,`...`,`*TypeExp*`)` *Annot*
	| `(`*TypeExp*`)`

El símbolo inicial de la gramática anterior es *TypedExp*, de manera que son válidos como entrada para el algoritmo los siguientes ejemplos de código fuente:

- ```
  let f = (\x->x)
  in (f (2,3),(f 3,4)^S)
  ```
- ```
  (fst (2,3)^S, \x->(x,x))
  ```
- ```
  \v->fst (fst (2,3)^S, \x->(x,x))
  ```

En la gramática anterior las mayoría de las expresiones posee tipado explícito, denotado esto con el noterminal *TypeSignature*. En los lugares donde aparece este símbolo puede haber *o no* una asignación explícita de tipo. De esta manera es posible escribir

```
(\x->x)::Int->Int
```

tanto como

```
(\x->x)
```
```

cuando se espere una expresión tipada. Es decir que por más que en la gramática se define una tupla como una secuencia de expresiones tipadas, no es necesario dar el tipo de éstas. Una tupla válida en el lenguaje de entrada puede ser entonces:

```
(2,3::Int,(\x->x),(\y->y)::Int->Int)
```

Un detalle más del lenguaje de entrada es que en los tipos explícitos es posible poner variables de tipo. Para utilizar una de estas variables se debe escribir `T_`*Integer*. Esta característica es importante para el *testing* del algoritmo; sin embargo su uso no es inherente al tema de la tesis y su explicación se obviará en este trabajo.

5.2 Lenguaje Fuente y Objeto

La aplicación del parser sobre un programa de entrada escrito en el código fuente genera una estructura de un tipo de datos que representa a tal programa.

A los efectos de la implementación, las únicas diferencias que existen entre el lenguaje fuente y el objeto son las marcas que se realizan en éste mediante el algoritmo de marcado descripto en el capítulo 2. Por este motivo el tipo de datos para el lenguaje fuente posee espacios para las marcas de tipo que se harán en una fase posterior.

El tipo de datos utilizado para representar los tipos monomórficos en el lenguaje fuente es el siguiente:

```
data SourceType =
    STInt
  | STVar   SourceTypeVble
  | STTuple Annot [ SourceType ]
  | STFun   SourceType SourceType
```

donde `SourceTypeVble` es el tipo de los identificadores de variables, y `Annot` es el tipo de las anotaciones. Éstos no se incluyen aquí pues no poseen mayor relevancia.

El tipo de datos para representar los esquemas de tipos polimórficos del lenguaje fuente es el siguiente:

```
data STypeScheme =
         SSForAll   SourceTypeVble STypeScheme
       | SSType     SourceType
```

Aquí básicamente se representa una secuencia de cuantificadores con variables seguido de un tipo monomórfico. Este diseño de los tipos de datos está guiado por la definición de los tipos τ y σ, del capítulo 2.

El tipo de datos para las expresiones del lenguaje fuente (y del lenguaje objeto) es el siguiente:

```
type STermVble       = String
type TypedSourceTerm = (STerm, SourceType)
data STerm      =
    Var [SourceType] STermVble
  | App SourceType STerm STerm
  | Lam STermVble STerm
  | Let [(STermVble, STypeScheme, STerm)] STerm
  | Tuple Annot [ STerm ]
  | Proj Annot Int Int SourceType STerm
  | Typed SourceTerm SourceType
  | CInt Int
```

donde, por ejemplo, la expresión $(\lambda v.v)2$ se representa con `App STUnknown (Lam "v"(Var "v")) (CInt 2)`, siendo `STUnknown` un espacio vacío que luego se decorará con la marca de tipo correspondiente.

5.3 Inferencia de tipos y marcas

Una vez realizado el parsing del texto de entrada, se posee la estructura de la expresión en el lenguaje fuente. Como el lenguaje posee la característica de permitir tipado explícito en algunas expresiones es posible que a esta altura ya existan algunas que estén marcadas con su tipo. Para el resto de las expresiones hay que obtener los tipos para poder realizar las marcas en los lugares que son necesarios.

Para la implementación de los sistemas presentados en este trabajo se utilizó la técnica de programación con *mónadas*[Wadler, 1993]. Si bien no se va a entrar en detalle con los tecnicismos de las mónadas, por el momento se puede entender una mónada como un mecanismo funcional para llevar un estado transparentemente a la computación de una función.

La función que realiza la tarea de marcado posee el siguiente tipo

```
tc :: Environment -> STerm -> STCM TypedSourceTerm
```

donde `Environment` es el contexto que posee el tipo de las variables libres que pudieran aparecer en la expresión a tipar (el conjunto A en la figura 2.1), `STerm` es dicha expresión (visto más arriba) y `TypedSourceTerm` es, como resultado de la función, el término con las marcas de tipo incorporadas, junto con su tipo. Para la implementación de esta función se utiliza la mónada `STCM`, la cual lleva en el estado las variables de tipo utilizadas hasta el momento y provee una primitiva esencial:

- `freshSType`: obtiene una nueva variable de tipo, o sea "fresca", que no ha sido utilizada hasta el momento.

La función `tc` implementa el sistema descripto en la figura 2.1. La implementación de este sistema es la estándar utilizada en inferencia de tipos, con el agregado de las marcas. Se mostrará a continuación, a manera de ejemplo, la implementación de dos reglas sencillas del sistema.

Veamos primero la implementación de la regla (MG-LAM).

```
tc g (Lam x e) =
   do t <- freshSType
      (te, t1) <- tc ((x,SSType t):g) e
      return (Lam x te, STFun t t1)
```

Más allá de los detalles de implementación se puede observar que para inferir el tipo de la expresión `(Lam x e)`, se crea una nueva variable de tipo (*fresh*) para la variable `x`, se infiere el tipo del cuerpo de la abstracción , `t1`, mientras se marca ésta (obteniendo `te`) y se retornan la expresión `Lam x te`, con las marcas realizadas, y su tipo. Si se observa la regla correspondiente, esta especifica básicamente lo mismo.

Veamos ahora que sucede con la regla (MG-APP).

```
tc g (App _ e1 e2) =
   do (te1, t1) <- tc g e1
      (te2, t2) <- tc g e2
      t <- freshSType
      unify t1 (STFun t2 t)
      return (App t2 te1 te2, t)
```

Aquí, más allá del esqueleto típico de inferencia de tipos, se observa que cuando se infieren los tipos de `e1` y `e2` se obtiene las versiones marcadas de estos, `te1` y `te2` respectivamente. Luego de construir el tipo final de la expresión, `t`, se devuelve éste, junto con la nueva aplicación con sus partes marcadas y con una nueva marca agregada `t2`.

De esta manera la aplicación de esta función a las distintas expresiones va "depositando" las marcas necesarias para *Arity Raising*, simultáneamente que va realizando la inferencia de tipos.

5.4 Lenguaje Destino

La representación del lenguaje destino es más sencilla pues en este ya no son necesarias tantas construcciones. Aquí se eliminan las marcas, las anotaciones de las tuplas y las proyecciones estáticas. Esta simplicidad no es azarosa sino que es, en parte, uno de los objetivos buscados en el trabajo.

Veamos en primer lugar el tipo de datos para los tipos de las expresiones en el lenguaje destino:

```
data ResType          =
     RTVar ResTypeVble
   | RTTuple     [ ResType ]
   | RTFun       ResType ResType
   | RTSusp      ResType
   | RTInt
```

De la misma manera que con el lenguaje anterior, aquí se agrega con respecto a la definición del capítulo 3, el tipo para las constantes numéricas. Al igual que con el lenguaje fuente el tipo `ResTypeVble` representa a los identificadores de las variables de tipo y no se incluye aquí pues no son de mayor importancia.

El tipo para los esquemas de tipos polimórficos del lenguaje destino es:

```
data ResTypeScheme      =
     RSForAll ResTypeVble ResTypeScheme
   | RSResType  ResType
```

que es muy similar al del lenguaje fuente.

Finalmente se diseñó el tipo de datos para las expresiones del lenguaje destino como sigue:

```
data ResTerm      =
     RVar     ResTermVble
   | RApp     ResTerm ResTerm
   | RLam     ResTermVble ResTerm
   | RLet     [(ResTermVble,ResTerm)] ResTerm
   | RTuple   [ ResTerm ]
   | RProj    Int Int ResTerm
   | RSusp    ResTerm
   | RForce   ResTerm
```

En la definición anterior las únicas construcciones que vale la pena aclarar son `RSusp` y `RForce`. Estas construcciones representan los términos $\$e$ y `force` e, respectivamente.

5.5 Algoritmo

En esta sección se explica en detalle como fue implementado el algoritmo de *Arity Raising*. Primero se explica la mónada `ARM` y sus primitivas, luego se comienza con la implementación del sistema de *Arity Raising* de tipos definido en 3.10 y finalmente se comenta la implementación del sistema de *Arity Raising* propiamente dicho definido en 3.18.

La mónada `ARM` implementada lleva en el estado los contextos de los sistemas de *Arity Raising* definidos. Veamos brevemente las primitivas de modificación de estado implementadas para la mónada `ARM`:

- `getInstances`: Devuelve las diferentes instanciaciones (ver cap. 3) para cada variable de tipo, que han sido "vistas" hasta el momento (Γ en la figura 3.3).
- `addInstances`: Agrega instanciaciones nuevas al estado (Γ).
- `forgetInstances`: Elimina todas las instanciaciones del estado.
- `addTuplicity`: Agrega una tuplicidad para una variable de tipo nueva (a Δ en la figura 3.3).
- `getTuplicityFor`: Dada una variable `v` devuelve su tuplicidad ($\Delta(v)$).

Comencemos viendo el sistema para *Arity Raising* de tipos. El sistema, si bien no es funcional pues para los esquemas de tipos σ podría dar varios resultados, se implementa restringido para los tipos τ. Esto no implica ninguna restricción adicional pues en la implementación este sistema no es utilizado sobre esquemas de tipos. Por lo tanto puede verse el sistema como una función que transforma un tipo del lenguaje fuente en una secuencia de tipos del lenguaje destino. Precisamente el tipo utilizado para esta función es:

```
tar :: SourceType -> ARM [ResidualType]
```

Veamos la implementación de algunas de las reglas. En primer lugar tenemos la regla (TAR-V). Dicha regla se implementa como sigue:

```
tar (STVar t) = do n <- getTuplicityFor t
                   return (map (\s->RTVar (t,s)) [1..n])
```

Como se puede observar, en primer lugar se obtiene la tuplicidad para la variable de tipo `t`, y luego se retorna una secuencia de variables dependiendo de su tuplicidad. Como se vé, esta implementación es directa a partir de la definición de regla.

Una definición un poco más complicada es la que se encarga de las funciones:

```
tar (STFun tau eta) =
     do ts <- tar tau
        es <- tar eta
        case ts of
          [] -> return (map RTSusp es)
          _  -> return (map (\x->foldr RTFun x ts) es)
```

Aquí se implementaron dos reglas juntas: la regla (TAR-FUN) y la regla (TAR-SUSP). El motivo es que estas dos reglas resuelven el caso del tipo de las funciones. Para ambas reglas, primero se computan los resultados para el argumento, `tau`, y para el resultado, `eta`. Luego, si el argumento no produjo ningún tipo para el lenguaje

residual estamos en el caso de la regla (TAR-SUSP), y entonces devuelven todos los resultados, pero suspendidos. En otro caso estamos en la regla (TAR-FUN), con la que se construyen las funciones currificadas y se genera un tipo de función por cada tipo de resultado obtenido.

Finalmente vemos las reglas de tipos que trabajan sobre las tuplas.

```
tar (STTuple Static xs) = do rs <- mapM tar xs
                             return (concat rs)
```

En este caso la implementación es directa a partir de la definición de la regla (TAR-STUPLE). Primero se obtienen simultáneamente los resultados para cada tipo de la tupla y luego se los concatena.

```
tar (STTuple Dynamic xs)= do rs <- mapM tar xs
                             return [RTTuple (concat rs)]
```

La implementación de la regla (TAR-DTUPLE) es similar a la anterior salvo que luego de concatenar los tipos resultantes, éstos son embebidos en una nueva tupla del lenguaje residual que es el único resultado.

La implementación del sistema de reglas de *Arity Raising* es también funcional. Para su codificación se utilizo una función con el tipo:

```
ar :: STerm -> SourceType ->  ARM [ResidualTerm]
```

es decir que a partir de un término en el lenguaje fuente y su tipo, se obtiene una secuencia de términos en el lenguaje residual. Veamos la implementación de las reglas más características del sistema:

En primera instancia veamos la regla (AR-VAR):

```
ar (Var i v) t = do n <- tp t
                    itp <- arrtp i
                    if n>0 then vseen v i else return ()
                    return (map RVar (varnames v itp n))
```

La implementación de esta regla es, en algún sentido, similar a la de variables de tipos. Para una variable `v` se obtiene la tuplicidad de su tipo `t`, utilizando la función monádica `tp`, la secuencia de tuplicidades de la instanciación `i`, utilizando `arrtp`, y se genera la secuencia de variables de la regla (AR-VAR) utilizando la función `varnames`. Un aspecto a notar es la llamada a la función `vseen v i`. Esta llamada registra que se ha visto la variable `v` y se agrega al contexto con la instanciación `i`, como se realiza en la regla original.

Un par de reglas sencillas para observar su implementación son las reglas (AR-STUPLE) y (AR-DTUPLE).

```
ar (Tuple Static es) (STTuple Static ts) =
                           do cs <- mapM (uncurry ar) (zip es ts)
                              return (concat cs)
```

```
ar (Tuple Dynamic es) (STTuple Dynamic ts) =
                            do cs <- mapM (uncurry ar) (zip es ts)
                               return [RTuple (concat cs)]
```

Estas reglas tratan a las expresiones de manera similar a como el sistema anterior trataba a sus tipos. La primera procesa las componentes de la tupla y luego genera una secuencia con estos resultados, mientras que la segunda realiza la misma operatoria salvo que los resultados son embebidos en *una* tupla resultante.

Una regla para observar el uso de las marcas de tipos es la de la aplicación. Las reglas (AR-AP) y (AR-APS) dividen el caso de la aplicación en dos, uno cuando el argumento genera código y otro cuando no. La implementación agrupa ambos casos:

```
ar (App t e f) eta = do n <- tp t
                        es <- ar e (STFun t eta)
                        if n>0 then
                          do fs <- ar f t
                             return (map (\x->foldl RApp x fs) es)
                         else
                          return (map force es)
```

Aquí primero se obtiene la tuplicidad de la marca de tipo para el argumento, lo que indica también la longitud de la secuencia generada por el argumento `f`. Luego, si hay código generado por `f`, se generan las aplicaciones correspondientes; una para cada expresión obtenida de la función `e`. En el caso que la secuencia obtenida sea vacía, se fuerza la ejecución de las expresiones obtenidas para la función generando una operación `force` para cada una. Como se ve en el código, se llama a la función `force` en lugar de incluir directamente el constructor `RForce`, esto es porque aquí es el único lugar donde se introduce un `force` y esa función es utilizada para reducir cada aparición de un `force` sobre el operador $.

Finalmente veamos la implementación de la regla del `let`, (AR-LET), que involucra la mayor parte de los aspectos del trabajo:

```
ar (Let decls e) eta  =
        do es <- ar e eta
           i <- getInstances
           forgetInstances
           ndecls <- ardecls decls i
           addInstances (filter (not . declared decls) i)
           return (map (RLet ndecls) es)
```

Más allá de las funciones auxiliares, la regla respeta el mismo esquema de las anteriores. En primer lugar se procesa el cuerpo del `let`, como se explicó en la

regla; luego se procesan las declaraciones del `let` utilizando las instanciaciones vistas hasta el momento (notar que en la formalización sólo se habla de una declaración por vez en el lenguaje fuente, pero aquí se tienen simultáneas), y luego se reconstruye un `let` con aquellas necesarias.

Para concluir veamos el código de la función que implementa el algoritmo de *Arity Raising*, utilizando la implementación de los sistemas de reglas anteriores:

```
arityRaise :: STerm -> SourceType ->
              ARM ([ResidualTerm],[ResidualType])
arityRaise e t = do ts <- tar t
                    es <- ar e t
                    return (es,ts)
```

La función `arityRaise` toma como argumento un término del lenguaje fuente junto con su tipo y devuelve una secuencia de los términos del lenguaje destino junto con sus tipos.

5.6 Ejemplos

En esta sección se muestran algunos ejemplos de código procesado con el algoritmo de *Arity Raising*, así como también se explica el funcionamiento del prototipo provisto junto con esta tesis, para su utilización.

Ejemplo 5.1. En este primer ejemplo se observa el trabajo que se realiza con las tuplas estáticas. Las funciones que reciben tuplas vacías (conocidas como *void* en especialización de tipos) son prácticamente eliminadas y las proyecciones de tuplas estáticas desaparecen junto con éstas. Al realizar *Arity Raising* sobre el siguiente código:

```
(\z->snd z) ((\x->(x,3)^S) ()^S )
```

se obtiene como resultado:

```
[(\x->x) 3] :: [Int]
```

Es importante notar que no hubo ningún tipo de ejecución, sino detección de partes irrelevantes del código. Vale aclarar esto pues este proceso se hace estáticamente observando sólo la estructura del código.

Ejemplo 5.2. En este ejemplo se muestra un caso sencillo de polimorfismo. La función polimórfica `f` recibe un par estático de dos tipos cualesquiera, pero al ser utilizada una única vez se utiliza esa única instanciación. Tenemos el código:

```
 let f= (\x -> snd^S x) in f (1,2)
```

que, al aplicar *Arity Raising* da como resultado una nueva función `f`, también polimórfica:

```
[let f = \x -> \y -> y in f 1 2] :: [Int]
```

Ejemplo 5.3. El siguiente ejemplo representa una situación clásica en especialización de programas. Un `let` con una única definición que es una tupla estática con varias funciones. Estas funciones son utilizadas a lo largo del código accediendo a las componentes de la tupla.

```
let f = (\x->2,\y->y,\z->8)^S
in (Proj_1_3 f 5, Proj_2_3 f 2, Proj_3_3 f (\j->j))
```

Como resultado de remover la tupla estática se obtienen las tres funciones por separado y se eliminan las proyecciones. Notar además que estas funciones, tanto las del lenguaje fuente como las del destino *son* polimórficas.

```
[let f = \x -> 2
   ; g = \x -> x
   ; h = \x -> 8
  in ( f 5 , g 2 , h ( \i -> i ) )]
 ::
[( Int , Int , Int )]
```

Ejemplo 5.4. Un ejemplo de polimorfismo en el que se utiliza una misma función polimórfica con tipos completamente diferentes es el siguiente:

```
let f = (\x->x) in (f 2, f (fst (3,4)^S), f f)
```

Aquí los tres tipos utilizados para instanciar la función poseen la característica de tener la misma *tuplicidad*, con lo que es posible utilizar la misma definición polimórfica.

```
[let f = (\x->x) in ( f 2 , f 3 , f f )]
 ::
[( Int , Int , ( Int -> Int ) )]
```

Ejemplo 5.5. Un último ejemplo de polimorfismo en el que los llamados a una función polimórfica poseen distinta *tuplicidad* se obtiene con un código de esta forma:

```
let f = (\x->3) in (f 2, f (2,3)^S)
```

En el primer llamado a la función `f` la tuplicidad del argumento es diferente a la del segundo llamado. El algoritmo de *Arity Raising* genera dos funciones, una para cada tuplicidad necesitada, y devuelve el siguiente código:

```
[let f = \x -> 3
   ; g = \x -> \y -> 3
  in ( f 2 , g 2 3)]
 ::
[( Int , Int )]
```

5.7 Utilización del prototipo

Para utilizar el prototipo es necesario cargar el modulo `Main.hs` provisto con esta tesis en algún interprete del lenguaje *Haskell*. Para ejecutar el algoritmo de *Arity Raising* sobre alguna expresión se deben realizar los siguientes pasos:

- En el directorio `examples` situado en el mismo lugar que el código fuente, crear un archivo llamado `ex`*name*`.ar`.
- Cargar el módulo `Main.hs` en el interprete de *Haskell*.
- Ejecutar la función `mainEx` pasando el identificador *name* entre comillas dobles como parámetro.

Con esta tesis se proveen 5 ejemplos en el directorio `examples`. Para procesar el ejemplo 2, la ejecución de la función se ve como:

```
Main> mainEx "2"
Main>
```

lo cual procesara el archivo llamado `ex2.ar` del directorio `examples`. Si el archivo posee la siguiente expresión:

```
let f = (\x -> snd^S x) in f (1,2)
```

el archivo será modificado generando un salto de página luego de la expresión y a continuación la salida del programa en forma de código comentado para permitir volver a ejecutar sobre el mismo archivo. El archivo final se vé entonces como (tener el cuenta que el texto entre "{-" y "-}" es un comentario en el lenguaje fuente para permitir utilizar el archivo de salida como entrada nuevamente):

```
let f = (\x -> snd^S x) in f ( 1 , 2 )

{-

The program in the source language:
let f = \x -> snd^S x in f ( 1 , 2 )^S
 ::
Int

-------------------------------------------

The program in the object language:
let f ::(Vb.Vc.( b , c )^S -> c)
 in f [ Int , Int ]  ( 1 , 2 )^S::( Int , Int )^S
 ::
```

```
Int

-------------------------------------------

The residual programs:
[let f = \x -> \y -> y in f 1 2]
 ::
[Int]

-}
```

Aquí se puede observar en primer lugar, bajo el título de *The program in the source language*, el resultado del *parsing* y de la inferencia de tipo global. En el ejemplo es prácticamente la misma función para la que se ha inferido que la tupla pasada a la función f debe ser estática y que el tipo de toda la expresión es un entero.

En segundo lugar, bajo el título de *The program in the object language*, se observa el resultado de colocar todas las marcas de tipos en la expresión. Se puede observar el tipo polimórfico inferido para la función definida en el `let` y además la instanciación aplicada para esta.

Finalmente se muestra en *The residual programs* la secuencia de expresiones obtenidas al procesar la expresión, junto con los tipos de todas ellas. En el ejemplo se obtiene una única expresión de tipo entero.

5.8 Conclusiones

Se implementó un prototipo del algoritmo de *Arity Raising* que sirve como base para ser incorporado a un especializador completo como el desarrollado por [Martínez López and Hughes, 2002]. Para que el procesamiento del algoritmo sea más eficiente sería deseable que la implementación se encuentre embebida *dentro* del especializador y su ejecución sea realizada, cuando es posible, durante la especialización. Una de las ventajas de haber elegido un lenguaje funcional con *laziness* es que la realización de esta tarea es más sencilla y queda, en gran parte, delegada al compilador.

Capítulo 6

Trabajo Futuro

No al heroísmo laboral

Alejandro Donnantuoni

El trabajo descripto hasta aquí tiene muchas maneras de continuarse. Veremos dos líneas de trabajo que deberían ser las primeras en atacarse. En la primer sección se discute acerca de la recursión, aspecto fundamental a incorporar en el lenguaje tratado por *Arity Raising*; luego se delinean los puntos a tener en cuenta al enfocar el trabajo desde un punto de vista modular, y finalmente se explica el camino a seguir en la parte de implementación.

6.1 Recursión

El lenguaje descripto en el capítulo 2 es muy expresivo, pero no todo lo expresivo que se desearía. El ingrediente esencial que no posee es la recursión. Si bien la incorporación de un operador de recursión en el lenguaje debe ser estudiada con profundidad, ésta no trae demasiadas dificultades. En primera instancia es necesario agregar un operador de recursión al lenguaje fuente, modificando la definición 2.2, agregando un operador como

$$\texttt{fix}\ e$$

Además es necesario incluir una regla de inferencia de tipo y transformación al lenguaje objeto, la cual es la regla de tipado tradicional de este operador pues no necesita ninguna marca de tipo especial:

$$\text{(MG-FIX)}\quad \frac{A \vdash e \rightsquigarrow e' : \tau \to \tau}{A \vdash \texttt{fix}\ e \rightsquigarrow \texttt{fix}\ e' : \tau}$$

El siguiente paso es definir una regla para *Arity Raising* que contemple el caso recursivo. Antes de llegar a ese punto, es necesario extender el operador `fix` para el nuevo contexto impuesto por el lenguaje destino.

Definición 6.1. Sean i,n números naturales con $i < n$; se definen, para cada i y para cada n, los operadores:

$$\texttt{fix}^n_i\ e_1 ... e_n = e_i\ (\texttt{fix}^n_1\ e_1 ... e_n)\ (\texttt{fix}^n_2\ e_1 ... e_n) ...\ (\texttt{fix}^n_n\ e_1 ... e_n)$$

Utilizando los nuevos operadores de punto fijo, se puede definir una regla de *Arity Raising* como la siguiente:

$$\text{(AR-FIX)} \quad \frac{\Delta \mid \Gamma \vdash e : \tau \to \tau \rightarrowtail [e'_i]_{i:1..n}}{\Delta \mid \Gamma \vdash \texttt{fix}\ e : \tau \rightarrowtail [\texttt{fix}_i^n\ e'_1 ... e'_n]_{i:1..n}}$$

A partir de aquí el trabajo consiste en estudiar si la inclusión de esta regla afecta, o no, a la corrección del algoritmo.

6.2 Modularidad

Otra tarea a tener en cuenta para fortalecer el algoritmo de *Arity Raising*, es la de *modularidad*. Si bien éste es un terreno menos explorado que el anterior, es importante hacer notar las dificultades que deben afrontarse al atacar este problema.

En primer lugar, intentar realizar *Arity Raising* modular de funciones polimórficas trae un inconveniente fundamental si se realiza utilizando la técnica de monomorfización descripta en este trabajo. Una función polimórfica tiene varios resultados diferentes de acuerdo a las instanciaciones con las que se la utilice. Al existir un entorno modular no es posible tener a la mano todas las instanciaciones con las que se utilizará una función y esta tarea se complica.

Una alternativa para este tema es explorar la utilización de *qualified types* [Jones, 1994], para imponer restricciones sobre los tipos de las funciones utilizadas desde otros módulos y agregar una fase de post-procesamiento para finalizar *Arity Raising* luego de poseer la completa información de los módulos participantes.

6.3 Extensión de *Arity Raising* para Type Specialization

El siguiente paso obligado de este trabajo es continuar con más primitivas del lenguaje fuente para poder hacer de este algoritmo la fase real de post-procesamiento de un especializador. Hasta el momento se sentaron las bases más fuertes para este trabajo y sólo restan la incorporación de nuevas construcciones al lenguaje, realizando la implementación de éstas y uniéndolas con algún especializador como el implementado por [Martínez López and Hughes, 2002].

Capítulo 7

Conclusiones

Pero eso no parecía importar. Había nuevas cosas que aprender, sin duda.

Tehanu.
Ursula K. Le Guin.

El trabajo de esta tesis se concentró en cinco puntos principales:

- En primer lugar se realizó una formalización completa del proceso de *Arity Raising*;
- se incluyó en el trabajo el concepto real de alto orden;
- se realizó un sistema capaz de procesar código polimórfico;
- se incorporó el operador de suspensión para lograr corrección, y finalmente
- se demostró esta corrección dando una semántica a los lenguajes involucrados.

A continuación se resume brevemente lo conseguido en cada uno de estos puntos.

Los algoritmos de *Arity Raising* existentes aplicados en *Evaluación Parcial* o en *Especialización de programas*, al ser una fase posterior al trabajo principal, han sido dejados de lado, sin ser formalizados ni exploradas sus potencialidades. En este trabajo se da una formalización de un pequeño sistema de *Arity Raising* con el objetivo de proveer una base sobre la cual formalizar nuevas primitivas adecuadas para cada trabajo particular.

El concepto de alto orden, visto en esta tesis como la posibilidad de manejar funciones como elementos de primera clase, es aplicado ortogonalmente en la formalización. Las formalizaciones existentes hacen un uso incompleto del alto orden, permitiendo su utilización pero no usando el poder de éste para la generación del nuevo código. En este trabajo se utiliza alto orden tanto en los lenguajes objeto y destino, como en el código introducido automáticamente.

Una nueva característica no explorada hasta el momento fue la de poseer un algoritmo de *Arity Raising* que trabaje y genere funciones polimórficas. En esta tesis se desarrolló un sistema capaz de manipular este tipo de funciones. Para poder realizar este trabajo fue necesario definir un nuevo conjunto de nociones no

existentes hasta ahora, haciendo que esta tesis aporte nuevos conceptos e ideas para este tipo de tareas.

En la búsqueda de la corrección se introdujo un nuevo operador en el lenguaje destino que brinda la posibilidad de tener suspensión en la evaluación de algunas expresiones. Este nuevo operador es utilizado para compensar la eliminación de algunas estructuras del lenguaje, que de ser eliminadas sin más, modificarían radicalmente la semántica del programa generado.

Finalmente se definió la semántica de los lenguajes origen y destino y se demostró que un programa luego de ser convertido por el algoritmo de *Arity Raising* es, en el sentido propuesto, equivalente con el original, concluyendo así que el algoritmo descripto en esta tesis hace lo que tiene que hacer.

En resumen: se creó el *primer* algoritmo de *Arity Raising* polimórfico con alto orden real, mostrando el camino inmediato para agregar recursión y dando una demostración de corrección que captura incluso la noción de no terminación.

Apéndice A

Demostraciones

Let no one enter who does not know mathematics.

Inscripción en la puerta de Platón,
probablemente en la Academia de Atenas

En este apéndice se presentan todas las demostraciones *completas* de los lemas y teoremas enunciados en este trabajo. Se ha decidido apartarlas para proveer al lector una lectura más fluida de la primera parte y luego poder concentrase en las partes técnicas de las demostraciones.

Lema 3.12. Sea η tal que $\Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..m}$. Si $(t, m) \in \Delta$ y $\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n}$, entonces $\Delta \vdash \tau[t/\eta] \xrightarrow{tar} [\tau'_i[\vec{t}/\vec{\eta}]]_{i:1..n}$.

Demostración. Lo demostraremos por inducción sobre la estructura de la derivación de $\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..m}$. Notar que no es necesario tener en cuenta un caso donde se aplique la regla (TAR-SCHM), ya que el lema no tiene en cuenta los esquemas de tipo σ.

Caso (TAR-T) Tenemos una derivación de la forma

$$\Delta \vdash \iota \xrightarrow{tar} [\iota]$$

Como $\iota[t/\eta] = \iota$ se cumple trivialmente que

$$\Delta \vdash \iota[t/\eta] \xrightarrow{tar} [\iota[\vec{t}/\vec{\eta}]]$$

Caso (TAR-V) Tenemos una derivación de la forma

$$\frac{(t', n) \in \Delta}{\Delta \vdash t' \xrightarrow{tar} [t'_i]_{i:1..n}}$$

Aplicando las sustituciones tenemos que

$$\Delta \vdash t'[t/\eta] \xrightarrow{tar} [t'_i[\vec{t}/\vec{\eta}]]_{i:1..n}$$

Tenemos dos casos:

- $t = t'$) En este caso como $m = n$ y por definición de sustitución obtenemos que
$$\Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..m}$$
Lo cual es cierto por hipótesis.
- $t \neq t'$) En este caso la aplicación de la sustitución no modifica los tipos en ninguno de los lados de la relación. Por lo tanto, luego de aplicar la sustitución tenemos que
$$\Delta \vdash t' \xrightarrow{tar} [t'_i]_{i:1..n}$$

Caso (TAR-FUN) Tenemos una derivación de la forma

$$\frac{\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n>0} \quad \Delta \vdash \eta' \xrightarrow{tar} [\eta''_i]_{i:1..m}}{\Delta \vdash \tau \to \eta' \xrightarrow{tar} [\tau'_1 \to ... \to \tau'_n \to \eta''_i]_{i:1..m}}$$

Aplicando hipótesis inductiva sobre ambas premisas obtenemos que

$$\Delta \vdash \tau[t/\eta] \xrightarrow{tar} [\tau'_i[\vec{t}/\vec{\eta}]]_{i:1..n>0}$$
$$\Delta \vdash \eta'[t/\eta] \xrightarrow{tar} [\eta''_i[\vec{t}/\vec{\eta}]]_{i:1..m}$$

Aplicando (TAR-FUN) sobre estos resultados tenemos

$$\Delta \vdash \tau[t/\eta] \to \eta'[t/\eta] \xrightarrow{tar} [\tau'_1[\vec{t}/\vec{\eta}] \to ... \to \tau'_n[\vec{t}/\vec{\eta}] \to \eta''_i[\vec{t}/\vec{\eta}]]_{i:1..m}$$

Finalmente, por definición de sustitución

$$\Delta \vdash (\tau \to \eta')[t/\eta] \xrightarrow{tar} [(\tau'_1 \to ... \to \tau'_n \to \eta''_i)[\vec{t}/\vec{\eta}]]_{i:1..m}$$

Caso (TAR-SUSP) Tenemos una derivación de la forma

$$\frac{\Delta \vdash \tau \xrightarrow{tar} [\,] \quad \Delta \vdash \eta' \xrightarrow{tar} [\eta''_i]_{i:1..n}}{\Delta \vdash \tau \to \eta' \xrightarrow{tar} [\texttt{susp}\ \eta''_i]_{i:1..n}}$$

Aplicando hipótesis inductiva sobre ambas premisas obtenemos que

$$\Delta \vdash \tau[t/\eta] \xrightarrow{tar} [\,]$$
$$\Delta \vdash \eta'[t/\eta] \xrightarrow{tar} [\eta''_i[\vec{t}/\vec{\eta}]]_{i:1..n}$$

Aplicando (TAR-SUSP) sobre estos resultados, vale que

$$\Delta \vdash \tau[t/\eta] \to \eta'[t/\eta] \xrightarrow{tar} [\texttt{susp}\ (\eta''_i[\vec{t}/\vec{\eta}])]_{i:1..n}$$

Finalmente, por definición de sustitución

$$\Delta \vdash (\tau \to \eta')[t/\eta] \xrightarrow{tar} [(\texttt{susp}\ \eta''_i)[\vec{t}/\vec{\eta}]]_{i:1..n}$$

Caso (TAR-DTUPLE) Tenemos una derivación de la forma

$$\frac{\Delta \vdash \tau.i \xrightarrow{tar} [\tau.i'_j]_{j:1..m_i}}{\Delta \vdash (\tau)^D_n \xrightarrow{tar} [(\tau.1'_1, ..., \tau.1'_{m_1}, ..., \tau.n'_{m_n})_{m_1+\cdots+m_n}]}$$

Aplicando hipótesis inductiva sobre todas la premisas obtenemos que, para $1 \leq i \leq n$

$$\Delta \vdash \tau.i[t/\eta] \xrightarrow{tar} [\tau.i'_j[\vec{t}/\vec{\eta}]]_{j:1..m_i}$$

Utilizando (TAR-DTUPLE) sobre estos resultados tenemos que

$$\Delta \vdash (\tau[t/\eta])^D_n \xrightarrow{tar} [(\tau.1'_1[\vec{t}/\vec{\eta}], ..., \tau.1'_{m_1}[\vec{t}/\vec{\eta}], ..., \tau.n'_{m_n}[\vec{t}/\vec{\eta}])_{m_1+\cdots+m_n}]$$

Finalmente, por definición de sustitución

$$\Delta \vdash (\tau)^D_n[t/\eta] \xrightarrow{tar} [(\tau.1'_1, ..., \tau.1'_{m_1}, ..., \tau.n'_{m_n})_{m_1+\cdots+m_n}[\vec{t}/\vec{\eta}]]$$

Caso (TAR-STUPLE) Tenemos una derivación de la forma

$$\frac{\Delta \vdash \tau.i \xrightarrow{tar} [\tau.i'_j]_{j:1..m_i}}{\Delta \vdash (\tau)^S_n \xrightarrow{tar} [\tau.1'_j]_j \parallel ... \parallel [\tau.n'_j]_j}$$

Aplicando hipótesis inductiva sobre todas la premisas obtenemos que, para $1 \leq i \leq n$

$$\Delta \vdash \tau.i[t/\eta] \xrightarrow{tar} [\tau.i'_j[\vec{t}/\vec{\eta}]]_{j:1..m_i}$$

Utilizando (TAR-STUPLE) sobre estos resultados tenemos que

$$\Delta \vdash (\tau[t/\eta])^S_n \xrightarrow{tar} [\tau.1'_j[\vec{t}/\vec{\eta}]]_j \parallel ... \parallel [\tau.n'_j[\vec{t}/\vec{\eta}]]_j$$

Finalmente, por definición de sustitución

$$\Delta \vdash (\tau)^S_n[t/\eta] \xrightarrow{tar} [\tau.1'_j[\vec{t}/\vec{\eta}]]_j \parallel ... \parallel [\tau.n'_j[\vec{t}/\vec{\eta}]]_j$$

□

Propiedad 3.14. Sea Δ tal que $s, t \notin \Delta$, entonces vale que

$$\Delta; (s, \tau); (t, \eta) = \Delta; (t, \eta[s/\tau]); (s, \tau)$$

Demostración. Trivial utilizando el 3.12 □

Corolario 4.2. Sea $\tau \leq_{\vec{\eta}} \sigma$ entonces si $\Delta \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..n}$ es compatible con $\vec{\eta}$, $\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n}$ y $\tau'_i \leq \sigma'_i$

Demostración. Sea $\sigma = \forall \vec{t}.\tau'$. Luego, por hipótesis, $\tau'[\vec{t}/\vec{\eta}] = \tau$. Como $\Delta \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..n}$ es compatible con $\vec{\eta}$, vale que

$$\Delta; (\vec{t}, \vec{\eta}) \vdash \tau' \xrightarrow{tar} [\tau''_i]_{i:1..n} \tag{A.1}$$

$$\sigma'_i = \forall \vec{t'}.\tau''_i \tag{A.2}$$

Aplicando el lema anterior sobre (A.1), tenemos que existe la sustitución S tal que

$$\Delta; (\vec{t}, \vec{\eta}) \vdash \tau'[\vec{t}/\vec{\eta}] \xrightarrow{tar} [S\tau''_i]_{i:1..n}$$

Como $\vec{t}$ no aparecen libres en $\tau'[\vec{t}/\vec{\eta}]$ y $\tau'[\vec{t}/\vec{\eta}] = \tau$

$$\Delta \vdash \tau \xrightarrow{tar} [S\tau''_i]_{i:1..n}$$

Finalmente, por definición

$$S\tau''_i \leq \sigma'_i$$

□

Propiedad 4.4. Sea A una asignación de tipos, Γ,Γ' colecciones de instanciaciones y Δ una asignación de tuplicidades. Luego, se cumple que $A \times \Gamma_\Delta \subseteq A \times (\Gamma, \Gamma')_\Delta$ y este último está bien formado.

Demostración. Demostraremos primero la inclusión y luego veremos que agregar una nueva instanciación no genera asignaciones de tipos diferentes para una misma variable. Sea $v_{\rho(\Delta,\vec{\eta}),i} : \sigma'_i \in A \times \Gamma_\Delta$, entonces vale que

$$v : \sigma \in A \tag{A.3}$$

$$(v, \vec{\eta}) \in \Gamma \tag{A.4}$$

$$\Delta \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..n} \text{ (compatible con } \vec{\eta}) \tag{A.5}$$

Utilizando (A.4), vale que

$$(v, \vec{\eta}) \in \Gamma, \Gamma' \tag{A.6}$$

Utilizando (A.5), (A.3) y (A.6) vale entonces

$$v_{\rho(\Delta,\vec{\eta}),i} : \sigma'_i \in A \times \Gamma, \Gamma'_\Delta$$

Veamos ahora que agregar una nueva instanciacion no genera una asignación diferente a una misma variable. Supongamos que $v_{\rho(\Delta,\vec{\eta}),i} : \sigma'_i \in A \times \Gamma_\Delta$, entonces

$$v : \sigma \in A \tag{A.7}$$

$$(v, \vec{\eta}) \in \Gamma \tag{A.8}$$

$$\Delta \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..n} \text{ (compatible con } \vec{\eta}) \tag{A.9}$$

Sea $(v, \vec{\eta}')$ la nueva instanciación. Luego para agregar una nueva asignación en $A \times \Gamma_\Delta$ se tiene que cumplir que

$$(v, \vec{\eta}') \in \Gamma \tag{A.10}$$

$$\Delta \vdash \sigma \xrightarrow{tar} [\sigma_i'']_{i:1..m} \text{ (compatible con } \vec{\eta}') \tag{A.11}$$

Estas dos condiciones junto con (A.7) implican que

$$v_{\rho(\Delta,\vec{\eta}'),i} : \sigma_i'' \in A \times \Gamma_\Delta$$

Luego, si $\rho(\Delta, \vec{\eta}') \neq \rho(\Delta, \vec{\eta})$, estamos hablando de variables diferentes y no hay inconsistencia. Tomaremos el caso en que $\rho(\Delta, \vec{\eta}') = \rho(\Delta, \vec{\eta})$, en el cual a la variable $v_{\rho(\Delta,\vec{\eta}'),i}$ se le está asignando tanto σ_i' como σ_i''.
Supongamos $\sigma = \forall \vec{t}\,.\tau$, utilizando (A.9), y la definición de compatibilidad obtenemos que

$$\Delta; (\vec{t}\,, \vec{\eta}) \vdash \tau \xrightarrow{tar} [\tau_i']_{i:1..n}$$

donde $\sigma_i' = \forall \vec{t'}.\tau_i'$. Luego, como $\rho(\Delta, \vec{\eta}') = \rho(\Delta, \vec{\eta})$ es cierto que

$$\Delta; (\vec{t}\,, \vec{\eta}') \vdash \tau \xrightarrow{tar} [\tau_i']_{i:1..n}$$

Por lo tanto

$$\Delta \vdash \sigma \xrightarrow{tar} [\sigma_i']_{i:1..n} \text{ es compatible con } \vec{\eta}'$$

Concluyendo finalmente que $\sigma_i' = \sigma_i''$. □

Propiedad 4.5. Sea τ tal que $\rho(\Delta, \tau) = n$. Si $t \notin TV(A)$ y $t \notin \Delta$, entonces $A \times \Gamma_{\Delta,(t,n)} = A \times \Gamma[t/\tau]_\Delta$

Demostración. Tenemos que $v_{\rho((\Delta,(t,n)),\vec{\eta}),i} : \sigma_i' \in A \times \Gamma_{\Delta,(t,n)}$ si y sólo si se cumplen las siguientes condiciones

$$v : \sigma \in A \tag{A.12}$$

$$(v, \vec{\eta}) \in \Gamma \tag{A.13}$$

$$\Delta, (t, n) \vdash \sigma \xrightarrow{tar} [\sigma_i']_{i:1..n} \text{ (compatible con } \vec{\eta}) \tag{A.14}$$

Como $t \notin TV(\sigma)$ entonces (A.14) se cumple si y sólo si

$$\Delta \vdash \sigma \xrightarrow{tar} [\sigma_i']_{i:1..n} \text{ (compatible con } \vec{\eta}) \tag{A.15}$$

Por otra parte (A.13) es cierto si y sólo si

$$(v, \vec{\eta}[t/\tau]) \in \Gamma[t/\tau] \tag{A.16}$$

Luego, por definición de $\times$, tenemos que (A.12), (A.15) y (A.16) son ciertas si y sólo si

$$v_{\rho(\Delta,\vec{\eta}\,[t/\tau]),i} : \sigma'_i \in A \times \Gamma[t/\tau]_\Delta \tag{A.17}$$

Finalmente, utilizando el lema 3.12, podemos afirmar que

$$\rho(\Delta, \vec{\eta}[t/\tau]) = \rho(\Delta, (t,n), \vec{\eta})$$

Concluyendo con que (A.17) es cierto si y sólo si

$$v_{\rho(\Delta,(t,n),\vec{\eta}),i} : \sigma'_i \in A \times \Gamma[t/\tau]_\Delta$$

□

Corolario 4.6. Si $t \notin TV(A)$ y $t \notin TV(\Gamma)$ entonces $(\forall n)(A \times \Gamma_\Delta = A \times \Gamma_{\Delta,(t,n)})$

Demostración. Trivial, utilizando la definición de sustitución □

Teorema 4.7. Sea A tal que $A \vdash e \rightsquigarrow e' : \tau$, entonces $(\forall\Gamma)(\forall\Delta)(\Delta \,|\, \Gamma \vdash e' : \tau \rightarrowtail [e''_i]_{i:1..n} \Rightarrow \Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n} \wedge (\forall i : 1..n)(A \times \Gamma_\Delta \vdash e''_i : \tau'_i))$.

Demostración. Lo demostraremos por inducción sobre el árbol de derivación de $\Delta \,|\, \Gamma \vdash e' : \tau \rightarrowtail [e''_i]_{i:1..n}$

Caso (AR-VAR) Tenemos una derivación de la forma:

$$\Delta \,|\, (v, \vec{\eta}) \vdash v\ \vec{\eta} : \tau \rightarrowtail [v_{\rho(\Delta,\vec{\eta}),i}]_{i:1..\rho(\Delta,\tau)}$$

Además, como $e' = v\ \vec{\eta}$, existe σ tal que tenemos una derivación:

$$\frac{(v : \sigma) \in A \quad \tau \leq_{\vec{\eta}} \sigma}{A \vdash v \rightsquigarrow v\ \vec{\eta} : \tau} \tag{A.18}$$

Sean σ'_i tales que

$$\Delta \vdash \sigma \xrightarrow{tar} [\sigma'_i]_{i:1..n} \tag{A.19}$$

es compatible con $\vec{\eta}$. Luego, por el corolario 4.2, tenemos que

$$\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n} \tag{A.20}$$

$$\tau'_i \leq \sigma'_i \tag{A.21}$$

Por otra parte, por definición 4.3 con las premisas de (A.18) y (A.19) tenemos que para $1 \leq i \leq n$,

$$v_{\rho(\Delta,\vec{\eta}),i} : \sigma'_i \in A \times (v, \vec{\eta})_\Delta \tag{A.22}$$

Aplicando (HMD-VAR) sobre este resultado tenemos que

$$A \times (v, \vec{\eta})_\Delta \vdash v_{\rho(\Delta,\vec{\eta}),i} : \sigma'_i$$

Dado (A.21), sabemos que existe una sustitución S tal que $S\sigma'_i = \tau'_i$, por lo tanto, aplicando (HMD-INST) obtenemos

$$A \times (v, \vec{\eta})_\Delta \vdash v_{\rho(\Delta,\vec{\eta}),i} : \tau'_i$$

Finalmente, por definición de ρ con (A.20), podemos afirmar también que $n = \rho(\Delta, \tau)$

Caso (AR-AP) Tenemos una derivación de la forma:

$$\frac{\Delta \,|\, \Gamma \vdash e : \tau \to \eta \rightarrowtail [e'_i]_{i:1..n} \quad \Delta \,|\, \Gamma' \vdash f : \tau \rightarrowtail [f'_j]_{j:1..m>0}}{\Delta \,|\, \Gamma, \Gamma' \vdash e\ f_\tau : \eta \rightarrowtail [e'_i\ f'_1 ... f'_m]_{i:1..n}} \tag{A.23}$$

Por otra parte, tenemos una derivación como la siguiente:

$$\frac{A \vdash e' \rightsquigarrow e : \tau \to \eta \quad A \vdash f' \rightsquigarrow f : \tau}{A \vdash e'\ f' \rightsquigarrow e\ f_\tau : \eta}$$

Aplicando la hipótesis inductiva sobre ambas premisas de (A.23) obtenemos que

$$\Delta \vdash \tau \to \eta \xrightarrow{tar} [\tau''_i]_{i:1..n} \tag{A.24}$$
$$A \times \Gamma_\Delta \vdash e'_i : \tau''_i \tag{A.25}$$
$$\Delta \vdash \tau \xrightarrow{tar} [\tau'_j]_{j:1..m>0} \tag{A.26}$$
$$A \times \Gamma'_\Delta \vdash f'_j : \tau'_j \tag{A.27}$$

Pero, para probar (A.24), como (A.26) es cierto, tenemos una derivación de la forma

$$\frac{\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..m>0} \quad \Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..n}}{\Delta \vdash \tau \to \eta \xrightarrow{tar} [\tau'_1 \to ... \to \tau'_m \to \eta'_i]_{i:1..n}}$$

Por lo tanto $\tau''_i = \tau'_1 \to ... \to \tau'_m \to \eta'_i$. Luego, por propiedad 4.4 $A \times \Gamma_\Delta \subseteq A \times (\Gamma, \Gamma')_\Delta$. Utilizando esto en (A.25) y (A.27) tenemos que

$$A \times (\Gamma, \Gamma')_\Delta \vdash e'_i : \tau'_1 \to ... \to \tau'_m \to \eta'_i$$
$$A \times (\Gamma, \Gamma')_\Delta \vdash f'_j : \tau'_j$$

Finalmente, aplicando m veces la regla (HMD-APP) concluimos que, para $1 \leq i \leq n$,

$$A \times (\Gamma, \Gamma')_\Delta \vdash e'_i\ f'_1 ... f'_m : \eta'_i$$

Caso (AR-APS) Tenemos una derivación de la forma:

$$\frac{\Delta \,|\, \Gamma \vdash e : \tau \to \eta \rightarrowtail [e'_i]_{i:1..n} \quad \Delta \,|\, \Gamma' \vdash f : \tau \rightarrowtail []}{\Delta \,|\, \Gamma \vdash e\ f_\tau : \eta \rightarrowtail [\texttt{force}\ e'_i]_{i:1..n}} \tag{A.28}$$

Por otra parte, tenemos una derivación como la siguiente:

$$\frac{A \vdash e' \rightsquigarrow e : \tau \to \eta \quad A \vdash f' \rightsquigarrow f : \tau}{A \vdash e'\, f' \rightsquigarrow e\, f_\tau : \eta}$$

Aplicando la hipótesis inductiva sobre ambas premisas de (A.28) obtenemos que

$$\Delta \vdash \tau \to \eta \xrightarrow{tar} [\tau''_i]_{i:1..n} \tag{A.29}$$
$$A \times \Gamma_\Delta \vdash e'_i : \tau''_i \tag{A.30}$$
$$\Delta \vdash \tau \xrightarrow{tar} [] \tag{A.31}$$

Pero, para probar (A.29), como (A.31) es cierto, tenemos una derivación de la forma

$$\frac{\Delta \vdash \tau \xrightarrow{tar} [\,] \quad \Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..n}}{\Delta \vdash \tau \to \eta \xrightarrow{tar} [\texttt{susp}\ \eta'_i]_{i:1..n}}$$

Por lo tanto $\tau''_i = \texttt{susp}\ \eta'_i$. Aplicando la regla (TAR-SUSP) sobre A.30, concluimos

$$A \times \Gamma_\Delta \vdash \texttt{force}\ e'_i : \eta'_i$$

Caso (AR-SPROJ) Tenemos una derivación de la forma:

$$\frac{\Delta \mid \Gamma \vdash e : (\tau)^S_n \rightarrowtail [e'_i]_{i:1..m} \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \mid \Gamma \vdash \pi^S_{p,n}\ e_{(\tau)^S_n} : \tau.p \rightarrowtail [e'_{k+j}]_{j:1..\rho(\Delta,\tau.p)}} \tag{A.32}$$

Por otra parte, tenemos una derivación como la siguiente:

$$\frac{A \vdash e' \rightsquigarrow e : (\tau)^S_n}{A \vdash \pi^S_{p,n} e' \rightsquigarrow \pi^S_{p,n} e_{(\tau)^S_n} : \tau.p}$$

Aplicando la hipótesis inductiva sobre la primer premisa de (A.32), obtenemos

$$\Delta \vdash (\tau)^S_n \xrightarrow{tar} [\tau'_i]_{i:1..m} \tag{A.33}$$
$$A \times \Gamma_\Delta \vdash e'_i : \tau'_i \tag{A.34}$$

Para probar (A.33) tenemos una derivación de la forma

$$\frac{\Delta \vdash \tau.i \xrightarrow{tar} [\tau.i'_j]_{j:1..m_i}}{\Delta \vdash (\tau)^S_n \xrightarrow{tar} [\tau.1'_j]_{j:1..m_1} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} [\tau.n'_j]_{j:1..m_n}}$$

Luego, como $k = m_1 + \cdots + m_{p-1}$, tenemos que para $1 \leq j \leq m_p$, $\tau'_{k+j} = \tau.p'_j$. Aplicando este resultado en (A.34), obtenemos

$$A \times \Gamma_\Delta \vdash e'_{k+j} : \tau.p'_j$$

Finalmente sólo resta notar que, por definición, $\rho(\Delta, \tau.p) = m_p$.

Caso (AR-DPROJ) Tenemos una derivación de la forma:

$$\frac{\Delta \,|\, \Gamma \vdash e : (\tau)^D_n \rightarrowtail [e'] \quad m = \sum_s \rho(\Delta, \tau.s) \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \,|\, \Gamma \vdash \pi^D_{p,n}\ e_{(\tau)^D_n} : \tau.p \rightarrowtail [\pi_{k+j,m} e']_{j:1..\rho(\Delta,\tau.p)}} \tag{A.35}$$

Por otra parte, tenemos una derivación como la siguiente:

$$\frac{A \vdash e'' \rightsquigarrow e : (\tau)^D_n}{A \vdash \pi^D_{p,n} e'' \rightsquigarrow \pi^D_{p,n} e_{(\tau)^D_n} : \tau.p}$$

Aplicando la hipótesis inductiva sobre la primer premisa de (A.35), obtenemos

$$\Delta \vdash (\tau)^D_n \xrightarrow{tar} [\tau'] \tag{A.36}$$
$$A \times \Gamma_\Delta \vdash e' : \tau' \tag{A.37}$$

Para probar (A.36) tenemos una derivación de la forma

$$\frac{\Delta \vdash \tau.i \xrightarrow{tar} [\tau.i'_j]_{j:1..m_i}}{\Delta \vdash (\tau)^D_n \xrightarrow{tar} [(\tau.1'_1, ..., \tau.1'_{m_1}, ..., \tau.n'_{m_n})_{m_1+\cdots+m_n}]}$$

Luego, como $k = m_1 + \cdots + m_{p-1}$, la $(k+j)$-ésima componente de τ' es $\tau.p'_j$. Luego, al aplicar la regla (HMD-PROJ) sobre (A.37) tenemos que para $1 \leq j \leq m_p$,

$$A \times \Gamma_\Delta \vdash \pi_{k+j,m} e' : \tau.p'_j$$

Finalmente sólo resta notar que, por definición, $\rho(\Delta, \tau.p) = m_p$.

Caso (AR-STUPLE) Tenemos una derivación de la forma:

$$\frac{\Delta \,|\, \Gamma \vdash e.i : \tau.i \rightarrowtail [e.i'_j]_{j:1..m_i}}{\Delta \,|\, \Gamma \vdash (e)^S_n : (\tau)^S_n \rightarrowtail [e.1'_j]_{j:1..m_1} \mathbin{\#\!\!\#} ... \mathbin{\#\!\!\#} [e.n'_j]_{j:1..m_n}} \tag{A.38}$$

Por otra parte tenemos una derivación de la forma

$$\frac{A \vdash e.i'' \rightsquigarrow e.i : \tau.i}{A \vdash (e'')^S_n \rightsquigarrow (e)^S_n : (\tau)^S_n}$$

Aplicando la hipótesis inductiva sobre la primer premisa de (A.38), obtenemos

$$\Delta \vdash \tau.i \xrightarrow{tar} [\tau.i'_j]_{j:1..m_i} \tag{A.39}$$
$$A \times \Gamma_\Delta \vdash e.i'_j : \tau.i'_j \tag{A.40}$$

Aplicando la regla (TAR-STUPLE) sobre (A.39), obtenemos

$$\Delta \vdash (\tau)^S_n \xrightarrow{tar} [\tau.1'_j]_{j:1..m_1} \mathbin{\#\!\!\#} ... \mathbin{\#\!\!\#} [\tau.n'_j]_{j:1..m_n}$$

que, junto con la ecuación (A.40), prueban lo que queremos demostrar.

Caso (AR-DTUPLE) Tenemos una derivación de la forma:

$$\frac{\Delta \,|\, \Gamma \vdash e.i : \tau.i \rightarrowtail [e.i'_j]_{j:1..m_i}}{\Delta \,|\, \Gamma \vdash (e)^D_n : (\tau)^D_n \rightarrowtail [(e.1'_1, ..., e.1'_{m_1}, ..., e.n'_{m_n})_{m_1+\cdots+m_n}]} \tag{A.41}$$

Por otra parte tenemos una derivación de la forma

$$\frac{A \vdash e.i'' \rightsquigarrow e.i : \tau.i}{A \vdash (e'')^D_n \rightsquigarrow (e)^D_n : (\tau)^D_n}$$

Aplicando la hipótesis inductiva sobre la primer premisa de (A.41), obtenemos

$$\Delta \vdash \tau.i \xrightarrow{tar} [\tau.i'_j]_{j:1..m_i} \tag{A.42}$$
$$A \times \Gamma_\Delta \vdash e.i'_j : \tau.i'_j \tag{A.43}$$

Aplicando la regla (TAR-DTUPLE) sobre (A.42), obtenemos

$$\Delta \vdash (\tau)^D_n \xrightarrow{tar} [(\tau.1'_1, ..., \tau.1'_{m_1}, ..., \tau.n'_{m_n})_{m_1+\cdots+m_n}]$$

Aplicando ahora la regla (HMD-TUPLE) sobre (A.43) con $1 \leq j \leq n$, concluimos

$$A \times \Gamma_\Delta \vdash (e.1'_1, ..., e.1'_{m_1}, ..., e.n'_{m_n})_{m_1+\cdots+m_n} : (\tau.1'_1, ..., \tau.1'_{m_1}, ..., \tau.n'_{m_n})_{m_1+\cdots+m_n}$$

Caso (AR-LAM) Tenemos una derivación de la forma:

$$\frac{\Delta \,|\, \Gamma \vdash e : \eta \rightarrowtail [e'_i]_{i:1..m} \quad \Delta \,|\, \Gamma' \vdash v : \tau \rightarrowtail [v'_j]_{j:1..n>0}}{\Delta \,|\, \Gamma_v \vdash \lambda v.e : \tau \rightarrow \eta \rightarrowtail [(\lambda v'_1 ... v'_n . e'_i)]_{i:1..m}} \tag{A.44}$$

Por otro lado, tenemos una derivación de la forma

$$\frac{A_v, v : \tau \vdash e' \rightsquigarrow e : \eta}{A \vdash \lambda v.e' \rightsquigarrow \lambda v.e : \tau \rightarrow \eta}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.44), obtenemos

$$\Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..m} \tag{A.45}$$
$$(A_v, v : \tau) \times \Gamma_\Delta \vdash e'_i : \eta'_i \tag{A.46}$$

Utilizando la regla (MG-VAR) podemos afirmar que

$$A_v, v : \tau \vdash v \rightsquigarrow v : \tau$$

Aplicando hipótesis inductiva con la segunda premisa de (A.44) y con este resultado, obtenemos

$$\Delta \vdash \tau \xrightarrow{tar} [\tau'_i]_{i:1..n>0} \tag{A.47}$$
$$(A_v, v : \tau) \times \Gamma'_\Delta \vdash v'_i : \tau'_i \tag{A.48}$$

Aplicando la regla (TAR-FUN) sobre (A.45) y (A.47) obtenemos

$$\Delta \vdash \tau \to \eta \xrightarrow{tar} [\tau'_1 \to ... \to \tau'_n \to \eta'_i]_{i:1..m}$$

Utilizando (A.47), podemos afirmar que:

$$(A_v, v : \tau) \times \Gamma_\Delta =$$
$$(A_v \times \Gamma_{v\Delta}), v'_1 : \tau'_1, \ldots, v_n : \tau'_n =$$
$$(A \times \Gamma_{v\Delta})_{v_1,...,v_n}, v'_1 : \tau'_1, \ldots, v_n : \tau'_n$$

Aplicando esto nuevamente sobre (A.46)

$$(A \times \Gamma_{v\Delta})_{v'_1,...,v'_n}, v'_1 : \tau'_1, \ldots, v'_n : \tau'_n \vdash e'_i : \eta'_i$$

Utilizando n veces la regla (HMD-LAM), concluímos que

$$A \times \Gamma_{v\Delta} \vdash \lambda v'_1 ... v'_n . e'_i : \tau'_1 \to ... \to \tau'_n \to \eta'_i$$

Caso (AR-LAMS) Tenemos una derivación de la forma:

$$\frac{\Delta \,|\, \Gamma \vdash e : \eta \rightarrowtail [e'_i]_{i:1..m} \quad \Delta \,|\, \Gamma' \vdash v : \tau \rightarrowtail []}{\Delta \,|\, \Gamma_v \vdash \lambda v.e : \tau \to \eta \rightarrowtail [\$e'_i]_{i:1..m}} \tag{A.49}$$

Por otro lado, tenemos una derivación de la forma

$$\frac{A_v, v : \tau \vdash e' \rightsquigarrow e : \eta}{A \vdash \lambda v.e' \rightsquigarrow \lambda v.e : \tau \to \eta}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.49), obtenemos

$$\Delta \vdash \eta \xrightarrow{tar} [\eta'_i]_{i:1..m} \tag{A.50}$$
$$(A_v, v : \tau) \times \Gamma_\Delta \vdash e'_i : \eta'_i \tag{A.51}$$

Utilizando la regla (MG-VAR) podemos afirmar que

$$A_v, v : \tau \vdash v \rightsquigarrow v : \tau$$

Aplicando hipótesis inductiva con la segunda premisa de (A.49) y con este resultado, obtenemos

$$\Delta \vdash \tau \xrightarrow{tar} [] \tag{A.52}$$

Aplicando la regla (TAR-SUSP) sobre (A.50) y (A.52) obtenemos

$$\Delta \vdash \tau \to \eta \xrightarrow{tar} [\texttt{susp}\ \eta'_i]_{i:1..m}$$

Utilizando (A.52), podemos afirmar que:

$$(A_v, v : \tau) \times \Gamma_\Delta = A \times \Gamma_{v\Delta}$$

Aplicando esto nuevamente sobre (A.46)

$$A \times \Gamma_{v\Delta} \vdash e'_i : \eta'_i$$

Utilizando la regla (HMD-SUSP), concluímos que

$$A \times \Gamma_{v\Delta} \vdash \$e'_i : \texttt{susp}\ \eta'_i$$

Caso (AR-LET) Tenemos una derivación de la forma:

$$\frac{\Delta \mid \Gamma \vdash f : \eta \rightarrowtail [f'_t]_t \quad (\forall \vec{\tau}_{i:1..m} \in \Gamma(v))(\Delta; (\vec{t}, \vec{\tau}_i) \mid \Gamma'_i \vdash e : \tau \rightarrowtail [e'_i]_{j:1..n_i})}{\begin{array}{l}\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash \texttt{let}\ v_{\forall \vec{t}\,.\tau} = \Lambda \vec{t}\,.e\ \texttt{in}\ f : \eta \rightarrowtail [\ \texttt{let}\ v_{\rho(\Delta,\vec{\tau}_1),1} = e'_{1_1} \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad v_{\rho(\Delta,\vec{\tau}_1),n_1} = e'_{1_{n_1}} \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad v_{\rho(\Delta,\vec{\tau_m}),n_m} = e'_{m_{n_m}}\ \texttt{in}\ f'_t\ ]_t\end{array}} \tag{A.53}$$

Por otra parte, tenemos que

$$\frac{A \vdash e'' \rightsquigarrow e : \tau \quad A_v, v : \sigma \vdash f'' \rightsquigarrow f : \eta \quad \sigma = \forall \vec{t}\,.\tau = Gen(A, \tau)}{A \vdash \texttt{let}\ v = e''\ \texttt{in}\ f'' \rightsquigarrow \texttt{let}\ v_\sigma = \Lambda \vec{t}\,.e\ \texttt{in}\ f : \eta} \tag{A.54}$$

Aplicando la hipótesis inductiva sobre la primer premisa de (A.53), tenemos que

$$\Delta \vdash \eta \xrightarrow{tar} [\eta'_t]_t \tag{A.55}$$

$$(A_v, v : \sigma) \times \Gamma_\Delta \vdash f_t : \eta_t \tag{A.56}$$

Aplicando la propiedad 4.4 sobre (A.56), tenemos que

$$(A_v, v : \sigma) \times (\Gamma, \Gamma_1[\vec{t}/\vec{\tau}_1], \ldots, \Gamma_m[\vec{t}/\vec{\tau_m}])_\Delta \vdash f_t : \eta_t \tag{A.57}$$

Para cada $\vec{\tau}_i$ sean σ_{i_j} tales que

$$\Delta \vdash \sigma \xrightarrow{tar} [\sigma'_{i_j}]_{j:1..n_i} \tag{A.58}$$

es compatible con $\vec{\tau}_i$. De aquí en adelante, notaremos con $\vec{v} : \vec{\sigma'}$ a la secuencia de asignaciones de tipos $v_{\rho(\Delta,\vec{\tau}_1),1} : \sigma'_{1_1}, \ldots, v_{\rho(\Delta,\vec{\tau}_1),n_1} : \sigma'_{1_{n_1}}, \ldots, v_{\rho(\Delta,\vec{\tau_m}),n_m} : \sigma'_{1_{n_m}}$. Luego, como v sólo aparece en Γ y no en los Γ_i podemos afirmar que:

$$\begin{array}{l}(A_v, v : \sigma) \times (\Gamma, \Gamma_1[\vec{t}/\vec{\tau}_1], \ldots, \Gamma_m[\vec{t}/\vec{\tau_m}])_\Delta = \\ A_v \times (\Gamma_v, \Gamma_1[\vec{t}/\vec{\tau}_1], \ldots, \Gamma_m[\vec{t}/\vec{\tau_m}])_\Delta, \vec{v} : \vec{\sigma'} = \\ (A \times (\Gamma_v, \Gamma_1[\vec{t}/\vec{\tau}_1], \ldots, \Gamma_m[\vec{t}/\vec{\tau_m}])_\Delta)_{\vec{v}}, \vec{v} : \vec{\sigma'}\end{array}$$

Aplicando este resultado en (A.57) tenemos

$$(A \times (\Gamma_v, \Gamma_1[\vec{t}/\vec{\tau_1}], \ldots, \Gamma_m[\vec{t}/\vec{\tau_m}])_\Delta)_{\vec{v}}, \vec{v} : \vec{\sigma}' \vdash f_t : \eta_t \tag{A.59}$$

Por otro lado, aplicando m veces la hipótesis inductiva sobre el resto de las premisas de (A.53), obtenemos

$$\Delta; (\vec{t}, \vec{\tau_i}) \vdash \tau \xrightarrow{tar} [\tau_{i_j}]_{j:1..n_i} \tag{A.60}$$

$$A \times \Gamma'_{i\,\Delta;(\vec{t},\vec{\tau_i})} \vdash e_{i_j} : \tau_{i_j} \tag{A.61}$$

Notar que al aplicar la regla (TAR-SCHM) sobre (A.60) obtenemos (A.58), entonces $\sigma'_{i_j} = \forall \vec{t'_{i_j}}.\tau_{i_j}$. Aplicando ahora la propiedad 4.4, sobre A.61 tenemos

$$A \times (\Gamma_v, \Gamma'_1, \ldots, \Gamma'_m)_{\Delta;(\vec{t},\vec{\tau_i})} \vdash e_{i_j} : \tau_{i_j}$$

Luego, aplicando la propiedad 4.4, obtenemos

$$A \times (\Gamma_v, \Gamma'_1[\vec{t}/\vec{\tau_1}], \ldots, \Gamma'_i, \ldots, \Gamma'_m[\vec{t}/\vec{\tau_m}])_{\Delta;(\vec{t},\vec{\tau_i})} \vdash e_{i_j} : \tau_{i_j}$$

Aplicando ahora la propiedad 4.5, y el hecho de que los $\vec{t}$ no aparecen libres en los $\vec{\tau_i}$ ni en Γ_v, tenemos

$$A \times (\Gamma_v, \Gamma'_1[\vec{t}/\vec{\tau_1}], \ldots, \Gamma'_i[\vec{t}/\vec{\tau_i}], \ldots, \Gamma'_m[\vec{t}/\vec{\tau_m}])_\Delta \vdash e_{i_j} : \tau_{i_j}$$

Como $\vec{t}$ no aparecen libres en A, ni en los Γ, entonces las variables de tipo $\vec{t'_{i_j}}$ no aparecen libres en el contexto anterior. Luego, utilizando la regla (HMD-GEN) obtenemos

$$A \times (\Gamma_v, \Gamma'_1[\vec{t}/\vec{\tau_1}], \ldots \Gamma'_m[\vec{t}/\vec{\tau_m}])_\Delta \vdash e_{i_j} : \sigma'_{i_j} \tag{A.62}$$

Finalmente, aplicando la regla (HMD-LET) sobre (A.59) y (A.62) concluímos que

$$\begin{array}{l} A \times (\Gamma_v, \Gamma'_1[\vec{t}/\vec{\tau_1}], \ldots \Gamma'_m[\vec{t}/\vec{\tau_m}])_\Delta \vdash \texttt{let } v_{\rho(\Delta,\vec{\tau_1}),1} = e'_{1_1} \quad \ldots \quad : \eta'_t \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad v_{\rho(\Delta,\vec{\tau_1}),n_1} = e'_{1_{n_1}} \quad \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad v_{\rho(\Delta,\vec{\tau_m}),n_m} = e'_{m_{n_m}} \texttt{ in } f'_t \end{array}$$

□

Lema 4.10. Si $\Delta; (t, \eta) \,|\, \Gamma \vdash e : \tau \rightarrowtail [e'_i]_{i:1..n}$ entonces $\Delta \,|\, \Gamma[t/\eta] \vdash e[t/\eta] : \tau[t/\eta] \rightarrowtail [e'_i]_{i:1..n}$

Demostración. Lo demostraremos por inducción en el árbol de derivación de $\Delta; (t, \eta) \,|\, \Gamma \vdash e : \tau \rightarrowtail [e'_i]_{i:1..n}$

Caso (AR-VAR) Tenemos una derivación de la forma

$$\Delta;(t,\eta)\mid(v,\vec{\eta})\vdash v\ \vec{\eta}:\tau\rightarrowtail[v_{\rho(\Delta;(t,\eta),\vec{\eta}),i}]_{i:1..\rho(\Delta;(t,\eta),\tau)}$$

Aplicando la regla (AR-VAR) tenemos

$$\Delta\mid(v,\vec{\eta}[t/\eta])\vdash v\ \vec{\eta}[t/\eta]:\tau[t/\eta]\rightarrowtail[v_{\rho(\Delta,\vec{\eta}[t/\eta]),i}]_{i:1..\rho(\Delta,\tau[t/\eta])}$$

Utilizando el lema 3.12 obtenemos que

$$\Delta\mid(v,\eta[\vec{t/\eta}])\vdash v\ \vec{\eta}[t/\eta]:\tau[t/\eta]\rightarrowtail[v_{\rho(\Delta;(t,\eta),\vec{\eta}),i}]_{i:1..\rho(\Delta;(t,\eta),\tau)}$$

Caso (AR-AP) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\mid\Gamma\vdash e:\tau\to\eta'\rightarrowtail[e'_i]_{i:1..n}\quad\Delta;(t,\eta)\mid\Gamma'\vdash f:\tau\rightarrowtail[f'_j]_{j:1..m>0}}{\Delta;(t,\eta)\mid\Gamma,\Gamma'\vdash e\ f_\tau:\eta'\rightarrowtail[e'_i\ f'_1...f'_m]_{i:1..n}}\tag{A.63}$$

Aplicando hipótesis inductiva sobre ambas premisas obtenemos que

$$\Delta\mid\Gamma[t/\eta]\vdash e[t/\eta]:(\tau\to\eta')[t/\eta]\rightarrowtail[e'_i]_{i:1..n}$$
$$\Delta\mid\Gamma'[t/\eta]\vdash f[t/\eta]:\tau[t/\eta]\rightarrowtail[f'_j]_{j:1..m>0}$$

Por definición de sustitución y aplicando la regla (AR-AP) sobre estos resultados obtenemos

$$\Delta\mid\Gamma[t/\eta],\Gamma'[t/\eta]\vdash e[t/\eta]\ f[t/\eta]_{\tau[t/\eta]}:\eta'[t/\eta]\rightarrowtail[e'_i\ f'_1...f'_m]_{i:1..n}$$

Aplicando la definición de sustitución concluimos con

$$\Delta\mid(\Gamma,\Gamma')[t/\eta]\vdash(e\ f_\tau)[t/\eta]:\eta'[t/\eta]\rightarrowtail[e'_i\ f'_1...f'_m]_{i:1..n}$$

Caso (AR-APS) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\mid\Gamma\vdash e:\tau\to\eta'\rightarrowtail[e'_i]_{i:1..n}\quad\Delta;(t,\eta)\mid\Gamma'\vdash f:\tau\rightarrowtail[]}{\Delta;(t,\eta)\mid\Gamma\vdash e\ f_\tau:\eta'\rightarrowtail[\texttt{force}\ e'_i]_{i:1..n}}\tag{A.64}$$

Aplicando hipótesis inductiva sobre ambas premisas obtenemos que

$$\Delta\mid\Gamma[t/\eta]\vdash e[t/\eta]:(\tau\to\eta')[t/\eta]\rightarrowtail[e'_i]_{i:1..n}$$
$$\Delta\mid\Gamma'[t/\eta]\vdash f[t/\eta]:\tau[t/\eta]\rightarrowtail[]$$

Por definición de sustitución y aplicando la regla (AR-APS) sobre estos resultados obtenemos

$$\Delta;(t,\eta)\mid\Gamma[t/\eta]\vdash e[t/\eta]\ f[t/\eta]_{\tau[t/\eta]}:\eta'[t/\eta]\rightarrowtail[\texttt{force}\ e'_i]_{i:1..n}$$

Aplicando la definición de sustitución concluimos con

$$\Delta;(t,\eta)\mid\Gamma[t/\eta]\vdash(e\ f)_\tau[t/\eta]:\eta'[t/\eta]\rightarrowtail[\texttt{force}\ e'_i]_{i:1..n}$$

Caso (AR-SPROJ) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\,|\,\Gamma\vdash e:(\tau)^S_n \rightarrowtail [e'_i]_i \quad k=\sum_{s=1}^{p-1}\rho(\Delta;(t,\eta),\tau.s)}{\Delta;(t,\eta)\,|\,\Gamma\vdash \pi^S_{p,n}\ e_{(\tau)^S_n}:\tau.p \rightarrowtail [e'_{k+j}]_{j:1..\rho(\Delta;(t,\eta),\tau.p)}} \tag{A.65}$$

Aplicando hipótesis inductiva sobre la primer premisa de esta derivación tenemos que

$$\Delta\,|\,\Gamma[t/\eta]\vdash e[t/\eta]:(\tau)^S_n[t/\eta] \rightarrowtail [e'_i]_i$$

Utilizando el lema 3.12 sobre la segunda premisa de (A.65) tenemos

$$k=\sum_{s=1}^{p-1}\rho(\Delta,\tau.s[t/\eta])$$

Utilizando (AR-SPROJ) sobre estos resultados tenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash \pi^S_{p,n}\ e[t/\eta]_{(\tau)^S_n[t/\eta]}:\tau.p[t/\eta] \rightarrowtail [e'_{k+j}]_{j:1..\rho(\Delta,\tau.p[t/\eta])}$$

Finalmente, por definición de sustitución y por lema 3.12

$$\Delta\,|\,\Gamma[t/\eta]\vdash (\pi^S_{p,n}\ e_{(\tau)^S_n})[t/\eta]:\tau.p[t/\eta] \rightarrowtail [e'_{k+j}]_{j:1..\rho(\Delta;(t,\eta),\tau.p)}$$

Caso (AR-DPROJ) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\,|\,\Gamma\vdash e:(\tau)^D_n \rightarrowtail [e'] \quad m=\sum_{s}\rho(\Delta;(t,\eta),\tau.s) \quad k=\sum_{s=1}^{p-1}\rho(\Delta;(t,\eta),\tau.s)}{\Delta;(t,\eta)\,|\,\Gamma\vdash \pi^D_{p,n}\ e_{(\tau)^D_n}:\tau.p \rightarrowtail [\pi_{k+j,m}e']_{j:1..\rho(\Delta;(t,\eta),\tau.p)}} \tag{A.66}$$

Aplicando hipótesis inductiva sobre la primer premisa de esta derivación obtenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash e[t/\eta]:(\tau)^D_n[t/\eta] \rightarrowtail [e']$$

Utilizando el lema 3.12 sobre el resto de las premisas de (A.66) obtenemos

$$m=\sum_{s}\rho(\Delta,\tau.s[t/\eta])$$

$$k=\sum_{s=1}^{p-1}\rho(\Delta,\tau.s[t/\eta])$$

Utilizando (AR-DPROJ) sobre estos resultados obtenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash \pi^D_{p,n}\ e[t/\eta]_{(\tau)^D_n[t/\eta]}:\tau.p[t/\eta] \rightarrowtail [\pi_{k+j,m}e']_{j:1..\rho(\Delta,\tau.p[t/\eta])}$$

Utilizando el lema 3.12 y la definición de sustitución sobre este resultado obtenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash (\pi^D_{p,n}\ e_{(\tau)^D_n})[t/\eta]:\tau.p[t/\eta] \rightarrowtail [\pi_{k+j,m}e']_{j:1..\rho(\Delta;(t,\eta),\tau.p)}$$

Caso (AR-STUPLE) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\,|\,\Gamma\vdash e.i:\tau.i\rightarrowtail[e.i'_j]_{j:1..m_i}}{\Delta;(t,\eta)\,|\,\Gamma\vdash(e)^S_n:(\tau)^S_n\rightarrowtail[e.1'_j]_{j:1..m_1}\mathbin{\#}...\mathbin{\#}[e.n'_j]_{j:1..m_n}} \tag{A.67}$$

Aplicando hipótesis inductiva sobre las premisas de esta derivación obtenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash e.i[t/\eta]:\tau.i[t/\eta]\rightarrowtail[e.i'_j]_{j:1..m_i}$$

Aplicando (AR-STUPLE), y la definición de sustitución sobre este resultado tenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash(e)^S_n[t/\eta]:(\tau)^S_n[t/\eta]\rightarrowtail[e.1'_j]_{j:1..m_1}\mathbin{\#}...\mathbin{\#}[e.n'_j]_{j:1..m_n}$$

Caso (AR-DTUPLE) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\,|\,\Gamma\vdash e.i:\tau.i\rightarrowtail[e.i'_j]_{j:1..m_i}}{\Delta;(t,\eta)\,|\,\Gamma\vdash(e)^D_n:(\tau)^D_n\rightarrowtail[(e.1'_1,...,e.1'_{m_1},...,e.n'_{m_n})_{m_1+...+m_n}]} \tag{A.68}$$

Aplicando hipótesis inductiva sobre las premisas de esta derivación obtenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash e.i[t/\eta]:\tau.i[t/\eta]\rightarrowtail[e.i'_j]_{j:1..m_i}$$

Aplicando nuevamente (AR-DTUPLE), y la definición de sustitución tenemos

$$\Delta\,|\,\Gamma\vdash(e)^D_n[t/\eta]:(\tau)^D_n[t/\eta]\rightarrowtail[(e.1'_1,...,e.1'_{m_1},...,e.n'_{m_n})_{m_1+...+m_n}]$$

Caso (AR-LAM) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\,|\,\Gamma\vdash e:\eta'\rightarrowtail[e'_i]_i\quad\Delta;(t,\eta)\,|\,\Gamma'\vdash u:\tau\rightarrowtail[u'_j]_{j:1..n>0}}{\Delta;(t,\eta)\,|\,\Gamma_u\vdash\lambda u.e:\tau\rightarrow\eta'\rightarrowtail[(\lambda u'_1...u'_n.e'_i)]_i} \tag{A.69}$$

Aplicando hipótesis inductiva sobre las premisas de (A.69) obtenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash e[t/\eta]:\eta'[t/\eta]\rightarrowtail[e'_i]_i$$
$$\Delta\,|\,\Gamma'[t/\eta]\vdash u:\tau[t/\eta]\rightarrowtail[u'_j]_{j:1..n>0}$$

Aplicando (AR-LAM) sobre estos resultados tenemos

$$\Delta\,|\,\Gamma_u[t/\eta]\vdash\lambda u.e[t/\eta]:\tau[t/\eta]\rightarrow\eta'[t/\eta]\rightarrowtail[(\lambda u'_1...u'_n.e'_i)]_i$$

Finalmente, por definición de sustitución

$$\Delta\,|\,\Gamma_u[t/\eta]\vdash(\lambda u.e)[t/\eta]:(\tau\rightarrow\eta')[t/\eta]\rightarrowtail[(\lambda u'_1...u'_n.e'_i)]_i$$

Caso (AR-LAMS) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\,|\,\Gamma\vdash e:\eta'\rightarrowtail[e'_i]_i \quad \Delta;(t,\eta)\,|\,\Gamma'\vdash u:\tau\rightarrowtail[]}{\Delta;(t,\eta)\,|\,\Gamma_u\vdash\lambda u.e:\tau\to\eta'\rightarrowtail[\$e'_i]_i} \tag{A.70}$$

Aplicando hipótesis inductiva sobre las premisas de (A.70) obtenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash e[t/\eta]:\eta'[t/\eta]\rightarrowtail[e'_i]_i$$
$$\Delta\,|\,\Gamma'[t/\eta]\vdash u:\tau[t/\eta]\rightarrowtail[]$$

Aplicando (AR-LAMS) sobre estos resultados tenemos

$$\Delta\,|\,\Gamma_u[t/\eta]\vdash\lambda u.e[t/\eta]:\tau[t/\eta]\to\eta'[t/\eta]\rightarrowtail[\$e'_i]_i$$

Finalmente, por definición de sustitución

$$\Delta\,|\,\Gamma_u[t/\eta]\vdash(\lambda u.e)[t/\eta]:(\tau\to\eta')[t/\eta]\rightarrowtail[\$e'_i]_i$$

Caso (AR-LET) Tenemos una derivación de la forma

$$\frac{\Delta;(t,\eta)\,|\,\Gamma\vdash f:\eta'\rightarrowtail[f'_t]_{t:1..n} \quad (\forall\vec{\tau'}_{i:1..m}\in\Gamma(u))(\Delta;(t,\eta);(\vec{s},\vec{\tau'_i})\,|\,\Gamma_i\vdash e:\tau\rightarrowtail[e'_i]_{j:1..n_i})}{\begin{array}{l}\Delta;(t,\eta)\,|\,\Gamma_u,\Gamma_i[\vec{s}/\vec{\tau'_i}]\vdash\texttt{let}\ u_{\forall\vec{s}.\tau}=\Lambda\vec{s}.e\ \texttt{in}\ f:\eta'\rightarrowtail[\ \texttt{let}\ u_{\rho(\Delta,\vec{\tau'_1}),1}=e'_{1_1}\quad ...\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_1}),n_1}=e'_{1_{n_1}}\quad ...\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_m}),n_m}=e'_{m_{n_m}}\ \texttt{in}\ f'_t\ ]_{t:1..n}\end{array}} \tag{A.71}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.71) tenemos

$$\Delta\,|\,\Gamma[t/\eta]\vdash f[t/\eta]:\eta'[t/\eta]\rightarrowtail[f'_t]_{t:1..n} \tag{A.72}$$

Por otra parte, utilizando el lema 3.12 podemos afirmar que

$$\Delta;(t,\eta);(\vec{s},\vec{\tau'_i})=\Delta;(\vec{s},\tau'_i[\vec{t}/\eta]);(t,\eta)$$

Además sabemos que

$$\vec{\tau}'_i\in\Gamma(u)\Leftrightarrow\vec{\tau}'_i[t/\eta]\in\Gamma[t/\eta](u)$$

Utilizando estos resultados, podemos aplicar hipótesis inductiva sobre el resto de las premisas de (A.71) y obtenemos que para todo $\vec{\tau}'_{i:1..m}[t/\eta]\in\Gamma[t/\eta](u)$

$$\Delta;(\vec{s},\tau'_i[\vec{t}/\eta])\,|\,\Gamma_i[t/\eta]\vdash e[t/\eta]:\tau[t/\eta]\rightarrowtail[e'_i]_{j:1..n_i} \tag{A.73}$$

Aplicando (AR-LET) sobre (A.72) y (A.73), y luego la definición de sustitución tenemos

$$\begin{array}{l}\Delta\,|\,\Gamma''\vdash(\texttt{let}\ u_{\forall\vec{s}.\tau}=\Lambda\vec{s}.e\ \texttt{in}\ f)[t/\eta]:\eta'[t/\eta]\rightarrowtail[\ \texttt{let}\ u_{\rho(\Delta,\vec{\tau'_1}),1}=e'_{1_1}\quad ...\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_1}),n_1}=e'_{1_{n_1}}\quad ...\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_m}),n_m}=e'_{m_{n_m}}\ \texttt{in}\ f'_t\ ]_{t:1..n}\end{array}$$

donde,

$$\Gamma'' = \Gamma[t/\eta]_u, \Gamma_i[t/\eta][\vec{s}/\vec{\tau_i'}[t/\eta]]$$

Finalmente, por propiedad de las sustituciones

$$\Gamma'' = \Gamma_u, \Gamma_i[\vec{s}/\vec{\tau_i'}][t/\eta]$$

□

Lema 4.11. Sea $e' = \Lambda\vec{t}.e$ tal que $v \notin FV(e)$, $v \in FV(f)$ y $(\forall\vec{\tau}_{i:1..m} \in \Gamma(v))$ $(\Delta; (\vec{t}, \vec{\tau_i}) \mid \Gamma_i' \vdash e : \eta \rightarrowtail [e'_{i_j}]_{j:1..n_i})$, si $\Delta \mid \Gamma \vdash f : \tau \rightarrowtail [f_i']_{i:1..n}$ entonces

$$\Delta \mid \Gamma_v, \Gamma_i'[\vec{t}/\vec{\tau_i}] \vdash f[v/e'] : \tau \rightarrowtail [f_i'[v_{\rho(\vec{\Delta},\vec{\tau_1})}/\vec{e'_1}]...[v_{\rho(\vec{\Delta},\vec{\tau_m})}/\vec{e'_m}]]_{i:1..n}$$

Demostración. Lo demostraremos por inducción sobre el árbol de derivación de $\Delta \mid \Gamma \vdash f : \tau \rightarrowtail [f_i']_{i:1..n}$.

Caso (AR-VAR) Tenemos una derivación de la forma

$$\Delta \mid (v, \vec{\eta}) \vdash v\ \vec{\eta} : \tau \rightarrowtail [v_{\rho(\Delta,\vec{\eta}),i}]_{i:1..\rho(\Delta,\tau)}$$

Podemos asegurar que la variable es v puesto que pertenece a las variables libres del término. Sabemos, por hipótesis, que para $1 \leq i \leq m$

$$\Delta; (\vec{t}, \vec{\tau_i}) \mid \Gamma_i' \vdash e : \eta \rightarrowtail [e'_{i_j}]_{j:1..n_i}$$

Como $\Gamma = (v, \vec{\eta})$, tenemos que $m = 1$ y

$$\Delta; (\vec{t}, \vec{\eta}) \mid \Gamma_1' \vdash e : \eta \rightarrowtail [e'_{1_j}]_{j:1..n_1}$$

Utilizando el lema 4.10 obtenemos

$$\Delta \mid \Gamma_1'[\vec{t}/\vec{\eta}] \vdash e[\vec{t}/\vec{\eta}] : \eta[\vec{t}/\vec{\eta}] \rightarrowtail [e'_{1_j}]_{j:1..n_1}$$

Como $\tau \leq_{\vec{\eta}} \forall\vec{t}.\eta$, $\tau = \eta[\vec{t}/\vec{\eta}]$ luego

$$\Delta \mid \Gamma_1'[\vec{t}/\vec{\eta}] \vdash e[\vec{t}/\vec{\eta}] : \tau \rightarrowtail [e'_{1_j}]_{j:1..n_1}$$

Por definición de sustitución

$$\Delta \mid \Gamma_1'[\vec{t}/\vec{\eta}] \vdash (v\ \vec{\eta})[v/e'] : \tau \rightarrowtail [v_{\rho(\Delta,\vec{\eta}),i}[v_{\rho(\vec{\Delta},\vec{\tau_1})}/\vec{e'_1}]...[v_{\rho(\vec{\Delta},\vec{\tau_m})}/\vec{e'_m}]]_{i:1..\rho(\Delta,\tau)}$$

Caso (AR-AP) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e'' : \tau \to \eta' \rightarrowtail [e'''_i]_{i:1..n} \quad \Delta \mid \Gamma' \vdash f : \tau \rightarrowtail [f'_j]_{j:1..m>0}}{\Delta \mid \Gamma, \Gamma' \vdash e''\ f_\tau : \eta' \rightarrowtail [e'''_i\ f'_1 ... f'_m]_{i:1..n}} \tag{A.74}$$

De aquí en adelante notaremos $[\vec{v}/\vec{e'}]$ a las sustituciones $[v_{\rho(\vec{\Delta},\vec{\tau_1})}/\vec{e'_1}]...[v_{\rho(\vec{\Delta},\vec{\tau_m})}/\vec{e'_m}]$
Aplicando hipótesis inductiva sobre la primer premisa de (A.74) tenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash e''[v/e'] : \tau \to \eta' \rightarrowtail [e'''_i[\vec{v}/\vec{e'}]]_{i:1..n}$$

Por otra parte, aplicando hipótesis inductiva sobre la segunda premisa de (A.74) tenemos

$$\Delta \mid \Gamma'_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash f[v/e'] : \tau \rightarrowtail [f'_i[\vec{v}/\vec{e'}]]_{i:1..m>0}$$

Aplicando (AR-AP) y la definición de sustitución sobre estos resultados obtenemos

$$\Delta \mid \Gamma'_v, \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash (e\ f_\tau)[v/e'] : \eta' \rightarrowtail [(e'''_i\ f'_1[\vec{v}/\vec{e'}]]_{i:1..m>0}$$

Caso (AR-APS) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e'' : \tau \to \eta' \rightarrowtail [e'''_i]_{i:1..n} \quad \Delta \mid \Gamma' \vdash f : \tau \rightarrowtail []}{\Delta \mid \Gamma \vdash e''\ f_\tau : \eta' \rightarrowtail [\texttt{force}\ e'''_i]_{i:1..n}} \tag{A.75}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.75) tenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash e''[v/e'] : \tau \to \eta' \rightarrowtail [e'''_i[\vec{v}/\vec{e'}]]_{i:1..n}$$

Por otra parte, aplicando hipótesis inductiva sobre la segunda premisa de (A.75) tenemos

$$\Delta \mid \Gamma'_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash f[v/e'] : \tau \rightarrowtail []$$

Aplicando (AR-APS) sobre estos resultados obtenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash e''[v/e']\ f_\tau[v/e'] : \eta' \rightarrowtail [\texttt{force}\ (e'''_i[\vec{v}/\vec{e'}])]_{i:1..n}$$

Finalmente, por definición de sustitución

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash (e''\ f_\tau)[v/e'] : \eta' \rightarrowtail [(\texttt{force}\ e'''_i)[\vec{v}/\vec{e'}]]_{i:1..n}$$

Caso (AR-SPROJ) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e'' : (\tau)^S_n \rightarrowtail [e'''_i]_i \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \mid \Gamma \vdash \pi^S_{p,n}\ e''_{(\tau)^S_n} : \tau.p \rightarrowtail [e'''_{k+j}]_{j:1..\rho(\Delta,\tau.p)}} \tag{A.76}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.76) obtenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash e''[v/e'] : (\tau)^S_n \rightarrowtail [e'''_i[\vec{v}/\vec{e'}]]_i$$

Aplicando nuevamente (AR-SPROJ) sobre este resultado y la segunda premisa de (A.76) obtenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash \pi^S_{p,n}\ e''[v/e']_{(\tau)^S_n} : \tau.p \rightarrowtail [e'''_{k+j}[\vec{v}/\vec{e'}]]_{j:1..\rho(\Delta,\tau.p)}$$

Finalmente, por definición de sustitución

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash (\pi^S_{p,n}\ e''_{(\tau)^S_n})[v/e'] : \tau.p \rightarrowtail [e'''_{k+j}[\vec{v}/\vec{e'}]]_{j:1..\rho(\Delta,\tau.p)}$$

Caso (AR-DPROJ) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e'' : (\tau)^D_n \rightarrowtail [e'''] \quad m = \sum_s \rho(\Delta, \tau.s) \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \mid \Gamma \vdash \pi^D_{p,n}\ e''_{(\tau)^D_n} : \tau.p \rightarrowtail [\pi_{k+j,m} e'''']_{j:1..\rho(\Delta,\tau.p)}} \tag{A.77}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.77) obtenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash e''[v/e'] : (\tau)^D_n \rightarrowtail [e'''[\vec{v}/\vec{e'}]]$$

Aplicando nuevamente (AR-DPROJ) sobre este resultado y el resto de las premisas de (A.76) obtenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash \pi^D_{p,n}\ e[v/e']''_{(\tau)^D_n} : \tau.p \rightarrowtail [\pi_{k+j,m}(e'''[\vec{v}/\vec{e'}])]_{j:1..\rho(\Delta,\tau.p)}$$

Finalmente, por definición de sustitución

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash (\pi^D_{p,n}\ e''_{(\tau)^D_n})[v/e'] : \tau.p \rightarrowtail [(\pi_{k+j,m} e''')[\vec{v}/\vec{e'}]]_{j:1..\rho(\Delta,\tau.p)}$$

Caso (AR-STUPLE) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e.i'' : \tau.i \rightarrowtail [e.i'''_j]_{j:1..m_i}}{\Delta \mid \Gamma \vdash (e'')^S_n : (\tau)^S_n \rightarrowtail [e.1'''_j]_{j:1..m_1} \Vdash \ldots \Vdash [e.n'''_j]_{j:1..m_n}} \tag{A.78}$$

Aplicando hipótesis inductiva sobre las premisas de (A.78) tenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash e.i''[v/e'] : \tau.i \rightarrowtail [e.i'''_j[\vec{v}/\vec{e'}]]_{j:1..m_i}$$

Aplicando nuevamente (AR-STUPLE)sobre estos resultados obtenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash (e''[v/e'])^S_n : (\tau)^S_n \rightarrowtail [e.1'''_j[\vec{v}/\vec{e'}]]_{j:1..m_1} \Vdash \ldots \Vdash [e.n'''_j[\vec{v}/\vec{e'}]]_{j:1..m_n}$$

Finalmente, por definición de sustitución

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash (e'')^S_n[v/e'] : (\tau)^S_n \rightarrowtail [e.1'''_j[\vec{v}/\vec{e'}]]_{j:1..m_1} \Vdash \ldots \Vdash [e.n'''_j[\vec{v}/\vec{e'}]]_{j:1..m_n}$$

Caso (AR-DTUPLE) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e.i'' : \tau.i \rightarrowtail [e.i'''_j]_{j:1..m_i}}{\Delta \mid \Gamma \vdash (e'')^D_n : (\tau)^D_n \rightarrowtail [(e.1'''_1, ..., e.1'''_{m_1}, ..., e.n'''_{m_n})_{m_1+...+m_n}]} \tag{A.79}$$

Aplicando hipótesis inductiva sobre las premisas de (A.79)

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash e.i''[v/e'] : \tau.i \rightarrowtail [e.i'''_j[\vec{v}/\vec{e'}]]_{j:1..m_i}$$

Aplicando nuevamente la regla (AR-DTUPLE) sobre estos resultados tenemos que

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash (e''[v/e'])^D_n : (\tau)^D_n \rightarrowtail [(e.1'''_1[\vec{v}/\vec{e'}], ..., e.n'''_{m_n}[\vec{v}/\vec{e'}])_{m_1+...+m_n}]$$

Finalmente, por definición de sustitución tenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash (e'')^D_n[v/e'] : (\tau)^D_n \rightarrowtail [(e.1'''_1, ..., e.n'''_{m_n})_{m_1+...+m_n}[\vec{v}/\vec{e'}]]$$

Caso (AR-LAM) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e'' : \eta' \rightarrowtail [e'''_i]_i \quad \Delta \mid \Gamma' \vdash u : \tau \rightarrowtail [u'_j]_{j:1..n>0}}{\Delta \mid \Gamma_u \vdash \lambda u.e'' : \tau \rightarrow \eta' \rightarrowtail [(\lambda u'_1 ... u'_n . e'''_i)]_i} \tag{A.80}$$

Podemos asumir sin perder generalidad que $u \neq v$. Aplicando hipótesis inductiva sobre las premisas de (A.80) tenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash e''[v/e'] : \eta' \rightarrowtail [e'''_i[\vec{v}/\vec{e'}]]_i$$
$$\Delta \mid \Gamma'_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash u : \tau \rightarrowtail [u'_j]_{j:1..n>0}$$

Aplicando nuevamente la regla (AR-LAM) sobre estos resultados obtenemos

$$\Delta \mid (\Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}])_u \vdash \lambda u.(e''[v/e']) : \tau \rightarrow \eta' \rightarrowtail [(\lambda u'_1 ... u'_n . (e'''_i)[\vec{v}/\vec{e'}])]_i$$

Finalmente, por definición de sustitución y el hecho de que $u \notin \Gamma'_i$

$$\Delta \mid \Gamma_{u,v}, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash (\lambda u.e'')[v/e'] : \tau \rightarrow \eta' \rightarrowtail [(\lambda u'_1 ... u'_n . e'''_i)[\vec{v}/\vec{e'}]]_i$$

Caso (AR-LAMS) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e'' : \eta' \rightarrowtail [e'''_i]_i \quad \Delta \mid \Gamma' \vdash u : \tau \rightarrowtail []}{\Delta \mid \Gamma_u \vdash \lambda u.e : \tau \rightarrow \eta' \rightarrowtail [\$e'''_i]_i} \tag{A.81}$$

Aplicando hipótesis inductiva sobre las premisas de (A.81) tenemos

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash e''[v/e'] : \eta' \rightarrowtail [e'''_i[\vec{v}/\vec{e'}]]_i$$
$$\Delta \mid \Gamma'_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash u : \tau \rightarrowtail []$$

Aplicando nuevamente la regla (AR-LAM) sobre estos resultados obtenemos

$$\Delta \mid (\Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}])_u \vdash \lambda u.(e[v/e']) : \tau \to \eta' \rightarrowtail [\$(e'''_i[\vec{v}/\vec{e'}])]_i$$

Finalmente, por definición de sustitución y el hecho de que $u \notin \Gamma'_i$

$$\Delta \mid \Gamma_{u,v}, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash (\lambda u.e)[v/e'] : \tau \to \eta' \rightarrowtail [(\$e'''_i)[\vec{v}/\vec{e'}]]_i$$

Caso (AR-LET) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash f : \eta' \rightarrowtail [f'_t]_{t:1..n} \quad (\forall \vec{\tau'}_{i:1..m} \in \Gamma(u))(\Delta; (\vec{s}, \vec{\tau'_i}) \mid \Gamma''_i \vdash e'' : \tau \rightarrowtail [e'''_i]_{j:1..n_i})}{\begin{array}{l}\Delta \mid \Gamma_u, \Gamma''_i[\vec{s}/\vec{\tau'_i}] \vdash \texttt{let}\ u_{\forall \vec{s}.\tau} = \Lambda \vec{s}.e''\ \texttt{in}\ f : \eta' \rightarrowtail [\ \texttt{let}\ u_{\rho(\Delta,\vec{\tau'_1}),1} = e'''_{1_1} \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_1}),n_1} = e'''_{1_{n_1}} \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_m}),n_m} = e'''_{m_{n_m}}\ \texttt{in}\ f'_t\ ]_{t:1..n}\end{array}} \tag{A.82}$$

Aplicando la hipótesis inductiva sobre la primer premisa de (A.82), obtenemos que

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash f[v/e'] : \eta' \rightarrowtail [f'_t[\vec{v}/\vec{e'}]]_{t:1..n} \tag{A.83}$$

Aplicando hipótesis inductiva sobre el resto de las premisas de (A.82), tenemos que para $1 \leq i \leq m$ vale

$$\Delta; (\vec{s}, \vec{\tau'_i}) \mid \Gamma''_{i_v}, \Gamma'_i[\vec{t}/\vec{\tau_i}] \vdash e''[v/e'] : \tau \rightarrowtail [e'''_i[\vec{v}/\vec{e'}]]_{j:1..n_i} \tag{A.84}$$

Sin perder generalidad, podemos asumir que $u \notin FV(e)$, luego $u \notin \Gamma'_i$, por lo que

$$\vec{\tau'}_{i:1..m} \in \Gamma(u) \Leftrightarrow \vec{\tau'}_{i:1..m} \in (\Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}])(u)$$

Aplicando (AR-LET) sobre (A.83) y (A.84) tenemos

$$\begin{array}{l}\Delta \mid \Gamma''' \vdash \texttt{let}\ u_{\forall \vec{s}.\tau} = \Lambda \vec{s}.e''[v/e']\ \texttt{in}\ f[v/e'] : \eta' \rightarrowtail [\ \texttt{let}\ u_{\rho(\Delta,\vec{\tau'_1}),1} = e'''_{1_1}[\vec{v}/\vec{e'}] \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_1}),n_1} = e'''_{1_{n_1}}[\vec{v}/\vec{e'}] \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_m}),n_m} = e'''_{m_{n_m}}[\vec{v}/\vec{e'}]\ \texttt{in}\ f'_t[\vec{v}/\vec{e'}]\ ]_{t:1..n}\end{array}$$

donde,

$$\Gamma''' = (\Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau_i}])_u, (\Gamma''_{i_v}, \Gamma'_i[\vec{t}/\vec{\tau_i}])[\vec{s}/\vec{\tau'_i}]$$

Luego, por definición de sustitución,

$$\begin{array}{l}\Delta \mid \Gamma''' \vdash (\texttt{let}\ u_{\forall \vec{s}.\tau} = \Lambda \vec{s}.e''\ \texttt{in}\ f)[v/e'] : \eta' \rightarrowtail [\ (\texttt{let}\ u_{\rho(\Delta,\vec{\tau'_1}),1} = e'''_{1_1} \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_1}),n_1} = e'''_{1_{n_1}} \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u_{\rho(\Delta,\vec{\tau'_m}),n_m} = e'''_{m_{n_m}}\ \texttt{in}\ f'_t)[\vec{v}/\vec{e'}]\ ]_{t:1..n}\end{array}$$

Finalmente, como $u \notin \Gamma'_i$ y $\vec{s} \notin FV(\Gamma'_i)$ vale que

$$\Gamma''' = (\Gamma_u, \Gamma''_{i_v}[\vec{s}/\vec{\tau'_i}])_v, \Gamma'_i[\vec{t}/\vec{\tau}_i]$$

□

Teorema 4.12. Si $\Delta \mid \Gamma \vdash e : \tau \rightarrowtail [e'_i]_{i:1..n}$ y $e \Downarrow_o f$ entonces existen términos f_i tales que $e'_i \Downarrow_d f_i$ y $\Delta \mid \Gamma' \vdash f : \tau \rightarrowtail [f_i]_{i:1..n}$

Demostración. Lo demostraremos por inducción en el árbol de derivación de $e \Downarrow_o f$. Separaremos en casos de acuerdo a la última regla de derivación utilizada para $\Delta \mid \Gamma \vdash e : \tau \rightarrowtail [e'_i]_{i:1..n}$.

Caso (AR-LAM) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e : \eta \rightarrowtail [e'_i]_{i:1..m} \quad \Delta \mid \Gamma' \vdash v : \tau \rightarrowtail [v'_j]_{j:1..n>0}}{\Delta \mid \Gamma_v \vdash \lambda v.e : \tau \to \eta \rightarrowtail [(\lambda v'_1 ... v'_n . e'_i)]_{i:1..m}} \tag{A.85}$$

Sabemos además que

$$\lambda v.e \Downarrow_o \lambda v.e$$

Luego, existen los término $f_i = \lambda v'_1 ... v'_n . e'_i$, con $1 \leq i \leq m$ tales que por la conclusión de (A.85)

$$\Delta \mid \Gamma_v \vdash \lambda v.e : \tau \to \eta \rightarrowtail [f_i]_{i:1..m}$$

Y utilizando (SD-LAM)

$$\lambda v'_1 ... v'_n . e'_i \Downarrow_d f_i$$

Caso (AR-LAMS) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e : \eta \rightarrowtail [e'_i]_{i:1..m} \quad \Delta \mid \Gamma' \vdash v : \tau \rightarrowtail []}{\Delta \mid \Gamma_v \vdash \lambda v.e : \tau \to \eta \rightarrowtail [\$e'_i]_{i:1..m}} \tag{A.86}$$

Sabemos además que

$$\lambda v.e \Downarrow_o \lambda v.e$$

Luego, existen los términos $f_i = \$e'_i$, con $1 \leq i \leq m$ tales, tales que por la conclusión de (A.86)

$$\Delta \mid \Gamma_v \vdash \lambda v.e : \tau \to \eta \rightarrowtail [f_i]_{i:1..m}$$

Y utilizando (SD-LAM)

$$\$e'_i \Downarrow_d f_i$$

Caso (AR-AP) Tenemos una derivación de la forma

$$\frac{\Delta \mid \Gamma \vdash e : \tau \to \eta \rightarrowtail [e'_i]_{i:1..n} \quad \Delta \mid \Gamma' \vdash f : \tau \rightarrowtail [f'_j]_{j:1..m>0}}{\Delta \mid \Gamma, \Gamma' \vdash e\ f_\tau : \eta \rightarrowtail [e'_i\ f'_1 ... f'_m]_{i:1..n}} \tag{A.87}$$

Por otra parte, tenemos entonces una derivación de la forma

$$\frac{e \Downarrow_o \lambda v.e' \quad e'[v/f] \Downarrow_o f'}{e\ f_\tau \Downarrow_o f'} \tag{A.88}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.87) y (A.88) obtenemos que existen e''_i, con $i \leq 1 \leq n$, tales que

$$e'_i \Downarrow_d e''_i \tag{A.89}$$
$$\Delta \mid \Gamma'' \vdash \lambda v.e' \rightarrowtail [e''_i]_{i:1..n} \tag{A.90}$$

Luego como, por la segunda premisa de (A.87), $\rho(\Delta, \tau) > 0$ para probar (A.90) tenemos la siguiente derivación

$$\frac{\Delta \mid \Gamma_1 \vdash e' : \eta \rightarrowtail [b_i]_{i:1..n} \quad \Delta \mid \Gamma_2 \vdash v : \tau \rightarrowtail [v'_j]_{j:1..m>0}}{\Delta \mid \Gamma'' \vdash \lambda v.e' : \tau \to \eta \rightarrowtail [(\lambda v'_1 ... v'_m.b_i)]_{i:1..n}} \tag{A.91}$$

Luego, como $e''_i = \lambda v'_1 ... v'_m.b_i$, por (A.89)

$$e'_i \Downarrow_d \lambda v'_1 ... v'_m.b_i \tag{A.92}$$

Por otra parte, aplicando el lema 4.11 sobre la primer premisa de (A.91) y con (A.87) obtenemos

$$\Delta \mid \Gamma_{1v}, \Gamma' \vdash e'[v/f] : \eta \rightarrowtail [b_i[\vec{v}/\vec{f'}]]_{i:1..n}$$

Aplicando hipótesis inductiva sobre este resultado y la segunda premisa de (A.88) obtenemos que existen b''_i tales que

$$b_i[\vec{v}/\vec{f'}] \Downarrow_d b''_i \tag{A.93}$$
$$\Delta \mid \Gamma_3 \vdash f' \rightarrowtail [b''_i]_{i:1..n} \tag{A.94}$$

Finalmente, aplicando m veces la regla (SD-APP) sobre (A.92) y (A.93) obtenemos

$$e'_i\ f'_1 ... f'_m \Downarrow_d b''_i$$

Caso (AR-APS)

$$\frac{\Delta \mid \Gamma \vdash e : \tau \to \eta \rightarrowtail [e'_i]_{i:1..n} \quad \Delta \mid \Gamma' \vdash f : \tau \rightarrowtail []}{\Delta \mid \Gamma \vdash e\ f_\tau : \eta \rightarrowtail [\texttt{force}\ e'_i]_{i:1..n}} \tag{A.95}$$

Por otra parte, tenemos entonces una derivación de la forma

$$\frac{e \Downarrow_o \lambda v.e' \quad e'[v/f] \Downarrow_o f'}{e\ f_\tau \Downarrow_o f'} \tag{A.96}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.95) y (A.96) obtenemos que existen e''_i, con $i \leq 1 \leq n$, tales que

$$e'_i \Downarrow_d e''_i \tag{A.97}$$
$$\Delta \mid \Gamma'' \vdash \lambda v.e' \rightarrowtail [e''_i]_{i:1..n} \tag{A.98}$$

Luego como, por la segunda premisa de (A.95), $\rho(\Delta, \tau) = 0$ para probar (A.98) tenemos la siguiente derivación

$$\frac{\Delta \mid \Gamma_1 \vdash e' : \eta \rightarrowtail [b_i]_{i:1..n} \quad \Delta \mid \Gamma_2 \vdash v : \tau \rightarrowtail []}{\Delta \mid \Gamma'' \vdash \lambda v.e' : \tau \to \eta \rightarrowtail [\$b_i]_{i:1..n}} \tag{A.99}$$

Luego, como $e''_i = \$b_i$, por (A.97)

$$e'_i \Downarrow_d \$b_i \tag{A.100}$$

Por otra parte, aplicando el lema 4.11 sobre la primer premisa de (A.99) y con la segunda premisa de (A.95) obtenemos

$$\Delta \mid \Gamma_{1v}, \Gamma' \vdash e'[v/f] : \eta \rightarrowtail [b_i]_{i:1..n}$$

Aplicando hipótesis inductiva sobre este resultado y la segunda premisa de (A.88) obtenemos que existen b''_i tales que

$$b_i \Downarrow_d b''_i \tag{A.101}$$
$$\Delta \mid \Gamma_3 \vdash f' \rightarrowtail [b''_i]_{i:1..n} \tag{A.102}$$

Finalmente, aplicando la regla (SD-FORCE) sobre (A.100) y (A.101) obtenemos

$$\texttt{force}\ e'_i \Downarrow_d b''_i$$

Caso (AR-SPROJ)

$$\frac{\Delta \mid \Gamma \vdash e : (\tau)^S_n \rightarrowtail [e'_i]_{i:1..m} \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \mid \Gamma \vdash \pi^S_{p,n}\ e_{(\tau)^S_n} : \tau.p \rightarrowtail [e'_{k+j}]_{j:1..\rho(\Delta,\tau.p)}} \tag{A.103}$$

Por otro lado tenemos una derivación de la forma

$$\frac{e \Downarrow_o (e')^S_n}{\pi^S_{p,n} e \Downarrow_o e.p'} \tag{A.104}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.103) y de (A.104) obtenemos que existen b_i, tales que

$$e'_i \Downarrow_d b_i \tag{A.105}$$
$$\Delta \,|\, \Gamma' \vdash (e')^S_n : (\tau)^S_n \rightarrowtail [b_i]_{i:1..m} \tag{A.106}$$

Pero, para probar (A.106) tenemos una derivación de la forma

$$\frac{\Delta \,|\, \Gamma' \vdash e.i' : \tau.i \rightarrowtail [e.i''_j]_{j:1..m_i}}{\Delta \,|\, \Gamma' \vdash (e')^S_n : (\tau)^S_n \rightarrowtail [e.1''_j]_{j:1..m_1} \Vdash ... \Vdash [e.n''_j]_{j:1..m_n}} \tag{A.107}$$

Aplicando este resultado sobre (A.106) tenemos que

$$b_i = e.t''_j \qquad \text{,donde } i = \sum_{s=1}^{t-1} \rho(\Delta, \tau.s) + j \tag{A.108}$$

Luego, por la p-ésima premisa de A.107 tenemos

$$\Delta \,|\, \Gamma' \vdash e.p' : \tau.p \rightarrowtail [e.p''_j]_{j:1..m_p}$$

Aplicando (A.108) a este resultado tenemos

$$\Delta \,|\, \Gamma' \vdash e.p' : \tau.p \rightarrowtail [b_{k+j}]_{j:1..m_p}$$

Caso (AR-DPROJ)

$$\frac{\Delta \,|\, \Gamma \vdash e : (\tau)^D_n \rightarrowtail [e'] \quad m = \sum_s \rho(\Delta, \tau.s) \quad k = \sum_{s=1}^{p-1} \rho(\Delta, \tau.s)}{\Delta \,|\, \Gamma \vdash \pi^D_{p,n}\ e_{(\tau)^D_n} : \tau.p \rightarrowtail [\pi_{k+j,m} e']_{j:1..\rho(\Delta,\tau.p)}} \tag{A.109}$$

Por otro lado tenemos una derivación de la forma

$$\frac{e \Downarrow_o (e')^D_n \quad e.p' \Downarrow_o f}{\pi^D_{p,n} e \Downarrow_o f} \tag{A.110}$$

Aplicando hipótesis inductiva sobre la primer premisa de (A.109) y de (A.110) obtenemos que existe b, tal que

$$e' \Downarrow_d b \tag{A.111}$$
$$\Delta \,|\, \Gamma' \vdash (e')^D_n : (\tau)^D_n \rightarrowtail [b] \tag{A.112}$$

Pero, para probar (A.112) tenemos una derivación de la forma

$$\frac{\Delta \,|\, \Gamma' \vdash e.i' : \tau.i \rightarrowtail [e.i''_j]_{j:1..m_i}}{\Delta \,|\, \Gamma' \vdash (e')^D_n : (\tau)^D_n \rightarrowtail [(e.1''_1, ..., e.1''_{m_1}, ..., e.n''_{m_n})_{m_1+...+m_n}]} \tag{A.113}$$

Aplicando este resultado sobre (A.111) tenemos que

$$b.i = e.t''_j \qquad \text{,donde } i = \sum_{s=1}^{t-1} \rho(\Delta, \tau.s) + j \tag{A.114}$$

Luego, por la p-ésima premisa de (A.113) tenemos

$$\Delta \,|\, \Gamma' \vdash e.p' : \tau.p \rightarrowtail [e.p''_j]_{j:1..m_p}$$

Aplicando (A.114) a este resultado tenemos

$$\Delta \,|\, \Gamma' \vdash e.p' : \tau.p \rightarrowtail [b.(k+j)]_{j:1..m_p}$$

Aplicando la hipótesis inductiva sobre este resultado y la segunda premisa de (A.110), obtenemos que existen b'_j, con $1 \leq j \leq m_p$ tales que

$$b.(k+j) \Downarrow_d b'_j \tag{A.115}$$
$$\Delta \,|\, \Gamma' \vdash f : \tau.p \rightarrowtail [b'_j]_{j:1..m_p} \tag{A.116}$$

Aplicando (SD-PROJ) sobre (A.111) y (A.115), obtenemos

$$\pi_{k+j,m} e \Downarrow_d b'_j$$

Caso (AR-STUPLE)

$$\frac{\Delta \,|\, \Gamma \vdash e.i : \tau.i \rightarrowtail [e.i'_j]_{j:1..m_i}}{\Delta \,|\, \Gamma \vdash (e)^S_n : (\tau)^S_n \rightarrowtail [e.1'_j]_{j:1..m_1} \mathbin{\#} \ldots \mathbin{\#} [e.n'_j]_{j:1..m_n}} \tag{A.117}$$

Por otra parte, tenemos una derivación de la forma

$$\frac{e.i \Downarrow_o f.i}{(e)^S_n \Downarrow_o (f)^S_n} \tag{A.118}$$

Aplicando la hipótesis inductiva sobre la primer premisa de (A.118) y de (A.117) obtenemos que existen $f.i'_j$ tales que

$$\Delta \,|\, \Gamma' \vdash f.i \rightarrowtail [f.i'_j]_{j:1..m_i} \tag{A.119}$$
$$e.i'_j \Downarrow_d f.i'_j \tag{A.120}$$

Finalmente, aplicando (AR-STUPLE) sobre (A.119), tenemos

$$\Delta \,|\, \Gamma \vdash (f)^S_n : (\tau)^S_n \rightarrowtail [f.1'_j]_{j:1..m_1} \mathbin{\#} \ldots \mathbin{\#} [f.n'_j]_{j:1..m_n}$$

Caso (AR-DTUPLE)

$$\frac{\Delta \mid \Gamma \vdash e.i : \tau.i \rightarrowtail [e.i'_j]_{j:1..m_i}}{\Delta \mid \Gamma \vdash (e)^D_n : (\tau)^D_n \rightarrowtail [(e.1'_1, ..., e.1'_{m_1}, ..., e.n'_{m_n})_{m_1+\cdots+m_n}]} \tag{A.121}$$

Por otra parte, tenemos una derivación de la forma

$$\frac{}{(e)^D_n \Downarrow_o (e)^D_n} \tag{A.122}$$

Luego, existe el término $f = (e.1'_1, ..., e.1'_{m_1}, ..., e.n'_{m_n})_{m_1+\cdots+m_n}$, tal que por la conclusión de (A.86)

$$\Delta \mid \Gamma_v \vdash (e)^D_n : (\tau)^D_n \rightarrowtail [f]$$

Y utilizando (SD-TUPLE)

$$(e.1'_1, ..., e.1'_{m_1}, ..., e.n'_{m_n})_{m_1+\cdots+m_n} \Downarrow_d f$$

Caso (AR-LET)

$$\frac{\Delta \mid \Gamma \vdash f : \eta \rightarrowtail [f'_t]_{t:1..n} \quad (\forall \vec{\tau}_{i:1..m} \in \Gamma(v))(\Delta, (\vec{t}, \rho(\Delta, \vec{\tau}_i)) \mid \Gamma'_i \vdash e : \tau \rightarrowtail [e'_i]_{j:1..n_i})}{\begin{array}{l}\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash \texttt{let } v_{\forall \vec{t}.\tau} = \Lambda \vec{t}.e \texttt{ in } f : \eta \rightarrowtail [\texttt{ let } v_{\rho(\Delta,\vec{\tau}_1),1} = e'_{1_1} \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad v_{\rho(\Delta,\vec{\tau}_1),n_1} = e'_{1_{n_1}} \quad ... \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad v_{\rho(\Delta,\vec{\tau}_m),n_m} = e'_{m_{n_m}} \texttt{ in } f'_t \;]_{t:1..n}\end{array}} \tag{A.123}$$

Además tenemos una derivación de la forma

$$\frac{f[v/e] \Downarrow_o f'}{\texttt{let } v_\sigma = e \texttt{ in } f \Downarrow_o f'} \tag{A.124}$$

Denotaremos $\vec{v_i}$ a la secuencia de variables $v_{\rho(\Delta,\vec{\tau}_i),1}, ..., v_{\rho(\Delta,\vec{\tau}_i),m_1}$. Luego, aplicando el lema 4.11 sobre las premisas de (A.123) obtenemos que

$$\Delta \mid \Gamma_v, \Gamma'_i[\vec{t}/\vec{\tau}_i] \vdash f[v/e] : \tau \rightarrowtail [f'_i[\vec{v_1}/\vec{e'_1}]...[\vec{v_m}/\vec{e'_m}]]_{i:1..n}$$

Aplicando la hipótesis inductiva sobre este resultado y la premisa de (A.124) obtenemos que existen f''_t, con $1 \leq t \leq n$, tales que

$$f'_i[\vec{v_1}/\vec{e'_1}]...[\vec{v_m}/\vec{e'_m}] \Downarrow_d f''_t \tag{A.125}$$

$$\Delta \mid \Gamma'' \vdash f' : \eta \rightarrowtail [f''_t]_{t:1..n} \tag{A.126}$$

Finalmente, aplicando (SD-LET) sobre (A.125) concluimos con

$$\begin{array}{l}\texttt{let } v_{\rho(\Delta,\vec{\tau}_1),1} = e'_{1_1} \quad ... \quad \Downarrow_d f''_t \\ \qquad v_{\rho(\Delta,\vec{\tau}_1),n_1} = e'_{1_{n_1}} \quad ... \\ \qquad v_{\rho(\Delta,\vec{\tau}_m),n_m} = e'_{m_{n_m}} \texttt{ in } f'_t\end{array}$$

□

Bibliografía

[Damas and Milner, 1982] Damas, L. and Milner, R. (1982). Principal type schemes for functional programs. *In Proceedings 9'th ACM Symposium on Principles of Programming Languages, Albuquerque, N.M.*

[Hannan and Hicks, 1998] Hannan, J. and Hicks, P. (1998). Higher-order arity raising. *In ICPF'98, Baltimore, MD USA.*

[Hughes, 1996] Hughes, J. (1996). Type specialisation for the lambda-calculus; or, a new paradigm for partial evaluation based on type inference. *In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, Partial Evaluation, volume 1110 of Lecture Notes in Computer Science (LNCS).*

[Jones, 1994] Jones, M. P. (1994). *Qualified Types: Theory and Practice.* Cambridge University Press.

[Jones et al., 1993] Jones, N., Gomard, C., and Sestoft, P. (1993). Partial evaluation and automatic program generation. *Prentice Hall International.* Available online at URL: `http://www.dina.dk/~sestoft/pebook/pebook.html`.

[Jones et al., 1999] Jones, S. P., Hughes, J., et al. (1999). Haskell 98: A non-strict, purely functional language. Technical report, Yale University. Available on-line: `http://www.haskell.org/onlinereport/`.

[Martínez López and Hughes, 2002] Martínez López, P. E. and Hughes, J. (2002). Principal type specialisation. *ASIAN Symposium on Partial Evaluation and Semantics Based Program Manipulations (ASIA- PEPM '02).*

[Milner, 1999] Milner, R. (1999). Communicating and mobile systems: the π-calculus. ISBN 0-521-65869-1, 1999. Cambridge Univeristy Press.

[Milner et al., 1997] Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The definition of Standard ML (revised).* The MIT Press.

[Okasaki, 1998] Okasaki, C. (1998). *Purely Functional Data Structures.* Cambridge University Press, Cambridge, UK.

[Reynolds, 1998] Reynolds, J. C. (1998). *Theory of Programming Languages.* ISBN 0-521-59414-6.

[Romanenko, 1990] Romanenko, S. (1990). Arity raiser and its use in program specialization. In Neil D. Jones, editor, Proc. 3rd European Symposium on Programming

1990, volume 432 of Lecture Notes in Computer Science, pages 341-360, Copenhagen, Denmark, 1990. Springer-Verlag.

[Thiemann, 2000] Thiemann, P. (2000). First-class polyvariant functions and co-arity raising. Unpublished manuscript. URL: `www.informatik.uni-freiburg.de/~thiemann/papers/`.

[Wadler, 1993] Wadler, P. (1993). Monads for functional programming. In Broy, M., editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag.