TRABAJO ESPECIAL

LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

HACIA UN ASISTENTE DE DEMOSTRACIÓN BASADO EN PTS

Alejandro Sánchez
Director: Daniel Fridlender

20 de Diciembre, 2007

Facultad de Matemática, Astronomía y Física

Universidad Nacional de Córdoba
Argentina

Abstract

Los asistentes de demostraciones han demostrado ser valiosas herramientas en el campo de la teoría matemática y computacional, permitiendo especificar, verificar y formalizar teoremas y, en particular, propiedades de programas. En el presente trabajo nos abocamos a la tarea de desarrollar un prototipo de asistente de demostración parametrizable en el sistema de tipos. Para ello, primero estudiamos los *Sistemas de Tipos Puros*, una generalización de varios de los sistemas de tipos utilizados en el cálculo lambda. Luego, damos una idea de cómo empleando el isomorfismo de *Curry-Howard* resulta posible modelar pruebas de programas a través de expresiones lambda. Finalmente, formalizamos un prototipo de asistente basado en los principios propuestos por dicha correspondencia, concluyendo con una implementación en *Haskell* del mismo.

Palabras claves: Type Theory, Intuitionistic Logic, Sistemas de Tipos, Cálculo Lambda, Prueba de Programas

F.4.1 Mathematical Logic

Agradecimientos

Quisiera comenzar por agradecer a mi familia, ya que sin su esfuerzo nada de esto hubiera sido posible. En especial a Silvina, quien se preocupó por que pudiera destinar cada uno de mis minutos libres a la conclusión de este trabajo.

También quisiera reconocer el aporte efectuado por compañeros y amigos, el cuál sin duda va mas allá de toda sugerencia o ayuda aportada, ya que sin ellos, éstos últimos años hubieran sido mucho menos entretenidos :D.

Finalmente, agradezco a Miguel Pagano y Héctor "El Flaco" Gramaglia, por las ideas, comentarios y correcciones que ayudaron a mejorar este trabajo y en especial a Daniel "Frito" Fridlender, quien además supo aconsejarme, guiarme y alentarme durante este último año y medio de trabajo.

A todas y cada una de las personas que hicieron posible que este día finalmente llegara, MUCHAS GRACIAS.

Alejandro.

Tabla de Contenidos

Indice de figuras

Capítulo 1

Introducción

Los asistentes de demostraciones son herramientas valiosas en el campo de la teoría matemática y computacional, permitiendo a desarrolladores especificar, verificar y formalizar teoremas y, en particular, propiedades de programas. Incluso en los últimos años, herramientas de este tipo han permitido numerosos avances de la computación teórica. Una de las maneras que existen para llevar a cabo tales demostraciones de proposiciones lógicas es empleando la teoría de tipos [CH88, ML84, Wil99].

En la actualidad ya existen algunas herramientas similares que han probado ser de gran utilidad en varias ramas de investigación teórica, entre las cuales se destacan *COQ*, *Epigram*, *Cayenne* o *Agda* [Coq99, Epi04, Cay99, Agd00]. Si bien estas aplicaciones se encuentran basadas sobre principios similares a los empleados para llevar a cabo nuestro prototipo, es de destacar que el nivel de maduración que presentan tales asistentes sobrepasa en gran medida el alcance propuesto para una tesis de grado.

Bajo este contexto, la *Teoría de Tipos Intuicionista* [ML84] nos permite trabajar sobre pruebas constructivas que emplean proposiciones, ya que está basada en una correspondencia entre proposiciones y tipos, donde una proposición se encuentra definida por el tipo de una potencial prueba de ella y una prueba no es mas que un término de dicho tipo. De esta manera, lo que se intenta hacer es lograr una prueba donde se mantiene la justificación de la derivación de una proposición, más que el valor de verdad que posea la misma. Es decir que una fórmula en esta *Lógica Intuicionista* es considerada verdadera solamente si es posible construir una prueba de la misma.

Esta correspondencia entre proposiciones y tipos se conoce como el Isomorfismo (o correspondencia) de *Curry-Howard* [CF58, How80]. Ésta establece la existencia de una analogía entre un teorema y el tipo de un valor obtenido al computar una función, siendo además el programa empleado en tal computación una prueba de dicho teorema. Teniendo esto en mente, las pruebas buscadas ahora pasan a ser programas.

Para todo esto, son de vital importancia los Sistemas de Tipos Puros (*Pure Type Systems*) [Wil99, Ber88, Ber90], que pueden ser vistos como una generalización de los lenguajes funcionales tipados, ya que son una base ideal para representar proposiciones como tipos. Esto viene del hecho de que cada *PTS* es un sistema formal a partir del cual se pueden derivar juicios que permiten asignar un tipo a un término específico del cálculo lambda, bajo un contexto dado, hecho de vital importancia al trabajar bajo el isomorfismo de *Curry-Howard*.

De esta manera, combinando la idea proporcionada por la *Correspondencia de*

Curry-Howard con el poder expresivo de los *Sistemas de Tipos Puros*, es posible el desarrollo de pruebas proposicionales complejas que incluyen además de operadores básicos como conjunción, disyunción e implicación, algunos más complejos tales como cuantificadores existenciales y universales, entre otros. Los operadores lógicos conviven de esta manera en un sistema de tipos que contiene estructuras de datos expresivas (tuplas, listas, tipos numéricos, tipos dependientes, etc.).

Nuestro aporte

En el trabajo que presentamos, exponemos un prototipo de asistente de demostraciones que cuenta con la virtud de ser completamente parametrizable respecto al sistema de tipos sobre el cual se este trabajando. El mismo permite efectuar pruebas de teoremas basados en afirmaciones proposicionales, asistiendo al usuario en cada una de las decisiones tomadas que conduzcan al desarrollo de dicha prueba.

Una de las virtudes con que cuenta esta herramienta es la posibilidad de permitirle al usuario trabajar con múltiples teoremas que se encuentren basados en teorías de tipos no necesariamente iguales, ya que cada demostración cuenta en todo momento con su propia traza de demostración, lo cual nos posibilita experimentar con diferentes sistemas de tipos, el cual era una de los objetivos principales de este trabajo. Incluso, si bien existen numerosas familias de sistemas de tipos que conviven bajo la clasificación de Sistemas de Tipos Puros (*PTS*), el hecho de que los comandos o tácticas implementadas que van asistiendo al usuario en la construcción de un termino a partir de su tipo se mantuvieron en el marco más general en que fuere posible, queda garantizada la reusabilidad de esta aplicación para derivaciones bajo cualquier clase de PTS.

Teniendo en mente que nuestra contribución sentaría las bases para el posible desarrollo a largo plazo de un asistente mucho mas avanzado y completo, durante toda la etapa de desarrollo se puso especial énfasis en garantizar que todo el código generado pueda ser reutilizado o extendido con completa facilidad, ofreciendo ya en el nivel de desarrollo actual numerosas abstracciones e interfaces de funcionalidades que harán mas amena la tarea de futuros desarrolladores, llegando incluso a producir documentos de referencia que sirva de guía a los futuros desarrolladores. Todas estas características se verán en mayor detalle en capítulos posteriores.

En relación a la madurez de desarrollo que se ha logrado con esta herramienta, cabe destacar que la contribución que estamos haciendo en virtud de la elaboración de una aplicación propia, puede ser vista como una segunda etapa en el afán de alcanzar tal objetivo. Esto se debe a que ya en una primera etapa, Fabian Fleitas en su trabajo final de carrera de grado [Fle06] presentó ya un chequeador de tipos (*typechecker*) basado en principios similares a los expuestos en el presente trabajo.

Estructura del trabajo

A lo largo de este informe comentaremos las distintas etapas y complicaciones que debimos enfrentar durante el desarrollo de la herramienta, como así también expondremos las soluciones hayadas para dichos inconvenientes. Para lograr tal cometido, en un primer momento explicaremos en detalle el marco teórico en el cual se ubica este trabajo. Luego, habiendo ya establecido la base teórica sobre la cual se sustenta nuestro trabajo, nos abocaremos ya si a brindar un análisis mas detallado y exiguo de la aplicación que desarrollamos, dando a conocer los detalles de cada una de las etapas que

se vieron involucradas en el proceso que derivó en la obtención de nuestra herramienta. Más precisamente, los temas desarrollados en los próximos capítulos incluirán:

Capítulo 2

Damos una visión general de sistemas de tipos simples. Luego nos abocamos al estudio de sistemas de tipos mas complejos, mostrando un incremento gradual en la complejidad de los mismos. Finalmente exponemos el concepto de Sistema de Tipo Puro (*PTS*) y enunciamos algunas propiedades de los mismos. Para cada sistema de tipos detallamos las reglas de tipado que los rigen.

Capítulo 3

Exponemos un resumén acerca de los principios sobre los cuales se maneja la lógica y la teoría de tipos intuicionista. Explicaremos de que se tratan los tipos dependientes y su estrecha relación con la idea de *PTS* definida en el capítulo anterior. Por último enunciaremos el isomorfismo de *Curry-Howard* junto a algunos ejemplos y posibles maneras de implementarlo.

Capítulo 4

Comentamos el esfuerzo que implica desarrollar un asistente como el elaborado. Emprendemos la tarea de definir estructuras y algoritmos que nos permita manipular los sistemas de tipos en que estamos interesados. Formalizamos los bloques principales sobre los cuales se sustenta nuestra herramienta. También planteamos los inconvenientes a que nos enfrentamos al intentar llevar a cabo tareas mas complejas como los son la generación de premisas, el manejo de incógnitas, y la determinación de restricciones para las derivaciones entre otras. Además damos a conocer las soluciones encontradas para cada uno de los problemas expuestos.

Capítulo 5

Damos una idea acabada de la implementación de nuestra herramienta. Así mismo brindamos una descripción detallada de los módulos que la componen y la intercomunicación entre los mismos. También mostramos como todo el desarrollo logrado en el capítulo 4 se traduce a una implementación concreta. Finalmente mostramos la interfaz de nuestra aplicación y damos a conocer algunas de las funcionalidades que ésta ofrece.

Capítulo 6

Reflexionamos sobre las conclusiones obtenidas en el desarrollo del presente trabajo. Así mismo sugerimos algunas ideas de como mejorar la herramienta y planteamos algunas tareas de trabajo a futuro.

Capítulo 2

Sistemas de tipos: de $\lambda\!\rightarrow$ a los *PTS*

Una de las ideas principales sobre la cual se sustenta nuestro trabajo es la de los Sistemas de Tipos Puros. En este capítulo centraremos nuestra atención en el estudio de algunos sistemas de tipos sencillos, para luego ir añadiendo nuevos conceptos que finalmente nos acerquen al análisis de los sistemas de tipos más sofisticados en los cuales estamos verdaderamente interesados.

A medida que avancemos a través de los distintos sistemas de tipos iremos dando a conocer las reglas de tipado que involucra a cada uno de ellos. De esa manera esperamos resulte más comprensible el conjunto de reglas de los *PTS* a las cuales arribaremos y que desde luego serán de gran importancia en razonamientos que haremos en capítulos posteriores.

2.1 Comprendiendo sistemas de tipos simples

Para embarcarnos definitivamente en el estudio de los *PTS*, primero comentaremos algunos de los sistemas de tipos más simples, en el afán de que partiendo de los mismos resulte más sencillo comprender los sistemas que son de nuestro interés. Daremos sólo una idea general de estos sistemas, dado que nuestro interés se centra en los sistemas más avanzados que veremos posteriormente. Una descripción mas detallada puede encontrarse en [Bar92].

2.1.1 *a la Curry* vs. *a la Church*

Antes de comenzar, debemos aclarar que todos los sistemas de tipos expuestos de aquí en adelante, como así también toda expresión del calculo lambda que empleemos se llevará a cabo en la notación conocida como *a la Church*. A quien ya ha efectuado un curso de programación funcional o se ha topado en algún momento con el cálculo lambda, no le resultará extraño encontrarse con expresiones de la forma:

$$\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,xz(yz)$$

expresiones de este tipo tienen la peculiaridad de que no llevan explícitamente detallado el tipo de las variables en las abstracciones utilizadas. Esto se debe a que en la mayoría de los sistemas de tipos simples es posible inferir el tipo de las mismas. Cuando el tipo de dichas variables no se da en forma explícita dentro de la misma expresión, decimos que se encuentra *a la Curry*[1].

La notación en que el tipo de las variables en las abstracciones se da a conocer de manera explícita, es conocida como *a la Church*. Usando esta notación, la misma expresión que mostramos arriba queda expresada de la forma:

$$\lambda x : \alpha \to \beta \to \gamma \,.\, \lambda y : \alpha \to \beta \,.\, \lambda z : \alpha \,.\, xz(yz)$$

A lo largo del presente trabajo emplearemos la notación *a la Church*. Uno de los motivos que lleva a tal elección radica en el hecho de que al efectuar el chequeo de tipos de una aplicación, por ejemplo, sabemos el tipo de la expresión resultante, pero deberíamos en cierta forma, inferir el tipo del argumento al cual esta siendo aplicado la función. Pero esta inferencia se torna innecesaria si se emplea la notación *a la Church* como consecuencia de que también conocemos el tipo del elemento al cual se le esta aplicando una función, e incluso el tipo de la función misma. Esto no significa que el problema de inferir los tipos en una notación se simplifique más que en la otra, pero al menos el problema del chequeo de tipos se ve facilitado.

De todas formas la elección de la notación *a la Curry* dificultaría nuestro trabajo ya que nos privaría de vital información. Esto se debe primordialmente a que, como veremos más adelante, intentaremos hallar pruebas para ciertas proposiciones basandonos en el hecho de que las proposiciones pueden ser vistas como tipos mientras que las pruebas serían términos de dicho tipo. Dicho de otro modo, estaríamos en la situación en la que tenemos una prueba y ¡desviaríamos nuestra atención hacia la busqueda de una proposición que sea satisfecha por dicha prueba! Claramente, nuestra situación es la opuesta, de manera tal que en todo momento estaremos interesados en el tipo de cada uno de los términos que estemos manipulando.

Si prestamos atención a la notación *a la Church*, a primera vista es mucho más engorrosa que nuestra primera versión, pero la información dada por el tipo de cada una de las variables que ocurren en la expresión otorga valiosa información en la tarea de identificar el tipo de la misma. Además no sólo se da el hecho de que fácilmente se puede determinar el tipo de un término dado, sino que además determina la unicidad en el tipado de dicho término para muchos de los sistemas que veremos más adelante, algo que no ocurre en la versión *a la curry*, donde por ejemplo tenemos expresiones polimórficas como:

$$\lambda x \,.\, x \;:\; \alpha \to \alpha$$

$$\lambda x \,.\, x \;:\; (\alpha \to \alpha) \to (\alpha \to \alpha)$$

Si bien conocer el tipo de cada variable nos va a ser de gran ayuda más adelante en nuestro trabajo, esto no es suficiente para obtener una solución a algunos de los problemas más comunes sobre los sistemas de tipos que resultan de interés para los lenguajes de programación funcionales. Por ejemplo, aún con el tipado explicito, el problema de la inferencia de tipos para sistemas más complejos puede llegar a ser no decidible. A medida que avancemos se irá haciendo evidente la necesidad de dicha conveniencia.

[1] Empleada comúnmente por el matemático y lógico americano Haskell Brooks Curry.

Para todos los sistemas de tipos que veremos es posible definir los problemas conocidos como chequeo, inferencia y correctitud de tipos, de la siguiente manera:

Definición 2.1 (Type Checking, Type Inference y Type Correctness).

- El problema de *Type Checking* consiste en determinar si, para dados un contexto Γ, un término M y un tipo α, se satisface que $\Gamma \vdash M : \alpha$.
- El problema de *Type Inference* consiste en encontrar, dados un contexto Γ y un término M, un tipo α tal que se satisfaga que $\Gamma \vdash M : \alpha$.
- El problema de *Type Correctness* consiste en decidir si, dados un contexto Γ y un término M, existe un tipo α tal que se satisfaga que $\Gamma \vdash M : \alpha$. †

Cuando tenemos que $\Gamma \vdash M : \sigma$ diremos que M *tiene tipo* σ en Γ y diremos que M es *tipable* si y sólo si existen Γ y σ tales que $\Gamma \vdash M : \sigma$.

Para cada sistema de tipos a estudiar, diremos cómo se encuentran definidos sus tipos, términos, contextos, reglas de tipado y algunas propiedades interesantes que puedan presentar. Para denotar los elementos que forman parte de cada uno de los sistemas de tipos, emplearemos la siguiente notación.

Notación 2.1 (tipos):

- Usaremos α, β, γ,... para denotar *variables de tipos* arbitrarias.
- Usaremos σ, τ,... para denotar *tipos* arbitrarios.
- Siempre que usemos $\rightarrow$, diremos que la misma asocia a derecha. $\diamondsuit$

A continuación analizaremos algunos sistemas de tipos simples con la notación elegida: *a la Church*.

2.1.2 El sistema de tipos $\lambda\rightarrow$

Emprenderemos nuestro estudio de los sistemas de tipos partiendo de uno de los sistemas más simples que existen. Sólo nos limitaremos a denotar la forma en que se encuentran definidos los tipos, términos y contextos válidos, junto a sus reglas de tipado y ejemplos que nos ayuden a comprender la ubicación que ocupa este sistema de tipos dentro de la generalización de sistemas, la cual estudiaremos posteriormente. Por más detalles sobre el sistema $\lambda\rightarrow$ se puede consultar [Chu40], [Bar81] o [HS86].

Definición 2.2 (tipos).
Sea U un conjunto infinito numerable de *variables de tipos*. El conjunto Π de *tipos* de $\lambda\rightarrow$ está definido por la siguiente gramática:

$$\Pi ::= U \mid \Pi \rightarrow \Pi$$

donde $\rightarrow$ asocia a derecha. Ésto permite no sólo nos permite escribir tipos tales como U, $U \rightarrow U$, $U \rightarrow U \rightarrow U$,..., sino que también $(U \rightarrow U) \rightarrow (U \rightarrow U)$ y así sucesivamente. †

Definición 2.3 (términos).
Sea V un conjunto numerable de variables. El conjunto Δ de términos del sistema $\lambda\!\rightarrow$ está definido por la siguiente gramática:

$$\Delta ::= V \mid \lambda V : \Pi . \Delta \mid \Delta\Delta \qquad \dagger$$

Estableceremos como convención que la aplicación de términos asocia a izquierda, teniendo ésta mayor precedencia que la abstracción.

Notación 2.2 (términos):

- Usaremos x, y,... para denotar variables de términos arbitrarias.
- Usaremos M, N,... para denotar términos arbitrarios. $\diamondsuit$

Definición 2.4 (contextos).
El conjunto C de *contextos* es el conjunto de todos los conjuntos de pares de la forma

$$x_1 : \tau_1 , \ldots , x_n : \tau_n$$

donde

$$\tau_1, \ldots, \tau_n \in \Pi$$

$$x_1, \ldots, x_n \in V$$

y además $\forall i, j \in \{1, \ldots, n\} : i \neq j$ se satisface que $x_i \neq x_j$ $\dagger$

Notación 2.3:
Dado que por el momento los contextos son tratados como conjuntos, denotaremos al contexto vacío por $\{\}$. Además emplearemos Γ, Δ,... para denotar contextos arbitrarios. $\diamondsuit$

Definición 2.5 (dominio de contextos).
Dado un contexto $\Gamma = \{x_1 : \tau_1 , \ldots , x_n : \tau_n\}$, definimos el dominio de dicho contexto por:

$$dom\,\Gamma = \{x_1 , \ldots , x_n\} \qquad \dagger$$

Definición 2.6 (reglas de tipado).
La relación de tipado $\vdash$ definida sobre $C \times \Delta \times \Pi$ se encuentra dada por las reglas de tipado detalladas en la figura 2.1, donde $\Gamma , x : \sigma$ significa $\Gamma \cup \{x : \sigma\}$.

$$\text{(start)} \qquad \frac{}{\Gamma \vdash x : \sigma} \qquad (x:\sigma) \in \Gamma$$

$$\text{($\rightarrow$ introduction)} \qquad \frac{\Gamma, x:\sigma \vdash M : \tau}{\Gamma \vdash \lambda x:\sigma . M : \sigma \rightarrow \tau} \qquad x \notin dom(\Gamma)$$

$$\text{($\rightarrow$ elimination)} \qquad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

Figura 2.1: Reglas de tipado de $\lambda\!\rightarrow$

La primera regla establece que si $(x:\sigma)$ ocurre en un dominio Γ, entonces podemos decir que x tiene tipo σ en Γ.

La regla de *introduction* nos dice que si en un contexto conocemos que x tiene tipo σ y que usando esa información pudimos deducir que M tiene tipo τ, entonces podemos deducir que el término $\lambda x:\sigma . M$ tiene tipo $\sigma \rightarrow \tau$.

La regla de *elimination* en tanto nos dice que si bajo un contexto Γ pudimos deducir que M es una función de tipo $\sigma \rightarrow \tau$ y que el término N tiene tipo σ, entonces la aplicación de MN tiene tipo τ.

De esta manera, el conjunto de todos los términos tipables resulta ser un subconjunto propio de todos los términos, en donde se establecen restricciones sobre qué términos pueden ser aplicados a qué otros términos. Como el tipo $\sigma \rightarrow \tau$ denota el conjunto de funciones que van de σ a τ, si tenemos que $M : \sigma \rightarrow \tau$ entonces M puede ser aplicada solamente a un término de tipo σ.

Ejemplo 2.1 (La función identidad):
La función identidad está dada por:

$$I = \lambda x:\sigma . x : \sigma \rightarrow \sigma$$

siendo su prueba en el sistema $\lambda\!\rightarrow$:

$$\dfrac{\dfrac{}{x:\sigma \vdash x : \sigma}\ \text{(start)}}{\vdash \lambda x:\sigma . x : \sigma \rightarrow \sigma}\ \text{($\rightarrow$ introduction)}$$

*

Evidentemente, en realidad existe una cantidad infiníta de funciones de idéntidad: una por cada tipo σ.

Ejemplo 2.2 (La constante K):
La constante K está dada por:

$$K = \lambda x:\sigma . \lambda y:\tau . x : \sigma \rightarrow \tau \rightarrow \sigma$$

siendo su prueba en el sistema $\lambda\!\rightarrow$:

$$\dfrac{\dfrac{\dfrac{}{x:\sigma, y:\tau \vdash x : \sigma}\ \text{(start)}}{x:\sigma \vdash \lambda y:\tau . x : \tau \rightarrow \sigma}\ \text{($\rightarrow$ introduction)}}{\vdash \lambda x:\sigma . \lambda y:\tau . x : \sigma \rightarrow \tau \rightarrow \sigma}\ \text{($\rightarrow$ introduction)}$$

*

Para simplificar un poco la notación empleada en las pruebas, a menudo emplearemos:

- S, para denotar la regla de start
- $\rightarrow$ I, para denotar la regla de $\rightarrow$ introduction
- $\rightarrow$ E, para denotar la regla de $\rightarrow$ elimination

Ejemplo 2.3 (La constante *S*):
La constante *S* está dada por:

$$S = \lambda x : \sigma \rightarrow \tau \rightarrow \rho \,.\, \lambda y : \sigma \rightarrow \tau \,.\, \lambda z : \sigma \,.\, xz(yz) \;:\; (\sigma \rightarrow \tau \rightarrow \rho) \rightarrow (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \rho$$

siendo su prueba en el sistema $\lambda\!\rightarrow$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash x \;:\; \sigma \rightarrow \tau \rightarrow \rho}\,(\mathrm{S}) \quad \dfrac{}{\Gamma' \vdash z \;:\; \sigma}\,(\mathrm{S})}{\Gamma' \vdash xz \;:\; \tau \rightarrow \rho}\,(\rightarrow \mathrm{E}) \quad \dfrac{\dfrac{}{\Gamma' \vdash y \;:\; \sigma \rightarrow \tau}\,(\mathrm{S}) \quad \dfrac{}{\Gamma' \vdash z \;:\; \sigma}\,(\mathrm{S})}{\Gamma' \vdash yz \;:\; \tau}\,(\rightarrow \mathrm{E})}{x : \sigma \rightarrow \tau \rightarrow \rho \,,\, y : \sigma \rightarrow \tau \,,\, z : \sigma \;\vdash\; xz(yz) \;:\; \rho}\,(\rightarrow \mathrm{E})}{x : \sigma \rightarrow \tau \rightarrow \rho \,,\, y : \sigma \rightarrow \tau \;\vdash\; \lambda z : \sigma \,.\, xz(yz) \;:\; \sigma \rightarrow \rho}\,(\rightarrow \mathrm{I})}{x : \sigma \rightarrow \tau \rightarrow \rho \;\vdash\; \lambda y : \sigma \rightarrow \tau \,.\, \lambda z : \sigma \,.\, xz(yz) \;:\; (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \rho}\,(\rightarrow \mathrm{I})}{\vdash \lambda x : \sigma \rightarrow \tau \rightarrow \rho \,.\, \lambda y : \sigma \rightarrow \tau \,.\, \lambda z : \sigma \,.\, xz(yz) \;:\; (\sigma \rightarrow \tau \rightarrow \rho) \rightarrow (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \rho}\,(\rightarrow \mathrm{I})$$

donde $\Gamma' = x : \sigma \rightarrow \tau \rightarrow \rho \,,\, y : \sigma \rightarrow \tau \,,\, z : \sigma$. $*$

2.1.3 El sistema de tipos $\lambda 2$

Si repasamos los argumentos esgrimidos en la sección 2.1.1 sobre las diferencias presentadas entre la notación *a la Church* y la versión *a la Curry*, recordaremos que en esta última era factible la escritura de funciones polimórficas de la forma $\lambda x.x$, mientras que en la primera, dado que resultaba necesario explicitar el tipo de los parámetros de las funciones, escribir una función polimórfica resultaba imposible.

En esta sección nos embarcaremos en el estudio de un nuevo sistema de tipos. El sistema $\lambda 2$, también denominado cálculo lambda polimórfico de segundo orden, fue desarrollado de manera independiente por el lógico francés Jean-Yves Girard [Gir72] y el investigador americano John C. Reynolds [Rey74] y brinda la posibilidad de definir funciones polimórficas, lo que nos permitirá especificar funciones tales como la identidad polimórfica, algo que hasta ahora nos resultaba imposible en la notación *a la Church*.

Definición 2.7 (tipos).
Sea U el conjunto de las variables de tipo. El conjunto Π de los tipos del sistema $\lambda 2$ está definido por la siguiente gramática:

$$\Pi \;::=\; U \mid \Pi \rightarrow \Pi \mid \forall U \Pi \qquad \dagger$$

Es esta nueva cláusula $\forall U \Pi$ la que permite generar los tipos de las funciones polimórficas.

Definición 2.8 (términos).
Sea V un conjunto numerable de variables. El conjunto Δ de términos del sistema $\lambda 2$ se encuentra definido por la siguiente gramática:

$$\Delta ::= V \mid \lambda V : \Pi . \Delta \mid \Delta\Delta \mid \Lambda U . \Delta \mid \Delta\Pi \qquad \dagger$$

Comparando los términos válidos para el sistema $\lambda 2$ en relación con los de $\lambda \rightarrow$, notamos que aparecen dos nuevas cláusulas.

La primera ($\Lambda U . \Delta$), la cual denota la *abstracción polimórfica*, genera términos de la forma $\Lambda\alpha . M$. A este nuevo término lo podemos interpretar como que el término M (que puede contener dentro de sí mismo α ubicada en los lugares en donde variables de tipos puedan ocurrir) resulta ser una función polimórfica con un parámetro de tipo α.

La segunda cláusula ($\Delta\Pi$), que define la *aplicación de tipos*, permite crear términos de la forma $(M\tau)$, donde M es un término y τ es un tipo. Estos nuevos términos son interpretados como una llamada a una función genérica M que recibe como parámetro un tipo. De esta manera, al aplicar M al tipo τ, estamos creando una instancia de función polimórfica. Resulta entoncés obvio por qué decimos que este sistema de tipos es polimórfico. Además decimos que es de segundo orden porque tiene tipos de la forma $\forall \alpha . T$.

Definición 2.9 (contextos).
El conjunto C de *contextos* se encuentra definido de manera idéntica al conjunto de contextos que dimos para el sistema $\lambda \rightarrow$ en la definición 2.4, aunque la misma definición nos da lugar a construir más contextos, ya que hay más tipos disponibles. $\dagger$

Usando las nuevas definiciones que tenemos para Δ y Π bajo el sistema $\lambda 2$, tenemos las siguientes reglas de tipado para el sistema $\lambda 2$.

Definición 2.10 (reglas de tipado).
La relación de tipado $\vdash$ definida sobre $C \times \Delta \times \Pi$ se encuentra dada por las reglas de la figura 2.2:
donde la restricción de que $\alpha \notin FV(\Gamma)$ se corresponde con la necesidad de que la variable de tipo que esta siendo abstraída debe ser un identificador local. $\dagger$

Como se puede apreciar, las primeras tres reglas coinciden con las definidas para el sistema $\lambda \rightarrow$.

La regla de ($\forall$ introduction), generaliza al término M. Es decir, dado un término arbitrario, forma una nueva función polimórfica que recibe una variable de tipo como argumento, permitiendo así generalizar las ocurrencias del tipo α que ocurran en M.

Por otro lado, la regla de ($\forall$ elimination) permite instanciar una función polimórfica. Es decir, suponiendo que M sea una función que recibe un tipo como parámetro, indica que M aplicada a un tipo τ pasa a ser un término de tipo σ donde todas las ocurrencias de α han sido sustituidas por el tipo al cual ha sido aplicado M.

$$\text{(start)} \quad \frac{}{\Gamma \vdash x : \tau} \quad (x:\tau) \in \Gamma$$

$$\text{($\to$ introduction)} \quad \frac{\Gamma, x:\sigma \vdash M : \tau}{\Gamma \vdash \lambda x:\sigma . M : \sigma \to \tau} \quad x \notin dom(\Gamma)$$

$$\text{($\to$ elimination)} \quad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$\text{($\forall$ introduction)} \quad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha . M : \forall\alpha . \sigma} \quad \alpha \notin FV(\Gamma)$$

$$\text{($\forall$ elimination)} \quad \frac{\Gamma \vdash M : \forall\alpha . \sigma}{\Gamma \vdash M\tau : \sigma[\alpha := \tau]} \quad \tau \in \Pi$$

Figura 2.2: Reglas de tipado de $\lambda 2$

Ejemplo 2.4 (La función identidad polimórfica):
La función identidad polimórfica está dada por:

$$I_p = \Lambda\alpha . \lambda x:\alpha . x : \forall\alpha . \alpha \to \alpha$$

siendo su prueba en el sistema $\lambda 2$:

$$\dfrac{\dfrac{\dfrac{}{x:\alpha \vdash x : \alpha}\ \text{(start)}}{\vdash \lambda x:\alpha . x : \alpha \to \alpha}\ (\to \text{introduction})}{\vdash \Lambda\alpha . \lambda x:\alpha . x : \forall\alpha . \alpha \to \alpha}\ (\forall \text{ introduction})$$

*

Ejemplo 2.5 (Aplicación de la función identidad polimórfica):
Si ahora tomamos la función de identidad polimórfica I_p y la aplicamos a un tipo arbitrario σ, obtenemos la función identidad para términos de ese tipo. Es decir que:

$$I_p\sigma : \sigma \to \sigma$$

siendo su prueba en el sistema $\lambda 2$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\alpha \vdash x : \alpha}\ \text{(start)}}{\vdash \lambda x:\alpha . x : \alpha \to \alpha}\ (\to \text{introduction})}{\vdash \Lambda\alpha . \lambda x:\alpha . x : \forall\alpha . \alpha \to \alpha}\ (\forall \text{ introduction})}{\vdash (\Lambda\alpha . \lambda x:\alpha . x)\sigma : \sigma \to \sigma}\ (\forall \text{ elimination})$$

*

Ejemplo 2.6 (La función de composición polimórfica):
La función de composición polimórfica se define como:

$$\begin{array}{rcl} \circ_p & = & \Lambda\alpha\,.\,\Lambda\beta\,.\,\Lambda\gamma\,.\,\lambda f:\alpha\to\beta\,.\,\lambda g:\beta\to\gamma\,.\,\lambda x:\alpha\,.\,g(fx)\ : \\ & & \forall\alpha\,.\,\forall\beta\,.\,\forall\gamma\,.\,(\alpha\to\beta)\to(\beta\to\gamma)\to\alpha\to\gamma \end{array}$$

digamos que

$$\begin{array}{rcl} M & = & \lambda f:\alpha\to\beta\,.\,\lambda g:\beta\to\gamma\,.\,\lambda x:\alpha\,.\,g(fx) \\ \sigma & = & (\alpha\to\beta)\to(\beta\to\gamma)\to\alpha\to\gamma \\ \Gamma' & = & f:\alpha\to\beta\,,\,g:\beta\to\gamma\,,\,x:\alpha \end{array}$$

luego la demostración de $\circ_p$ en $\lambda 2$ queda como:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'\vdash f\ :\ \alpha\to\beta}\,(\mathrm{S})\quad\dfrac{}{\Gamma'\vdash x\ :\ \alpha}\,(\mathrm{S})}{\Gamma'\vdash fx\ :\ \beta}\,(\to\mathrm{E})\quad\dfrac{}{\Gamma'\vdash g\ :\ \beta\to\gamma}\,(\mathrm{S})}{f:\alpha\to\beta\,,\,g:\beta\to\gamma,x:\alpha\vdash g(fx)\ :\ \gamma}\,(\to\mathrm{E})}{f:\alpha\to\beta\,,\,g:\beta\to\gamma\vdash\lambda x:\alpha\,.\,g(fx)\ :\ \alpha\to\gamma}\,(\to\mathrm{I})}{f:\alpha\to\beta\vdash\lambda g:\beta\to\gamma\,.\,\lambda x:\alpha\,.\,g(fx)\ :\ (\beta\to\gamma)\to\alpha\to\gamma}\,(\to\mathrm{I})}{\vdash\lambda f:\alpha\to\beta\,.\,\lambda g:\beta\to\gamma\,.\,\lambda x:\alpha\,.\,g(fx)\ :\ (\alpha\to\beta)\to(\beta\to\gamma)\to\alpha\to\gamma}\,(\to\mathrm{I})}{\vdash\Lambda\gamma\,.\,M\ :\ \forall\gamma\,.\,\sigma}\,(\forall\mathrm{I})}{\vdash\Lambda\beta\,.\,\Lambda\gamma\,.\,M\ :\ \forall\beta\,.\,\forall\gamma\,.\,\sigma}\,(\forall\mathrm{I})}{\vdash\Lambda\alpha\,.\,\Lambda\beta\,.\,\Lambda\gamma\,.\,M\ :\ \forall\alpha\,.\,\forall\beta\,.\,\forall\gamma\,.\,\sigma}\,(\forall\mathrm{I})$$

donde claramente, al aplicar $\circ_p$ a tres tipos cualesquiera, obtenemos la instanciación de la función de composición para los tipos que pasamos como parámetro. $*$

2.1.4 El sistema de tipos $\lambda\omega$

A esta altura se comienzan a manifestar algunas dependencias entre tipos y términos. Estas dependencias pueden clasificarse en:

- términos que dependen de términos
- términos que dependen de tipos
- tipos que dependen de tipos
- tipos que dependen de términos

La primer clase de dependencia ya fue vista cuando definimos el sistema $\lambda\!\to$. De hecho, en dicho sistema vimos que:

$$\frac{\Gamma\vdash F\ :\ A\to B\qquad\Gamma\vdash M\ :\ A}{\Gamma\vdash F\,M\ :\ B}$$

En tal caso, $F\ M$ es un término que depende de otro término (M en nuestro ejemplo).

En el caso de la segunda clase de dependencia, la de términos que dependen de tipos, en el sistema $\lambda 2$ podemos verificar que:

$$\frac{\Gamma \vdash G : \forall\alpha\,.\,\alpha \to \alpha}{\Gamma \vdash G\,A : A \to A}\ A \in \Pi \quad \text{(i.e. A es un tipo)}$$

Por lo tanto, si $G = \Lambda\alpha\,.\,\lambda a : \alpha\,.\,a$, se tiene que $G\,A$ es un término que depende de un tipo (A en nuestro caso).

En síntesis, para los sistemas $\lambda\!\to$ y $\lambda 2$ contamos con mecanismos de abstracción que permiten estos dos tipos de dependencias. Para los dos primeros tipos de dependencias podemos dar como ejemplo:

$$\begin{array}{ll} \lambda m : A\,.\,F\ m\ :\ A \to B & \text{en } \lambda\!\to \\ \Lambda\alpha\,.\,G\ \alpha\ :\ \forall\alpha\,.\,\alpha \to \alpha & \text{en } \lambda 2 \end{array}$$

Para poder ver la tercer clase de dependencia, ahora definiremos el sistema de tipos $\lambda\omega$ (*lambda omega*), introducido por Jean-Yves Girard [Gir72], el cual permite tipos que dependen de tipos. Para ello es necesario tener funciones que aplicadas a un tipo devuelven también un tipo. Para ello se utiliza el propio cálculo lambda en el nivel de los tipos, además de (tal y como ocurría anteriormente) en el de los términos. En los sistema de tipos anteriores, los tipos se generaban a partir de una gramática externa al sistema, mientras que en el $\lambda\omega$ el conjunto de tipos comienza a estar definido por las reglas de tipado del sistema mismo.

Para lograr esto, definimos una nueva constante $\star$. Diremos entonces que $\sigma : \star$ cuando se satisface que $\sigma \in \Pi$. Si recordamos las definiciones de tipos dadas para los sistemas $\lambda\!\to$ y $\lambda 2$, veremos que la gramática empleada para construir los mismo, nos permitía crear tipos tales como:

$$\alpha, \beta\ \in \Pi \Rightarrow (\alpha \to \beta) \in \Pi$$

correspondiendo ésta a una regla de tipado de la forma:

$$\frac{\Gamma \vdash A : \star \qquad \Gamma \vdash B : \star}{\Gamma \vdash A \to B : \star}$$

Teniendo esto en mente, es posible ahora en el sistema $\lambda\omega$ escribir una función $f = \lambda\alpha : \star\,.\,\alpha \to \alpha$ tal que $f\ \sigma = \sigma \to \sigma$. Como se puede apreciar, la función f toma como parámetro un tipo y retorna un nuevo tipo, por lo que empleando la notación de $\star$ propuesta, podríamos declarar el *"tipo"* de la función como $f : \star \to \star$. Llamaremos a $\star \to \star$ un *kind*, dejando reservada la palabra *tipo* para expresiones E, tales que $E : \star$.

Definición 2.11 (kind).
El conjunto de *kinds* K, es el definido por la siguiente gramática:

$$K\ ::=\ \star \mid K \to K$$

además introducimos una nueva constante, $\Box$, tal que cuando digamos que $k : \Box$, estaremos expresando que $k \in K$. †

Al igual que el conjunto de tipos, el conjunto de *kinds* también se encuentra generado por las reglas de tipado del $\lambda\omega$, como veremos más adelante. Definamos ahora las expresiones válidas que acepta este sistema.

Definición 2.12 (expresiones).
El conjunto de expresiones E se encuentra definido por la siguiente gramática:

$$E ::= V \mid \star \mid \Box \mid E\ E \mid \lambda V : E\,.\,E \mid E \to E \mid \Lambda V : E\,.\,E \mid \forall V : E\,.\,E \qquad \dagger$$

Una primera diferencia respecto a las definiciones de las estructuras fundamentales de los distintos sistemas que veníamos haciendo hasta ahora es que hemos unificado las definiciones de términos, tipos y kinds bajo la denominación de expresiones. Serán las reglas de tipado las que determinen cuando una expresión es un término, un tipo o un kind. Para ser más específicos:

Definición 2.13 (términos, tipos y *kinds*).
Dada una expresión E, decimos que E pertenece a la clase de:

- **kinds** si y sólo si $E : \Box$
- **tipos** si y sólo si existe una expresión k que sea de clase *kind* (i.e. $k : \Box$), tal que $E : k$
- **términos** si y sólo si existe una expresión T que sea de la clase de tipos (i.e. $\exists\, K$ tal que $T : K$ y $K : \Box$), tal que $E : T$ †

Antes de continuar, deberíamos hacer algunas aclaraciones sobre la nueva terminología que emplearemos. Dada la definición anterior, no resultaría extraño que digamos que una expresión es un kind, un tipo o un término. No obstante, al afirmar que una expresión E es un tipo podríamos estar haciendo referencia a que $E : \star$ (i.e. $E \in \Pi$) o tal vez que E es de la clase de tipos (lo cual implicaría que $E : \star, \star \to \star, \star \to \star \to \star, \ldots$, o $E : (\star \to \star) \to (\star \to \star)$, por ejemplo, pero no necesariamente que $E : \star$).

Otra confusión frecuente suele suscitarse al momento de emplear la frase *A tiene tipo B*, por lo que de ahora en más, al afirmar que *A tiene tipo B*, simplemente estaremos denotando que bajo un contexto Γ se tiene que $\Gamma \vdash A : B$.

Definición 2.14 (contextos).
El conjunto C de *contextos* es el conjunto de todas las listas de la forma:

$$[x_1 : A_1\,, \ldots, x_n : A_n]$$

donde tenemos que

$$A_1\,, \ldots, A_n \in E$$
$$x_1\,, \ldots, x_n \in V \qquad \dagger$$

Notación 2.4 (contextos):
Al contexto vacío lo denotaremos por $[]$. Si $X = [x_1 : A_1\,, \ldots, x_n : A_n]$ y $Y = [y_1 : B_1\,, \ldots, y_n : B_n]$, entonces:

- $X, x : A = [x_1 : A_1\,, \ldots, x_n : A_n, , x : A]$

- $X,Y = [x_1 : A_1 , \ldots , x_n : A_n, , y_1 : B_1 , \ldots , y_n : B_n]$ $\diamondsuit$

A diferencia de los contextos que habíamos definido para el sistema $\lambda\!\rightarrow$, en este nuevo sistema, los contextos dejan de ser conjuntos para pasar a ser listas, lo cual explica el cambio de notación para contextos. Esto resulta como consecuencia de que en el sistema $\lambda\omega$, el orden en el cual se toman los supuestos (es decir, los elementos del contexto) sí tiene importancia.

Si por ejemplo observamos las reglas de tipado dadas en la figura 2.3, notaremos por ejemplos que la regla de *start* nos dice que para poder asumir que $x : \alpha$, debemos primero demostrar algo sobre α (en particular, que $\alpha : s$ con $s \in \{\star, \Box\}$). Por otro lado, la regla de la *abstracción de tipos sobre términos* remueve el identificador que emplea en el orden inverso al cual se hicieron los supuestos sobre la expresión que está abstrayendo. Si el orden de los pares que conforman un contexto no fuese importante, entonces se podrían definir contextos que carecieran de todo sentido.

Ejemplo 2.7 (juicio mal formado):
Resulta evidente que bajo las reglas de tipado del sistema $\lambda\omega$, se puede efectuar la siguiente derivación

$$\dfrac{\dfrac{\dfrac{}{[] \vdash \star : \Box}\ (\text{axiom})}{[\alpha : \star] \vdash \alpha : \star}\ (\text{start})}{[\alpha : \star, x : \alpha] \vdash x : \alpha}\ (\text{start})$$

de todas formas, si ahora alteraramos el orden los elementos que conforman el contexto, veremos que ya no existen reglas de tipado bajo $\lambda\omega$ que nos permita derivar el juicio obtenido tal y como ocurre con:

$$[x : \alpha , \alpha : \star] \vdash x : \alpha \qquad *$$

No obstante, cuando demos a conocer las reglas de tipado que rigen a este sistema, debería quedar razonablemente claro que las únicas posibilidades de añadir nuevos elementos a un contexto son mediante la aplicación de las reglas *start* o *weak* y éstas lo permiten siempre y cuando se haya verificado que el tipo de la variable a añadir esté bien formado bajo los supuestos del contexto que está siendo incrementado.

Notación 2.5:
Sean M y N dos términos. Denotaremos la sustitución de las ocurrencias libres de la variables x en el término M por el término N como:

$$M[x := N]$$

a menudo también denotaremos esta sustitución como:

$$M[N/x] \qquad \diamondsuit$$

Definición 2.15 (reducción).
La relación $\rightarrow_\beta$ para el sistema $\lambda\omega$ es la menor relación sobre E la cual satisface que:

$$(\lambda x : A \,.\, M)N \;\rightarrow_\beta\; M[x ::= N]$$

$$(\Lambda A : M)N \;\rightarrow_\beta\; M[A ::= N]$$

y además, si $P, P' \in E$ y son tales que $P \rightarrow_\beta P'$, entonces dicha relación es cerrada bajo las reglas:

$$\lambda x : A \,.\, P \rightarrow_\beta \lambda x : A \,.\, P'$$

$$\lambda x : P \,.\, A \rightarrow_\beta \lambda x : P' \,.\, A$$

$$\Lambda x : A.P \rightarrow_\beta \Lambda x : A.P'$$

$$\Lambda x : P.A \rightarrow_\beta \Lambda x : P'.A$$

$$P\ Z \rightarrow_\beta P'\ Z$$

$$Z\ P \rightarrow_\beta Z\ P'$$

$\forall A, Z \in E$ y $\forall x \in V$.
A la clausura reflexiva, simétrica y transitiva de la relación $\rightarrow_\beta$ la denotaremos por $=_\beta$. †

Las reglas de tipado para el sistema $\lambda\omega$ se encuentran detalladas en la figura 2.3, pero aquí lo que haremos será ir introduciéndolas de a una por vez.

$$\text{(axiom)} \quad \frac{}{[] \vdash \star : \Box}$$

La regla de *axiom* dice simplemente que la constante $\star$ tiene tipo $\Box$. De esta forma, $\star$ queda definida como un *kind*, y partir de ella, todos los otros elementos que conformaban el conjunto K (*kinds*) se pueden construir empleando la regla de *kind formation* cuantas veces sea necesario.

$$\text{(start)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad s \in \{\star, \Box\} \wedge x \notin dom(\Gamma)$$

$$\text{(weak)} \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \qquad s \in \{\star, \Box\} \wedge x \notin dom(\Gamma)$$

La regla de *start* establece que si el tipo A se encuentra bien formado, entonces podemos derivar que una variable tiene tipo A, si es que el tipado de dicha variable es el último supuesto del contexto bajo en cual se esta trabajando. La regla de *weakening*, en tanto, nos dice que podemos descartar el último supuesto del contexto, siempre y cuando el tipo de dicho supuesto esté bien formado.

$$\text{(type formation 1)} \quad \frac{\Gamma \vdash A : \star \qquad \Gamma \vdash B : \star}{\Gamma \vdash A \rightarrow B : \star}$$

$$\text{(type formation 2)} \quad \frac{\Gamma \vdash A : \Box \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \forall x : A \,.\, B : \star} \qquad x \notin dom(\Gamma)$$

Las dos reglas de *type formation* nos permiten construir tipos. La primera, da la posibilidad de definir el tipo de las funciones. La segunda, en tanto, puede ser empleada para construir los tipos polimórficos que ya habíamos visto en el $\lambda 2$.

$$\text{(application term-term)} \quad \frac{\Gamma \vdash F : A \to B \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A : \star}{\Gamma \vdash F\ a : B}$$

$$\text{(application type-type)} \quad \frac{\Gamma \vdash F : A \to B \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A : \Box}{\Gamma \vdash F\ a : B}$$

$$\text{(application term-type)} \quad \frac{\Gamma \vdash F : \forall x : A\,.\,B \qquad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B[x ::= a]}$$

Luego nos encontramos con las tres reglas referidas a la aplicación de expresiones. La primera, indica que si contamos con una función que toma como parámetro un término retornando un nuevo término y además tenemos un término cuyo tipo coincide con el tipo del argumento de la función y dicho tipo se encuentra bien formado, entonces la función puede ser aplicada correctamente a este último término. La segunda se comporta de manera similar, con la salvedad de que la función recibe tipos como argumento, retornando un nuevo tipo. La tercera es una de las reglas nuevas que introduce este sistema, y nos permite aplicar una función que recibe un tipo como parámetro a un tipo en particular, para así obtener un nuevo término.

$$\text{(abstraction term-term)} \quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash A \to B : \star}{\Gamma \vdash \lambda x : A\,.\,b : A \to B} \quad x \notin dom(\Gamma)$$

$$\text{(abstraction type-type)} \quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash A \to B : \Box}{\Gamma \vdash \lambda x : A\,.\,b : A \to B} \quad x \notin dom(\Gamma)$$

$$\text{(abstraction type-term)} \quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash \forall x : A\,.\,B : \star}{\Gamma \vdash \Lambda x : A\,.\,b : \forall x : A\,.\,B} \quad x \notin dom(\Gamma)$$

Las reglas de abstracción se refieren a las maneras en las cuales se pueden abstraer las expresiones en este sistema. La primer regla nos indica que si contamos con un término cuyo tipo puede ser el valor de retorno de un valor funcional bien formado, entonces se puede abstraer ese término empleando el último supuesto del contexto en el cual resultaba válido dicho término. Más aún, el tipo del supuesto que emplearemos debe coincidir con el tipado del parámetro de entrada que recibía el valor funcional que verificamos estuviera bien formado. Las otras dos reglas se comportan de manera análoga, con la diferencia de que una la hace abstrayendo tipo sobre tipos y la otra abstrayendo tipos sobre términos.

$$\text{(conversion)} \quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s}{\Gamma \vdash a : B} \quad A =_\beta B \wedge s \in \{\star, \Box\}$$

Esta última regla, la de conversión, nos indica que si tenemos una expresión a de tipo A y además dicho tipo es β-equivalente a otra expresión B (la cual además debe estar bien formada), entonces es posible deducir que a tiene tipo B. Esta regla resulta necesaria basándonos en el hecho de que en el sistema $\lambda\omega$, los tipos de las expresiones no necesariamente deben encontrarse en su forma normal.

Ejemplo 2.8 (regla de conversión en $\lambda\omega$):
Sea $f = \lambda\alpha : \star . \alpha$, sea $id = \Lambda\alpha : \star . \lambda x : \alpha . x$ y sea $\Gamma = [Int : \star]$. Luego, tenemos que:

$$\dfrac{\dfrac{}{\Gamma \vdash id \ : \ \forall\alpha : \star . \alpha \to \alpha} \qquad \dfrac{\dfrac{}{\Gamma \vdash f \ : \ \star \to \star} \quad \dfrac{}{\Gamma \vdash Int \ : \ \star} \quad \dfrac{}{\Gamma \vdash \star \ : \ \square}\ (\text{Ax})}{\Gamma \vdash f\ Int \ : \ \star}\ (\text{ApTyTy})}{\Gamma \vdash id(f\ Int) \ : \ (f\ Int) \to (f\ Int)}\ (\text{ApTeTy})$$

Ahora bien, como $f\ Int = (\lambda\alpha : \star . \alpha)\ Int \ \to_\beta \ \alpha[\alpha ::= Int] = Int$, usando la regla de la conversión podemos efectuar la reducción que nos lleva a:

$$\dfrac{\dfrac{}{\Gamma \vdash id(f\ Int) \ : \ (f\ Int) \to (f\ Int)} \quad \dfrac{}{\Gamma \vdash Int \to Int \ : \ \star} \quad (f\ Int) \to (f\ Int) =_\beta Int \to Int}{\Gamma \vdash id(f\ Int) \ : \ Int \to Int}\ (\text{conversion})$$

*

Cuando en una derivación, la demostración de un juicio resulte trivial, omitiremos las premisas y la derivación de éstas, empleando una doble linea sobre el juicio en cuestión para denotar tal caso.

Definición 2.16 (reglas de tipado).
La relación de tipado $\vdash$ definida sobre $C \times E \times E$ se encuentra dada por las reglas de tipado:

(axiom)	$\dfrac{}{[] \vdash \star : \Box}$	
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A}$	$s \in \{\star, \Box\} \wedge x \notin dom(\Gamma)$
(weak)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x:C \vdash A : B}$	$s \in \{\star, \Box\} \wedge x \notin dom(\Gamma)$
(type formation 1)	$\dfrac{\Gamma \vdash A : \star \qquad \Gamma \vdash B : \star}{\Gamma \vdash A \to B : \star}$	
(type formation 2)	$\dfrac{\Gamma \vdash A : \Box \qquad \Gamma, x:A \vdash B : \star}{\Gamma \vdash \forall x:A.B : \star}$	$x \notin dom(\Gamma)$
(kind formation)	$\dfrac{\Gamma \vdash A : \Box \qquad \Gamma \vdash B : \Box}{\Gamma \vdash A \to B : \Box}$	
(application term-term)	$\dfrac{\Gamma \vdash F : A \to B \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A : \star}{\Gamma \vdash F\ a : B}$	
(application type-type)	$\dfrac{\Gamma \vdash F : A \to B \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A : \Box}{\Gamma \vdash F\ a : B}$	
(application term-type)	$\dfrac{\Gamma \vdash F : \forall x:A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B[x ::= a]}$	
(abstraction term-term)	$\dfrac{\Gamma, x:A \vdash b : B \qquad \Gamma \vdash A \to B : \star}{\Gamma \vdash \lambda x:A.b : A \to B}$	$x \notin dom(\Gamma)$
(abstraction type-type)	$\dfrac{\Gamma, x:A \vdash b : B \qquad \Gamma \vdash A \to B : \Box}{\Gamma \vdash \lambda x:A.b : A \to B}$	$x \notin dom(\Gamma)$
(abstraction type-term)	$\dfrac{\Gamma, x:A \vdash b : B \qquad \Gamma \vdash \forall x:A.B : \star}{\Gamma \vdash \Lambda x:A.b : \forall x:A.B}$	$x \notin dom(\Gamma)$
(conversion)	$\dfrac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s}{\Gamma \vdash a : B}$	$A =_\beta B \wedge s \in \{\star, \Box\}$

Figura 2.3: Reglas de tipado de $\lambda\omega$

Ejemplo 2.9:
Sean:

$$compose = \lambda\alpha : \star \to \star . \lambda\beta : \star \to \star . \lambda\gamma : \star . \alpha(\beta\ \gamma)$$

$$\text{Sea } \Delta = [] \text{ en AD} = \dfrac{\dfrac{}{\Delta \vdash \star : \Box}\ (\text{AX}) \qquad \dfrac{}{\Delta \vdash \star : \Box}\ (\text{AX})}{\Delta \vdash \star \to \star : \Box}\ (\text{KF})$$

$$\Gamma' = \alpha : \star \to \star, \beta : \star \to \star, \gamma : \star$$

Dadas las siguientes derivaciones:

$$D1 = \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \star \to \star : \Box}\ (\text{AD})}{\alpha : \star \to \star \vdash \alpha : \star \to \star}\ (\text{ST}) \qquad \dfrac{}{\alpha : \star \to \star \vdash \star \to \star : \Box}\ (\text{AD})}{\alpha : \star \to \star, \beta : \star \to \star \vdash \alpha : \star \to \star}\ (\text{WK}) \qquad \dfrac{\dfrac{\dfrac{}{\vdash \star : \Box}\ (\text{AX}) \qquad \dfrac{}{\vdash \star \to \star : \Box}\ (\text{AD})}{\alpha : \star \to \star \vdash \star : \Box}\ (\text{WK}) \qquad \dfrac{}{\alpha : \star \to \star \vdash \star \to \star : \Box}\ (\text{AD})}{\alpha : \star \to \star, \beta : \star \to \star \vdash \star : \Box}\ (\text{WK})}{\alpha : \star \to \star, \beta : \star \to \star, \gamma : \star \vdash \alpha : \star \to \star}\ (\text{WK})$$

$$D2 = \dfrac{\dfrac{\dfrac{}{\alpha : \star \to \star \vdash \star \to \star : \Box}\ (\text{AD})}{\alpha : \star \to \star, \beta : \star \to \star \vdash \beta : \star \to \star}\ (\text{ST}) \qquad \dfrac{\dfrac{\dfrac{}{\vdash \star : \Box}\ (\text{AX}) \qquad \dfrac{}{\vdash \star \to \star : \Box}\ (\text{AD})}{\alpha : \star \to \star \vdash \star : \Box}\ (\text{WK}) \qquad \dfrac{}{\alpha : \star \to \star \vdash \star \to \star : \Box}\ (\text{AD})}{\alpha : \star \to \star, \beta : \star \to \star \vdash \star : \Box}\ (\text{WK})}{\alpha : \star \to \star, \beta : \star \to \star, \gamma : \star \vdash \beta : \star \to \star}\ (\text{WK})$$

$$
D3 = \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \star : \Box}\,(\mathrm{AX})}{\alpha : \star \to \star \vdash \star : \Box}\,(\mathrm{WK}) \quad \dfrac{\dfrac{}{\vdash \star \to \star : \Box}\,(\mathrm{AD})}{\alpha : \star \to \star \vdash \star \to \star : \Box}\,(\mathrm{WK})}{\alpha : \star \to \star, \beta : \star \to \star \vdash \star : \Box}\,(\mathrm{ST})}{\alpha : \star \to \star, \beta : \star \to \star, \gamma : \star \vdash \gamma : \star}
$$

$$
D4 = \dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \alpha : \star \to \star}\,(\mathrm{D1}) \quad \dfrac{\dfrac{}{\Gamma' \vdash \beta : \star \to \star}\,(\mathrm{D2}) \quad \dfrac{}{\Gamma' \vdash \gamma : \star}\,(\mathrm{D3})}{\Gamma' \vdash \beta\ \gamma : \star}\,(\mathrm{ApTyTy}) \quad \dfrac{}{\Gamma' \vdash \star : \Box}\,(\mathrm{AX})}{\Gamma' \vdash \alpha(\beta\ \gamma) : \star}\,(\mathrm{ApTyTy})
$$

$$
D5 = \dfrac{\dfrac{\dfrac{}{\vdash \star \to \star : \Box}\,(\mathrm{AD})}{\vdash (\star \to \star) \to (\star \to \star) \to \star \to \star : \Box}\,(\mathrm{KF}) \quad \dfrac{\dfrac{}{\vdash \star \to \star : \Box}\,(\mathrm{AD})}{\vdash (\star \to \star) \to \star \to \star : \Box}\,(\mathrm{KF})}{\vdash (\star \to \star) \to (\star \to \star) \to \star \to \star : \Box}
$$

obtenemos así la prueba de *compose* en $\lambda\omega$:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha : \star \to \star, \beta : \star \to \star, \gamma : \star \vdash \alpha(\beta\ \gamma) : \star}\,(\mathrm{D4})}{\alpha : \star \to \star, \beta : \star \to \star \vdash \lambda\gamma : \star.\alpha(\beta\ \gamma) : \star \to \star}\,(\mathrm{AbTyTy}) \quad \dfrac{\dfrac{}{\alpha : \star \to \star, \beta : \star \to \star \vdash \star \to \star : \Box}\,(\mathrm{AD})}{\alpha : \star \to \star \vdash \star \to \star : \Box}\,(\mathrm{AbTyTy})}{\alpha : \star \to \star \vdash \lambda\beta : \star \to \star.\lambda\gamma : \star.\alpha(\beta\ \gamma) : (\star \to \star) \to \star \to \star}\,(\mathrm{D5})}{\Gamma \vdash \mathit{compose} : (\star \to \star) \to (\star \to \star) \to \star \to \star}\,(\mathrm{AbTyTy})
$$

*

En procura de conservar la cordura del autor, de ahora en más, las demostraciones de ejemplos que usemos serán bastante menos detallados, obviando aquellos pasos que puedan resultar triviales.

Ejemplo 2.10 (El tipo Lista de (Maybe Int)):
Renombraremos la expresión *compose* demostrada en ejemplo anterior como $\circ$. De esa manera, el tipo [Maybe Int], queda definido por:

$$LMI = List : \star \to \star, Maybe : \star \to \star \vdash \circ\ List\ Maybe\ Int\ :\ \star$$

Sea $\Gamma = List : \star \to \star, Maybe : \star \to \star$, y además

$$Der_1 = \dfrac{\dfrac{}{\Gamma \vdash \circ : (\star\to\star)\to(\star\to\star)\to\star\to\star}\ (\text{EJ 2.9}) \quad \dfrac{}{\Gamma \vdash List : \star\to\star} \quad \dfrac{}{\Gamma \vdash \star\to\star : \square}}{\dfrac{\Gamma \vdash \circ\ List : (\star\to\star)\to\star\to\star}{\Gamma \vdash \circ\ List\ Maybe : \star\to\star}\ (\text{ApTyTy}) \quad \dfrac{}{\Gamma \vdash Maybe : \star\to\star} \quad \dfrac{}{\Gamma \vdash \star\to\star : \square}}\ (\text{ApTyTy})$$

la prueba de *LMI* queda entonces como:

$$\dfrac{\dfrac{}{\Gamma \vdash \circ\ List\ Maybe\ :\ \star\to\star}\ (Der_1) \quad \dfrac{}{\Gamma \vdash Int\ :\ \star} \quad \dfrac{}{\Gamma \vdash \star : \square}\ (\text{AX})}{\Gamma \vdash \circ\ List\ Maybe\ Int\ :\ \star}\ (\text{ApTyTy}) \qquad *$$

2.1.5 El sistema de tipos λC

Hasta el momento hemos definido tres sistemas de tipos que cuentan con la peculiaridad de que en cada nuevo sistema que presentábamos las dependencias de sus antecesores se mantenían y al mismo tiempo una nueva dependencia sobre el conjunto de tipos y términos era añadida.

En el sistema $\lambda\!\to$ podíamos definir sólo términos que dependiesen de términos. Dado un término M, era posible construir el término $\lambda x : \sigma\ .M$, el cual es una función que espera un término como argumento, retornando como resultado un nuevo término. Por ejemplo $\lambda x : \sigma\ .x\ :\ \sigma \to \sigma$.

Luego, cuando estudiamos el sistema $\lambda 2$ nos encontramos con la situación en la que podíamos definir términos que dependiesen de tipos. Es decir, dado un término M, ya éramos capaces de construir un término de la forma $\Lambda\alpha.M$, el cual resultaba ser una función que tomaba un tipo como argumento y retornaba un nuevo término como resultado. Un ejemplo de este tipo de funciones que vimos era la identidad polimórfica: $\Lambda\alpha.\lambda x : \alpha\ .x\ :\ \forall\alpha.\alpha \to \alpha$.

Más tarde, cuando analizamos el sistema de tipos $\lambda\omega$, contamos con las herramientas necesarias como para poder definir tipos que dependiesen de tipos. Dicho de otro modo, dado un tipo σ, resultaba factible construir el tipo $\lambda\alpha : \kappa\ .\sigma$, el cual recibe un elemento cuyo tipo κ es un *kind* y como resultado retorna un nuevo tipo. Un ejemplo de esta situación se da en la función $(\lambda\alpha : \star\ .\alpha \to \alpha)\ :\ \star \to \star$.

Ahora entonces nos interesaría añadir la última dependencia que nos estaba faltando, la que nos permita escribir tipos que dependen de términos, usualmente denominados tipos dependientes. El sistema λC (también llamado *Cálculo de Construcciones*), introducido por Thierry Coquand y Gerard Huet [CH88] nos permitirá definir esta última dependencia.

Ejemplo 2.11 (tipos dependientes):
Un ejemplo clásico de tipos que dependen de términos es el de la generalización de la función *zip* que proporcionan algunos lenguajes funcionales. La función *zip* más empleada, es aquélla que se encarga de combinar los elementos de dos listas, obteniéndose una lista de pares formada por los elementos de las listas recibidas como argumentos. Dicho de otro modo:

$$zip : [a] \to [b] \to [(a,b)]$$
$$zip[a_1,a_2,\ldots,a_n][b_1,b_2,\ldots,b_n] = [(a_1,b_1),(a_2,b_2),\ldots,(a_n,b_n)]$$

De todas maneras, si bien lenguajes como Haskell [has90] proporcionan funciones estándares que permiten el *zipeo* de 2, 3 o incluso 4 listas de manera conjunta, un interrogante surge cuando se quieren *zipear n* listas de manera conjunta. Construir funciones cuyo tipo depende de términos resulta sencillo en lenguajes diseñados para tal fin, como lo es *Epigram* [Epi04].

No obstante, en lenguajes como *Haskell* ya no resulta tan sencillo definir funciones de tales características, lo cual no quiere decir que no existan mecanismos que nos permitan alcanzar resultados similares. A modo de ejemplo, citaremos la función *zipWith* desarrollada en [FI00], que examina una posible solución al problema de definir funciones cuyo tipo depende de algún término que reciben por parámetro.

Aquí sólo haremos mención de los resultados obtenidos en dicho artículo, para mayores detalles se recomienda consultar la publicación original. En la misma, los autores se encuentran abocados a la discusión sobre si realmente son necesarios los tipos dependientes en *Haskell*, para lo cual construyen una función *zipWith* dependendiente empleando las herramientas básicas del lenguaje. Para ello, definen las siguientes funciones auxiliares:

$$succ \quad :: \quad ([b] \to c) \to [a \to b] \to [a] \to c$$

$$repeat \quad :: a \to [a]$$

y redefiniendo los numerales inductivamente, de manera que el numeral *n* queda tipado como:

$$\overline{n} :: [a_1 \to \ldots \to a_n \to b] \to [a_1] \to \ldots \to [a_n] \to [b]$$

podemos definir el $0,1,\ldots$ como:

$$\begin{array}{rcl} zero & = & id \\ one & = & succ\ zero \\ two & = & succ\ one \\ \vdots & & \\ \overline{n} & = & succ\ \overline{n-1} \end{array}$$

Luego, si definimos *zipWith* como:

$$\begin{array}{lcl} zipWith & :: & ([a] \to b) \to a \to b \\ zipWith\ n\ f & = & n\ (repeat\ f) \end{array}$$

y aunque no hemos dados detalles sobre la implementación de *succ*, chequeando los tipos de las funciones que hemos dado, no debería resultar difícil notar que:

$$zipWith\ \bar{n} :: (a_1 \to \ldots \to a_n \to b) \to [a_1] \to \ldots \to [a_n] \to [b]$$

en particular:

$$\begin{array}{lll} \mathit{zipWith\ one} & :: & (a \to b) \to [a] \to [b] \\ \mathit{zipWith\ two} & :: & (a \to b \to c) \to [a] \to [b] \to [c] \\ \mathit{zipWith\ three} & :: & (a \to b \to c \to d) \to [a] \to [b] \to [c] \to [d] \\ & \vdots & \end{array}$$

o, si resulta más claro:

$$\begin{array}{lll} \mathit{zipWith\ one\ id} & :: & [a] \to [a] \\ \mathit{zipWith\ two}\ (,) & :: & [a] \to [b] \to [(a,b)] \\ \mathit{zipWith\ three}\ (,,) & :: & [a] \to [b] \to [c] \to [(a,b,c)] \\ & \vdots & \end{array}$$

aqui solo intentamos dar una idea de lo que podría llegar a ser un tipo dependiente. Sin embargo, si bien el tipo de la función *zipWith* parece depender en cada caso de uno de los términos que recibe como argumento, en realidad el tipo del resultado sólo depende del tipo del argumento. *

Si recordamos las reglas de tipado dadas para el sistema $\lambda\omega$ en la figura 2.3, notaremos que para cada clase de dependencia las reglas referidas a la abstracción y aplicación entre expresiones diferían para cada caso. Al definir el nuevo sistema λC, una opción plausible sería la de añadir algunas reglas nuevas que determinen el comportamiento a seguir en el caso de tener tipos que dependen de términos. Estas reglas serían:

(type formation 3) $$\frac{\Gamma \vdash A : \star \qquad \Gamma, x:A \vdash B : \Box}{\Gamma \vdash \forall x:A.B : \Box} \qquad x \notin dom(\Gamma)$$

(application type-term) $$\frac{\Gamma \vdash F : \forall x:A.B \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A : \star}{\Gamma \vdash F\ a : B[x ::= a]}$$

(abstraction term-type) $$\frac{\Gamma, x:A \vdash b : B \qquad \Gamma \vdash \forall x:A.B : \Box}{\Gamma \vdash \Lambda x:A.b : \forall x:A.B} \qquad x \notin dom(\Gamma)$$

A simple vista puede obeservarse que añadimos una nueva premisa a la regla de *application type-term*, para poder diferenciarla de la regla de *application term-type* que habíamos definido anteriormente. Esto se debe a que, de acuerdo a las reglas definidas en la figura 2.3, y siguiendo el razonamiento de la regla *type formation 2*, A, en aquel entonces podía ser sólamente de tipo $\Box$, mientras que con la nueva regla esta permitiendo que A tenga tipo $\star$. Si estas nuevas reglas fueran añadidas a las de entonces, deberíamos también incluir una nueva premisa en la regla de *application term-type*, pidiendo que A tenga tipo $\Box$.

No obstante, en lugar de ampliar el número de reglas para este nuevo sistema estamos interesados en definir este conjunto de manera más concisa. Persiguiendo este objetivo trataremos de definir una regla para la abstracción y otra para la aplicación que puedan ser aplicadas en cualquier clase de dependencia entre expresiones con que nos encontremos. Para lograrlo, primero deberemos definir el concepto de *producto dependiente*.

Definición 2.17 (producto dependiente).
Definiremos una nuevo sintaxis, el *producto dependiente* Π:

$$\Pi x : A \,.\, B$$

que denotará el tipo de las funciones que van de valores de tipo A a valores de tipo B, donde el tipo B puede depender del valor recibido a través del parámetro x. †

Notación 2.6 (producto dependiente):
A menudo usaremos $A \to B$ para denotar $\Pi x : A \,.\, B$ con $x \notin FV(B)$. ♢

Si pensamos ahora en Π como un renombre del ∀, ahora sí, ya estamos en condiciones de definir los cuatro tipos de dependencias posibles empleando el *producto dependiente*.

Primero veamos el caso en el que teníamos términos dependiendo de términos. Es decir, estamos en presencia de un término M, el cual recibe como parámetro un término de tipo A y retorna un nuevo término de tipo B (en el sistema $\lambda\to$ diríamos que $M : A \to B$). Luego M no es más que una función que toma valores de tipo A, retornando un valor de tipo B, pero este tipo B tiene la peculiaridad de que no depende del valor recibido por M. Entonces resulta razonable afirmar que $M \;:\; \Pi x : A \,.\, B$, donde x no ocurre libremente en B (lo que en algunos casos suele denotarse como $M \;:\; \Pi\_ : A \,.\, B$), y además $A, B : \star$.

Lo mismo ocurre cuando nos referimos a tipos que dependen de tipos. En tal caso, si por ejemplo $f : \star \to \star$, entonces f no es más que una función que mapea valores de tipo $\star$ en valores de tipo $\star$. Una vez mas, dado que el tipo del valor de retorno no depende del argumento recibido por f (que no debe confundirse con el valor de retorno de f, que es un tipo y sí depende del argumento de f), podemos describir el tipo de dicha función como $f : \Pi\_ : \star \,.\, \star$. En general, podemos tipar la dependencia entre tipos empleando la notación $\Pi x : A \,.\, B$, donde $A, B : \Box$.

Si ahora nos concentramos en los casos en los que términos dependían de tipos, veremos que $M \;:\; (\Pi\alpha : \star \,.\, B)$. De manera más general, podemos afirmar que es posible tipar términos que dependen de tipos empleando la notación $\Pi\alpha : A \,.\, B$, donde $A : \Box$ y $B : \star$.

Finalmente llegamos al caso en que nos enfrentamos a la situación de tener tipos que dependen de términos. Siguiendo el mismo razonamiento que hemos venido desarrollando, al querer tipar funciones que toman como parámetro términos y devuelven como resultado términos (que pueden depender del valor recibido por la función), seguramente denotaremos el tipo de dicha funciones como $\Pi x : A \,.\, B$, donde $A : \star$ y $B : \Box$.

Tal y como debe resultar claro a esta altura, todos los tipos de funciones que denoten cualquier tipo de dependencia entre términos y tipos puede ser resumida en una sola regla de tipado de la forma:

$$\frac{\Gamma \vdash A \;:\; s \qquad \Gamma, x : A \vdash B \;:\; t}{\Gamma \vdash \Pi x : A \,.\, B \;:\; t} \; s,t \in \{\star, \Box\}$$

Dependiendo de como se escojan los s y t, estaremos antes todas las posibles combinaciones de dependencias de términos y tipos. Para aclarar un poco la situación, tal vez resulte de utilidad la siguiente tabla:

s	t	Dependencia
$\star$	$\star$	Términos que dependen de términos
$\Box$	$\star$	Términos que dependen de tipos
$\Box$	$\Box$	Tipos que dependen de tipos
$\star$	$\Box$	Tipos que dependen de términos

De manera análoga a la que empleamos para definir el tipo de las funciones, es posible definir una única regla que determine el tipado de la aplicación y la abstracción sobre expresiones. Las mismas se verán en más detalle en la figura 2.21 de reglas de tipado para λC.

Definición 2.18 (expresiones).
El conjunto de *expresiones* E del sistema λC se encuentra formado por la siguiente gramática:

$$E ::= V \mid \star \mid \Box \mid E\ E \mid \lambda V : E\,.\,E \mid \Pi V : E\,.\,E \qquad \dagger$$

Como se puede apreciar, el conjunto de expresiones resulta más breve, ya que $\forall$ y $\rightarrow$ son unificados en Π, mientras que λ y Λ lo son en λ.

Definición 2.19 (reducciones).
La relación $\rightarrow_\beta$ es la menor relación sobre E, la cual satisface que:

$$(\lambda x : A\,.\,M)N \rightarrow_\beta M[x ::= N]$$

y además, dados $P, P' \in E$ tales que $P \rightarrow_\beta P'$, también resulta cerrada bajo las reglas:

$$\begin{aligned}
\lambda x : A\,.\,P &\rightarrow_\beta \lambda x : A\,.\,P' \\
\lambda x : P\,.\,A &\rightarrow_\beta \lambda x : P'\,.\,A \\
\Pi x : A\,.\,P &\rightarrow_\beta \Pi x : A\,.\,P' \\
\Pi x : P\,.\,A &\rightarrow_\beta \Pi x : P'\,.\,A \\
P\ Z &\rightarrow_\beta P'\ Z \\
Z\ P &\rightarrow_\beta Z\ P'
\end{aligned}$$

$\forall x \in V, \forall A, Z \in E.$ $\dagger$

Definición 2.20 (contextos).
El conjunto de contextos C del sistema λC se encuentra definido de la misma manera en que definimos los contextos válidos para el sistema $\lambda\omega$, en la definición 2.14. $\dagger$

Definición 2.21 (reglas de tipado).
La relación de tipado $\vdash$ definida sobre $C \times E \times E$ se encuentra dada por las reglas de tipado de la figura 2.21.

(axiom)	$\dfrac{}{[] \vdash \star : \Box}$	
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A}$	$x \notin dom(\Gamma) \wedge s \in \{\star, \Box\}$
(weak)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x:C \vdash A : B}$	$x \notin dom(\Gamma) \wedge s \in \{\star, \Box\}$
(pi)	$\dfrac{\Gamma \vdash A : s \qquad \Gamma, x:A \vdash B : t}{\Gamma \vdash (\Pi x:A.B) : t}$	$x \notin dom(\Gamma) \wedge s,t \in \{\star, \Box\}$
(abstraction)	$\dfrac{\Gamma, x:A \vdash b : B \qquad \Gamma \vdash (\Pi x:A.B) : t}{\Gamma \vdash (\lambda x:A.b) : (\Pi x:A.B)}$	$x \notin dom(\Gamma) \wedge t \in \{\star, \Box\}$
(application)	$\dfrac{\Gamma \vdash f : (\Pi x:A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x := a]}$	
(conversion)	$\dfrac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash a : B}$	$s \in \{\star, \Box\}$

Figura 2.4: Reglas de tipado de λC

Las tres primeras reglas son idénticas a las definidas para el sistema $\lambda\omega$. La regla de *product* es la que nos permite definir tipos de funciones que, como ya habíamos visto al momento de definir el *producto dependiente*, agrupa a todas las clases de dependencias que podríamos tener entre términos y tipos.

Todas las reglas que nos permitían abstraer expresiones en $\lambda\omega$, ahora se ven resumidas en una sola regla de *abstraction*. Ésta nos indica que si tenemos una expresión b de tipo B, bajo el supuesto de que x sea de un tipo A, y si además podemos determinar que las funciones que toman valores de tipo A y retornan valores de tipo B se encuentran bien formadas, entonces, es posible abstraer el identificador x de la expresión b, quedando ésta como una función que va de valores de tipo A en valores de tipo B.

De manera similar, la aplicación de expresiones se resume en una sola regla de *application* que establece el hecho de que si contamos con una expresión f, a la cual hemos logrado tipar como una función que toma valores de tipo A y retorna valores de tipo B (donde B puede depender del argumento que recibe f) y hemos logrado hallar una expresión a de tipo A, entonces es posible aplicar la función f a la expresión a, obteniéndose una expresión del tipo (tal vea dependiente) B. La regla de *conversion* aún se comporta tal y como lo hacía bajo el sistema $\lambda\omega$.

A continuación daremos algunos ejemplos de tipos que dependan de términos. Si bien estos ejemplos no resultan muy comunes en el ámbito de la programación funcional (e incluso puedan parecer absurdos para el lector), serviran para ilustrar el uso de las nuevas reglas de tipado que hemos introducido.

Ejemplo 2.12 (un tipo que depende de un término):
Veamos que en λC, asumiendo que α es un tipo, podemos probar que:

$$[\alpha : \star] \vdash (\lambda x:\alpha.\alpha) : \alpha \to \star$$

Sea $\Gamma = [\alpha : \star]$ en:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \alpha : \star} \quad \dfrac{}{\Gamma \vdash \alpha : \star}\,(\text{wk})}{\Gamma, x:\alpha \vdash \alpha : \star} \quad \dfrac{\dfrac{}{\Gamma \vdash \alpha : \star} \quad \dfrac{\dfrac{}{\Gamma \vdash \star : \Box} \quad \dfrac{}{\Gamma \vdash \alpha : \star}\,(\text{wk})}{\Gamma, x:\alpha \vdash \star : \Box}\,(\text{pi})}{\Gamma \vdash \Pi x:\alpha.\star : \Box}\,(\text{ab})}{\Gamma \vdash \lambda x:\alpha.\alpha \;:\; \Pi x:\alpha.\star}$$

En este caso nos encontramos en presencia de una función que recibe como argumento valores de tipo α, retornando el valor α. Aunque en cierta forma no estamos aprovechando del todo la dependencia, ya que se retorna siempre el mismo α. $*$

Ejemplo 2.13 (aplicación de un tipo dependiente):
Veamos que en el sistema λC la siguiente expresión tiene sentido:

$$\Gamma,\ d : Nat \to \star \vdash d\ 42\ :\ \star$$

donde:

$$\Gamma = [Int : \star,\ 42 : Nat]$$

siendo su prueba:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash Nat\ :\ \star}\,(\text{wk,st,ax}) \quad \dfrac{}{\Gamma \vdash \star : \Box}\,(\text{wk,st,ax})}{\Gamma,\ d : \Pi x : Nat.\star \vdash d\ :\ \Pi x : Nat.\star}\,(\text{st}) \quad \dfrac{}{\Gamma,\ d : \Pi x : Nat.\star \vdash 42\ :\ Nat}\,(\text{wk,st,ax})}{\Gamma,\ d : \Pi x : Nat.\star \vdash d\ 42\ :\ \star}\,(\text{ap})$$

Básicamente, lo que estamos diciendo es que podemos contar con una función que toma un valor de tipo *Int* (es decir que dicho valor es un término, 42 en nuestro ejemplo), retornando un valor de tipo $\star$ (i.e. un tipo), siempre y cuando estemos suponiendo que *Int* es un tipo y que 42 tiene tipo *Int*. En este caso particular, *d* 42 se correspondería con el tipo de las listas de longitud 42.

Un ejemplo de tal $d : Int \to \star$ es la función que para cada natural n devuelve el tipo de las listas de longitud n. Una situación similar ocurre con la función $zipWith$ introducida en el ejemplo 2.11, para la cual, si establecemos que:

$$zipWith\ :\ Nat \to [\sigma_1] \to \ldots \to [\sigma_n] \to (\sigma_1, \ldots, \sigma_n)$$

tenemos que el tipo de la función resultante depende del natural que recibe como argumento. $*$

2.2 Unificando sistema de tipos: el *Lambda Cubo*

Ahora que ya hemos dado un vistazo a algunos sistemas de tipos simples, estamos en condiciones de estudiar lo que se conoce como el *Lambda Cubo*, introducido por Barendregt [Bar91]. Éste recibe su nombre debido a que puede ser percibido como un cubo, conteniendo un sistema de tipos particular en cada uno de sus vértices, siendo desde luego, ocho en total. De esta manera, este $\lambda-cubo$ conforma un marco natural en el cual muchos de los sistemas de tipos frecuentemente utilizados en su versión *a la*

Church, pueden ser estudiados de manera uniforme. El $\lambda-cubo$ no es un sistema en sí mismo, sino mas bien una estructura que permite unificar los criterios de análisis de otros sistemas de tipos.

No debería extrañarnos que en algunos de los vértices de dicho $\lambda-cubo$ se encuentren los sistemas de tipos $\lambda\!\rightarrow$, $\lambda 2$, $\lambda\omega$ y λC, siendo este último el sistema más robusto y completo de los que conforman el cubo. De hecho, el cubo puede ser observado como una manera de representar la estructura del *Cálculo de Construcciones* en relación con otros sistemas de tipos del cálculo lambda. Ésto se debe a que todos los sistemas del $\lambda-cubo$ se encuentran definidos bajo el mismo conjunto de reglas de tipado, siendo la principal diferencia entre ellos la manera en que se permite efectuar la abstracción de términos y tipos, algo que resultará más claro cuando demos las reglas de tipado del sistema.

Si bien sistemas como el $\lambda\!\rightarrow$ y el $\lambda 2$ forman parte del cubo, las versiones que se emplean en el mismo no son la mismas que dimos en las secciones 2.1.2 y 2.1.3. A modo de ejemplo, a esta altura ya se ha unificado el $\forall$ y $\rightarrow$ bajo la notación de Π y algo similar hemos realizado con λ y Λ. No obstante las versiones que conviven en el $\lambda-cubo$ son en esencia equivalentes a las dadas anteriormente, de manera que el espíritu de dichos sistemas no va a verse alterado.

Definición 2.22 (expresiones).
El conjunto de *expresiones* E del sistema $\lambda-cubo$ se encuentra formado por la siguiente gramática:

$$E ::= V \mid S \mid E\ E \mid \lambda V : E\,.\,E \mid \Pi V : E\,.\,E$$

donde V es un conjunto infinito de variables. Por defecto, diremos que $\{\star,\Box\} = S$. Usaremos $s_1, s_2, s_3, \ldots$ para denotar elementos de S. †

Definición 2.23 (reducciones).
La noción de β-conversión y β-reducción sobre el conjunto de expresiones E se encuentran definidas por la regla de reducción:

$$(\lambda x : A\,.\,M)N \ \rightarrow_\beta \ M[x ::= N]$$

teniendo ésta que satisfacer las mismas propiedades que la relación de reducción para λC dadas en la definición 2.19. †

Definición 2.24 (contextos).
El conjunto de contextos C del sistema $\lambda-cubo$ se encuentra definido de la misma manera en que definimos los contextos válidos para el sistema $\lambda\omega$, en la definición 2.14. †

Antes de poder dar las reglas de tipado que rigen al sistema $\lambda-cubo$, necesitaremos definir el conjunto de *reglas* que aceptará cada sistema que lo compone.

Definición 2.25 (reglas).
Dado el conjunto de pares $R_{cubo} = \{(\star,\star),(\star,\Box),(\Box,\star),(\Box,\Box)\}$, para cada uno de los sistemas de tipos que conforman el $\lambda-cubo$ definimos R como el subconjunto de R_{cubo} dado por la siguiente tabla:

$\lambda\!\rightarrow$	$(\star,\star)$			
$\lambda 2$	$(\star,\star)$	$(\Box,\star)$		
λP	$(\star,\star)$		$(\star,\Box)$	
$\lambda P2$	$(\star,\star)$	$(\Box,\star)$	$(\star,\Box)$	
$\lambda\underline{\omega}$	$(\star,\star)$			$(\Box,\Box)$
$\lambda\omega$	$(\star,\star)$	$(\Box,\star)$		$(\Box,\Box)$
$\lambda P\underline{\omega}$	$(\star,\star)$		$(\star,\Box)$	$(\Box,\Box)$
$\lambda C = \lambda P\omega$	$(\star,\star)$	$(\Box,\star)$	$(\star,\Box)$	$(\Box,\Box)$

†

Nótese que el par $(\star,\star)$ es común a todos los conjuntos de reglas, esto se debe a que obviamente, en todos los sistemas que conforman el cubo es posible definir términos que dependen de términos, siendo las combinaciones de las demás clases de dependencias las que caracterizan a cada uno de los sistemas de tipos involucrados.

Si bien algunos de los sistemas dados en la tabla ya han sido analizados previamente en el presente trabajo, allí también se encuentran cuatro nuevos sistemas que no hemos mencionado hasta ahora: El sistema λP [dB70, HHP87], el sistema $\lambda P2$ [LM88], el sistema $\lambda\underline{\omega}$ [dLst] y el sistema $\lambda P\underline{\omega}$. De estos últimos no daremos mayores detalles, ya que no resultan completamente relevantes para el tema que es objeto de nuestro estudio.

A menudo se suele dar una representación gráfica del sistema $\lambda-cubo$ que ayuda a comprender la relación entre los distintos sistemas de tipos que lo componen. Dicha interpretación puede apreciarse en la figura 2.5

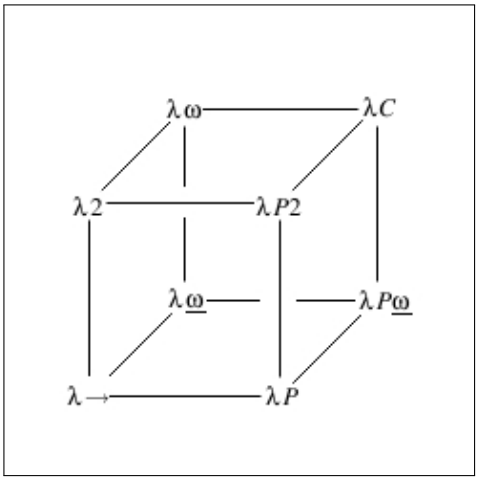

Figura 2.5: Representación gráfica del $\lambda-cubo$

Como se puede apreciar, en los vértices se ubican los sistemas de tipos que lo componen. En un espacio cartesiano tridimensional, el cubo se puede construir fijando el sistema $\lambda\!\rightarrow$ y luego el resto de los sistemas se obtienen añadiendo nuevos elementos al conjunto de reglas R que restringe cada sistema de acuerdo al eje en el cual se esta desplazando. Los pares de reglas que se añaden respetan la siguiente tabla:

eje	se añade a R
X	$(\star,\Box)$
Y	$(\Box,\star)$
Z	$(\Box,\Box)$

Además, si rotamos la representación del cubo 45 grados, de manera tal que el sistema $\lambda\!\rightarrow$ se ubique en la parte inferior y el sistema λC en la superior, obtendremos un conjunto parcialmente ordenado bajo la relación $\subseteq$, que conserva su significado usual.

Definición 2.26 (reglas de tipado).
La relación de tipado $\vdash$ definida sobre $C \times E \times E$ se encuentra dada por las reglas de tipado de la figura 2.6.

(axiom)	$\dfrac{}{[] \vdash \star : \Box}$	
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A}$	$x \notin dom(\Gamma) \wedge s \in S$
(weak)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x:C \vdash A : B}$	$x \notin dom(\Gamma) \wedge s \in S$
(pi)	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash (\Pi x:A.B) : s_2}$	$x \notin dom(\Gamma) \wedge (s_1,s_2) \in R$
(abstraction)	$\dfrac{\Gamma, x:A \vdash b : B \qquad \Gamma \vdash (\Pi x:A.B) : s}{\Gamma \vdash (\lambda x:A.b) : (\Pi x:A.B)}$	$x \notin dom(\Gamma) \wedge s \in S$
(application)	$\dfrac{\Gamma \vdash F : (\Pi x:A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B[x:=a]}$	
(conversion)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}$	$s \in S$

Figura 2.6: Reglas de tipado de $\lambda-cubo$

Como se puede apreciar, la mayoría de las reglas conservan la estructura que presentaban para el sistema λC. De todas maneras, la regla de *product* presenta una modificación, y es ésta la que nos permite construir valores de funciones que tomen y devuelvan términos o tipos[2] rigiéndose por las reglas de dependencias dadas por el conjunto R del sistema de tipos en que estemos interesados. En sí, la única diferencia con λC reside en la condición lateral de la regla *pi*.

Podríamos demostrar que cada uno de los sistemas aún conservan sus propiedades, pero sólo nos limitaremos a dar algunos ejemplos de expresiones pertenecientes a algunos de los sistemas que conforman el $\lambda-cubo$. En ciertos casos serán expresiones

[2] que a esta altura ya conviven bajo la denominación de expresiones

que ya habíamos demostrado en los sistemas de tipos más simples y nos servirán para probar que la esencia de los mismos aún se conserva.

Ejemplo 2.14 (sistema $\lambda\!\rightarrow$):
En todos los sistemas del $\lambda-cubo$ nos encontramos en condiciones de demostrar expresiones que ya habíamos probado en el sistema $\lambda\!\rightarrow$.

Por ejemplo, la función identidad dada por:

$$\dfrac{\dfrac{}{[A:\star,\, a:A] \vdash a : A}\ (\text{st,ax}) \qquad \dfrac{\dfrac{}{[A:\star] \vdash A : \star}\ (\text{st,ax}) \qquad \dfrac{}{[A:\star,\, x:A] \vdash A : \star}\ (\text{wk,st,ax})}{[A:\star] \vdash \Pi x:A.A : \star}\ (\text{pi})}{[A:\star] \vdash \lambda a:A.a : \Pi x:A.A}\ (\text{ab})$$

o la constante K, donde si suponemos que:

$$\begin{array}{lcl} \Gamma & = & [A:\star,\, b:\star] \\ \Gamma_a & = & \Gamma,\, a:A \\ \Gamma_{ab} & = & \Gamma,\, a:A,\, b:B \\ \Gamma_{ay} & = & \Gamma, a:A,\, y:B \\ \Gamma_x & = & \Gamma,\, x:A \\ \Gamma_{xy} & = & \Gamma,\, x:a,\, y:B \end{array}$$

tenemos que:

$$\dfrac{\dfrac{}{\Gamma_{ab} \vdash a : A} \qquad \dfrac{\dfrac{}{\Gamma_a \vdash B : \star} \qquad \dfrac{}{\Gamma_{ay} \vdash A : \star}\ (\text{pi})}{\Gamma_a \vdash \Pi y:B.A : \star}\ (\text{ab})}{\Gamma_a \vdash \lambda b:B.a : \Pi y:B.A} \qquad \dfrac{\dfrac{}{\Gamma \vdash A : \star} \qquad \dfrac{\dfrac{}{\Gamma_x \vdash B : \star} \qquad \dfrac{}{\Gamma_{xy} \vdash A : \star}\ (\text{pi})}{\Gamma_x \vdash \Pi y:B.A : \star}\ (\text{pi})}{\Gamma \vdash \Pi x:A.\Pi y:B.A : \star}\ (\text{ab})$$
$$\Gamma \vdash \lambda a:A.\lambda b:B.a : \Pi x:A.\Pi y:B.A$$

Y a pesar de que las pruebas parecieran haberse tornado más dificultosas y engorrosas, no debería resultar difícil notar que la esencia de la misma aún se conserva. $*$

También debería notarse que el espíritu de dependencia con que contaba cada sistema se encuentra plasmado en la regla del producto dependiente (*Pi*). Veamos a modo de ejemplo, que la función polimórfica de identidad que vimos para el sistema $\lambda 2$ aún vale en todos los sistemas del $\lambda-cubo$ tales que $(\star,\Box) \in R$.

Ejemplo 2.15 (sistema $\lambda 2$):
Sean:

$$\begin{array}{lcl} \Gamma_\alpha & = & [\alpha:\star] \\ \Gamma_{\alpha a} & = & [\alpha:\star,\, a:\alpha] \\ \Gamma_{\alpha x} & = & [\alpha.\star,\, x:\alpha] \end{array}$$

en

$$\dfrac{\dfrac{}{\Gamma_{\alpha a} \vdash a : \alpha} \qquad \dfrac{\dfrac{\dfrac{}{[] \vdash \star : \Box}\ (\text{ax})}{\Gamma_\alpha \vdash \alpha : \star}\ (\text{st}) \qquad \dfrac{}{\Gamma_{\alpha x} \vdash \alpha : \star}\ (\text{pi})}{\Gamma_\alpha \vdash \Pi x:\alpha.\alpha : \star}\ (\text{ab})}{\Gamma_\alpha \vdash \lambda a:\alpha.a : \Pi x:\alpha.\alpha} \qquad \dfrac{\dfrac{}{[] \vdash \star : \Box}\ (\text{ax}) \qquad \dfrac{\dfrac{}{\Gamma_\alpha \vdash \alpha : \star} \qquad \dfrac{}{\Gamma_{\alpha x} \vdash \alpha : \star}\ (\text{pi})}{\Gamma_\alpha \vdash \Pi x:\alpha.\alpha : \star}\ (\text{pi})}{[] \vdash \Pi\alpha:\star.\Pi x:\alpha.\alpha : \star}\ (\text{ab})$$
$$[] \vdash \lambda\alpha:\star.\lambda a:\alpha.a : \Pi\alpha:\star.\Pi x:\alpha.\alpha$$

$*$

No obstante, con lo expuesto hasta el momento nos basta para comprender los *Sistemas de Tipos Puros (PTS)* que son la base sobre la cual se erige nuestro trabajo y por lo tanto no enunciaremos más propiedades ni ejemplos del $\lambda-cubo$, abocándonos a partir de este momento al estudio de los *PTS*.

2.3 Generalizando sistemas de tipos: los *PTS*

El método empleado en la construcción del sistema $\lambda-cubo$, fue además generalizado de manera independiente por el matemático italiano Stefano Berardi y el holandés Jaw Terlouw [Ter89]. Dichos estudios derivaron en la noción de lo que hoy conocemos como los *Sistemas de Tipos Puros (PTS)*. Muchos de los sistemas de tipos del cálculo lambda en su versión *a la Church* pueden ser analizados como *PTS*s ya que las sutiles diferencias que existen entre éstos puede ser expresada empleando la notación de los *PTS* que veremos ahora a continuación.

Sólo daremos la noción básica de los *PTS* y algunas propiedades que nos resultarán de utilidad en desarrollos que haremos más adelante en el presente trabajo. Para un análisis más detallado de estos sistemas se sugiere consultar [Bar92], [Geu93] y [GN91]

2.3.1 Motivación

Como ya vimos, empleando los sistemas de tipos que conformaban el $\lambda-cubo$ estábamos en condiciones de escribir funciones de identidad que nos resultaban interesantes.

- En $\lambda\!\rightarrow$, la función de identidad monomórfica sobre términos:

$$\lambda x:\alpha\,.\,x\ :\ \alpha\rightarrow\alpha$$

- En $\lambda 2$, la función de identidad polimórfica sobre términos:

$$\Lambda\alpha\ .\ \lambda x:\alpha\,.\,x\ :\ \forall\alpha.\alpha\rightarrow\alpha$$

- En $\lambda\omega$, la función de identidad monomórfica sobre tipos (de *kind* κ):

$$\lambda\alpha:\kappa\,.\,\alpha\ :\ \kappa\rightarrow\kappa$$

Pero de todas formas aún nos resulta imposible definir una función de identidad polimórfica sobre tipos, es decir, no estamos en condiciones de definir una función que tome como argumento un *kind* κ y retorne una función sobre tipos de *kind* κ. Sería esperable poder escribir tal función como:

$$(\lambda\kappa:\square\,.\,\lambda\alpha:\kappa\,.\,\alpha)\ :\ \Pi\kappa:\square\,.\,\kappa\rightarrow\kappa$$

De todas maneras, si bien dicha expresión es sintácticamente correcta, si intentamos demostrar dicha expresión en el sistema $\lambda-cubo$, nos encontraremos con que:

$$\dfrac{\dfrac{\vdots}{[\kappa:\square]\vdash\lambda\alpha:\kappa\,.\,\alpha\ :\ \Pi x:\kappa\,.\,\kappa}\qquad\dfrac{[]\vdash\square:?_1\qquad\dfrac{\vdots}{[\kappa:\square]\vdash\Pi x:\kappa\,.\,\kappa:?_2}\ (\text{pi})}{[]\vdash\Pi\kappa:\square\,.\,\Pi x:\kappa\,.\,\kappa:?_2}\ (\text{ab})}{[]\vdash\lambda\kappa:\square\,.\,\lambda\alpha:\kappa\,.\,\alpha\ :\ \Pi\kappa:\square\,.\,\Pi x:\kappa\,.\,\kappa}$$

De acuerdo a la regla *pi*, necesitaríamos definir $?_1$ y $?_2$ de manera tal que se encuentren definido en el conjunto de reglas *R* del sistema de tipo que estamos usando. Incluso si asumimos que estamos empleando el conjunto *R* que dimos en la definición

2.25 para el sistema λC, veremos que $?_1$ debería tomar el valor de $\star$ o $\Box$. De tomas formas, sin importar cual de estos valores le asignemos a $?_1$, la primer premisa de la regla *pi* nos pide que tipemos $\Box$, pero ésto implicaría la existencia de una regla de axioma de la forma:

$$[] \vdash \Box : s \qquad \text{con } s \in \{\star, \Box\}$$

Pero tal regla no esta disponible. Ésto nos demuestra que el sistema $\lambda-cubo$ no es lo suficientemente general como para ser empleado en cualquier ocasión. La función de identidad polimórfica sobre tipos que intentamos definir quedó claramente fuera del alcance de dicho sistema. Ante la presencia de inconvenientes como el que acabamos de plantear, es que resultó conveniente definir una nueva generalización del $\lambda-cubo$, conocida como la teoría de los *PTS*s.

2.3.2 Definiendo los *PTS*

Como dijimos, los *PTS* pueden ser considerados una generalización del sistema $\lambda-cubo$, lo que nos permitirá trabajar con expresiones que hasta ahora escapaban del alcance de los sistemas de tipos que habíamos considerado. Esto resultará viable gracias a las extensiones que presentan los *PTS* respecto al sistema $\lambda-cubo$, entre las cuales se destacan:

- En los *PTS* es posible escoger el conjunto de *sorts* S de manera completamente libre, no así en el sistema $\lambda-cubo$, donde el conjunto de *sorts* estaba restringido a $\{\star, \Box\}$.
- Se define una nueva relación $A \subseteq S \times S$, que establece el conjunto de *axiomas*, en lugar del único axioma $\star : \Box$ que teníamos hasta ahora.
- La regla del producto también se generaliza, liberándose de la condición de que el producto mismo debe tener el mismo tipo que los valores que éste retornaba, lo que implica que ya no necesitaremos si $B : \alpha$ entonces $(\Pi x : A \,.\, B) : \alpha$. Esto se logra definiendo las reglas como una tripla en lugar de un par, como teníamos en el sistema $\lambda-cubo$. El conjunto de reglas R puede ser cualquier subconjunto de $S \times S \times S$, en lugar de las que habíamos fijado en la definición 2.25.

Definición 2.27 (sistemas de tipos puros).
Un Sistema de Tipos Puros (*PTS* por su sigla en inglés *Pure Type System*) es una tripla (S, A, R) donde:

- S es un conjunto de constantes al que llamaremos *sorts*.
- A es un subconjunto de $S \times S$, al que llamaremos *axiomas*.
- R es un conjunto de reglas de la forma (s_1, s_2, s_3), con $s_1, s_2, s_3 \in S$.

A menudo usaremos la notación (s_1, s_2) para significar (s_1, s_2, s_2) †

Definición 2.28 (expresiones).
El conjunto de expresiones E de un *PTS* se construye a partir de la siguiente gramática:

$$E ::= V \mid S \mid E\ E \mid \lambda V : E\,.\,E \mid \Pi V : E\,.\,E \qquad \dagger$$

Definición 2.29 (contextos).
El conjunto de contextos C de un *PTS* se encuentra definido de la misma manera en que definimos los contextos válidos para el sistema $\lambda\omega$, en la definición 2.14. †

Definición 2.30 (reducciones).
La noción de β-conversión y β-reducción sobre el conjunto de expresiones E se encuentran definidas por la regla de reducción:

$$(\lambda x : A\,.\,M)N \ \rightarrow_\beta \ M[x ::= N]$$

teniendo ésta que satisfacer las mismas propiedades que la relación de reducción para λC dadas en la definición 2.19. †

Definición 2.31 (reglas de tipado).
La relación de tipado $\vdash$ definida sobre $C \times E \times E$ se encuentra dada por las reglas de tipado de la figura 2.7.

Es cierto que la condición de que $B =_\beta B'$ de la regla de *conversion* puede a priori no ser decidible, pero de todas maneras nada evita que reemplacemos dicha condición por la de:

$$B' \rightarrow_\beta B \quad \vee \quad B \rightarrow_\beta B'$$

sin la necesidad de alterar el conjunto de sentencias derivables. †

Empleando la noción de *PTS*, tal y como ocurría en el caso del $\lambda-cubo$, podemos definir los sistemas de tipos simples con que trabajamos al inicio del capítulo, de manera tal que todas las expresiones que resultaban tipadas en dichos sistemas, ahora pueden serlo en su correspondiente versión de *PTS*.

Ejemplo 2.16 (sistemas de tipos expresados como *PTS*):
Vemos como resultaría la especificación de los *PTS* que representan a algunos de los sistemas de tipos que ya hemos mencionado.

- El sistema $\lambda\rightarrow$ dado en la sección 2.1.2:

$$\begin{array}{lcl} S & = & \{\star, \Box\} \\ A & = & \{(\star, \Box)\} \\ R & = & \{(\star, \star)\} \end{array}$$

(axiom)	$\dfrac{}{[] \vdash s_1 : s_2}$	$(s_1, s_2) \in A$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A}$	$x \notin dom(\Gamma) \wedge s \in S$
(weakening)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x:C \vdash A : B}$	$x \notin dom(\Gamma) \wedge s \in S$
(product)	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash (\Pi x:A.B) : s_3}$	$x \notin dom(\Gamma) \wedge (s_1, s_2, s_3) \in R$
(abstraction)	$\dfrac{\Gamma, x:A \vdash b : B \qquad \Gamma \vdash (\Pi x:A.B) : s}{\Gamma \vdash (\lambda x:A.b) : (\Pi x:A.B)}$	$x \notin dom(\Gamma) \wedge s \in S$
(application)	$\dfrac{\Gamma \vdash F : (\Pi x:A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B[x := a]}$	
(conversion)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}$	$s \in S$

Figura 2.7: Reglas de tipado de un *PTS*

- El sistema $\lambda 2$ dado en la sección 2.1.3:

$$\begin{array}{lcl} S & = & \{\star, \Box\} \\ A & = & \{(\star, \Box)\} \\ R & = & \{(\star,\star),\ (\Box,\star)\} \end{array}$$

- El sistema λC dado en la sección 2.1.5:

$$\begin{array}{lcl} S & = & \{\star, \Box\} \\ A & = & \{(\star, \Box)\} \\ R & = & \{(\star,\star),\ (\Box,\star),\ (\star,\Box),\ (\Box,\Box)\} \end{array}$$

- El sistema λHOL, el cual describe la lógica de alto orden de Church [Chu40]:

$$\begin{array}{lcl} S & = & \{\star, \Box, \triangle\} \\ A & = & \{(\star, \Box),\ (\Box, \triangle)\} \\ R & = & \{(\star,\star),\ (\Box,\star),\ (\Box,\Box)\} \end{array}$$

- Una conjetura efectuada por *de Bruijn* establece que toda la matemática anterior al 1800 puede ser formalizada en sistema λPAL, el cual es un subsistema del sistema $\lambda\!\rightarrow$ y se encuentra especificado por el *PTS*:

$$\begin{array}{lcl} S & = & \{\star, \Box, \triangle\} \\ A & = & \{(\star, \Box)\} \\ R & = & \{(\star,\star,\triangle),\ (\star,\Box,\triangle),\ (\Box,\star,\triangle),\ (\Box,\Box,\triangle),\ (\star,\triangle,\triangle),\ (\Box,\triangle,\triangle)\} \end{array}$$

y de manera similar, muchos otros sistemas de tipos pueden ser definidos como especificación de *PTS*. ∗

2.3.3 Algunas propiedades de los *PTS*

Ahora que ya hemos definido la noción de *PTS*, enumeraremos algunas definiciones y propiedades de los mismos que nos serán de utilidad en el presente trabajo.

Definición 2.32 (igualdad de sentencias).
Dadas dos sentencias $(x:A)$, $(y:B)$, diremos que éstas son iguales, esto es $x:A = y:B$, si $x = y$ y $A = B$, donde esta última es la igualdad sintáctica de expresiones generadas a partir de la gramática dada en la definición 2.28. †

Definición 2.33 (contextos).
Sean $\Gamma = [x_1 : A_1, \ldots, x_n : A_n]$ y $\Delta = [y_1 : B_1, \ldots, y_m : B_m]$ dos contextos, entonces:

1. Γ es llamado *legal* si existen $P, Q \in E$ tales que $\Gamma \vdash P : Q$.
2. Una sentencia $x : A$ *pertenece a* Γ, esto es $(x : A) \in \Gamma$, si $x = x_i$ y $A = A_i$ para algún i en $1, \ldots, n$.
3. Γ *está contenido en* Δ, esto es $\Gamma \subseteq \Delta$ cuando para todo $(x : A) \in \Gamma$, se tiene que $(x : A) \in \Delta$. †

Lema 2.1 (variables libres para *PTS*s):
Sea $\Gamma = [x_1 : A_1, \ldots, x_n : A_n]$ un contexto legal, tal que $\Gamma \vdash B : C$. Entonces se satisface que:

1. $x_1, \ldots, x_n$ son todos distintos.
2. $FV(B), FV(C) \subseteq \{x_1, \ldots, x_n\}$
3. $FV(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$, para $i \in \{1, \ldots, n\}$. ♠

Prueba Por inducción en la derivación de $\Gamma \vdash B : C$. ■

Lema 2.2 (*type checking* y *type inference* de *PTS*s normalizable):
Sea (S, A, R), con S finito, la especificación de un *PTS* (débil o fuertemente) normalizado, entonces los problemas de *type checking* y *type inference* son decidibles. ♠

Prueba La prueba puede ser consultada en [vBJ93]. ■

Lema 2.3 (inicio para *PTS*):
Si Γ es un contexto legal, entonces:

1. Si $(s_1 : s_2)$ es un axioma, entonces $\Gamma \vdash s_1 : s_2$
2. $(x : A) \in \Gamma \quad \Rightarrow \quad \Gamma \vdash x : A$ ♠

Dado el contexto Γ y el contexto $\Delta = [x_1 : M_1, \ldots, x_n : M_n]$, con $\Gamma \vdash \Delta$ querremos significar que:

$$\Gamma \vdash x_1 : M_1 \quad \wedge \quad \ldots \quad \wedge \Gamma \vdash x_n : M_n$$

Lema 2.4 (transitividad para *PTS*):
Sean Γ y Δ dos contextos, tales que Γ es legal. Entonces:

$$\Gamma \vdash \Delta \quad \wedge \quad \Delta \vdash A : B \qquad \Rightarrow \qquad \Gamma \vdash A : B$$ ♠

Prueba La prueba puede ser consultada en [Bar92] ■

Lema 2.5 (reducción de tipos para *PTS*):
Para todo *PTS* se satisface que:

$$\Gamma \vdash A : B \quad \wedge \quad B \rightarrow_\beta^* B' \qquad \Rightarrow \qquad \Gamma \vdash A : B'$$ ♠

Prueba La prueba puede ser consultada en [Bar92] ■

Es cierto que el conjunto de propiedades que satisfacen los PTS es mucho más amplio que el que acabamos de dar. Sin embargo, sólo requeriremos de estas últimas para comprender las bases de los conceptos que manejaremos en capítulos posteriores.

Capítulo 3

Sobre tipos, términos, teoremas y pruebas

En este capítulo nos abocaremos al estudio del razonamiento que nos permitirá obtener pruebas de teoremas a partir de derivaciones de términos. Para lograrlo, primero debemos comprender el pensamiento que plantea el intuicionismo y la lógica que involucra al mismo. Luego veremos el resultado fundamental que nos relaciona términos del cálculo lambda con pruebas de teoremas y finalmente cómo los *PTS* que vimos en el capítulo anterior le añaden expresividad a los resultados obtenidos.

A partir de este momento, el término *intuicionismo* se tornará recurrente en el resto del trabajo, de manera que consideramos pertinente dejar en claro el significado del mismo. El *intuicionismo* puede considerarse como un acercamiento a la matemática, en el que en la búsqueda de resultados se intenta mantener un razonamiento constructivista de los pasos que conducen a la demostración de la propiedad deseada, tal y como una persona *intuitivamente* lo haría. Esta nueva filosofía fue propuesta por Brouwer [Bro13] a comienzos del siglo pasado y en esencia se oponía al formalismo impulsado por Hilbert que prevalecía en el estudio de la matemática.

3.1 Hacia la unificación de términos y pruebas

Básicamente en la teoría intuicionista, para demostrar la existencia de un objeto que satisface ciertas propiedades, debe resultar posible construir paso a paso dicho objeto, lo cual claramente contrasta con la idea general que establece que la existencia de una entidad puede ser probada refutando su no existencia. A modo de ejemplo, siguiendo los lineamientos de la teoría intuicionista, no resulta factible hallar una prueba de $A \vee \neg A$, proposición que resulta demostrable en la lógica tradicional. Esto se debe a que de acuerdo al *Teorema de Incompletitud* de Gödel [Göd31] es posible construir una sentencia que no puede ser probada ni refutada, por ello, no hay manera de construir paso a paso una prueba de dicha sentencia.

Para cuando lleguemos al final de este capítulo, habremos comprendido la relación existente entre los *PTS* que vimos en el capítulo anterior y el *intuicionismo*. Pero para lograr dicho objetivo, primero deberemos fijar algunas ideas acerca de la lógica que rige el pensamiento *intuicionista*.

3.1.1 La semántica de Heyting

Muchos de los avances logrados en el área de la matemática intuicionista se deben gracias al desarrollo de una *lógica intuicionista* que estableció un sistema formal donde se puede trabajar y manipular los formalismos de dicha teoría. En esta lógica desarrollada por Heyting [Hey31] el énfasis está puesto en preservar la justificación en cada uno de los pasos de la derivación que conducen a la prueba buscada, más que en el valor de verdad de la misma. En realidad, lo que nosotros llamamos lógica intuicionista no es más que una familia de sistemas lógicos, algunos de los cuales veremos más adelante.

Surge entonces la duda de, dada una sentencia A, ¿qué es específicamente una prueba de A?. Aquí no estamos interesados en su notación sintáctica, sino más bien en su significado semántico. Pero al indagar un poco más, nos daremos cuenta de que lo que llamamos *prueba* no es más que una descripción de una entidad que ya es un proceso en sí mismo, de manera que podemos responder nuestra incógnita analizando la estructura de la sentencia A.

- Para sentencias atómicas, se asume que se conoce qué es una prueba de la misma, de manera intrínseca.
- Una prueba de $A \wedge B$, consiste en una prueba de A y una prueba de B, adoptando la forma de un par (p,q) donde p es una prueba de A y q es una prueba de B.
- Una prueba de $A \vee B$, consiste en una prueba de A o una prueba de B y se lo representa como un par (i,p) donde se da alguna de las siguientes condiciones:
  - $i = 0$, y p es una prueba de A, o
  - $i = 1$, y p es una prueba de B.
- Una prueba de $A \Rightarrow B$, consiste en un método que convierte cualquier prueba de A en una prueba de B. Es decir, resulta en una función f, que mapea cada prueba p de A a un prueba $f(p)$ de B.
- La negación $\neg A$ es considerada como $A \Rightarrow \bot$, donde $\bot$ es una sentencia que no tiene ninguna prueba posible.
- Una prueba de $\forall x.A$ es una función f, la cual mapea cada punto a del dominio de la cuantificación a una prueba $f(a)$ de $A[a/x]$.
- Una prueba de $\exists x.A$ es un par (a,p), donde a es un punto en el dominio de la cuantificación y p en una prueba de $A[a/x]$.

No profundizaremos más en la filosofía de la lógica intuicionista. Nuestra intención es sólo la de denotar su significado e importancia, de manera que resulte comprensible la relación existente entre ésta y el prototipo de herramienta que desarrollamos.

3.1.2 El isomorfismo de Curry-Howard

Un buen punto del cual partir en el entendimiento de nuestro trabajo, es la *correspondencia de Curry-Howard* [How80], planteada por el matemático Haskell Curry y el lógico William Howard. Ésta formaliza la estrecha relación que existe entre programas computacionales y pruebas matemáticas.

En esencia, la misma establece dos relaciones. Por un lado, afirma que el tipo de los valores retornados por una función resultan análogos a teoremas lógicos. Más aún, las

hipótesis bajo las cuales se satisface dicho teorema no son más que las correspondientes a las establecidas por los tipos de los argumentos de dicha función. Y por otro lado, afirma que el programa que computa dicha función, resulta análogo a la prueba de ese mismo teorema. De esta manera, resultan comunicados los sistemas del cálculo lambda que vimos en el capítulo 2 con la teoría de tipos que enunciaremos en las próximas dos secciones.

Si lo vemos en una dirección, el isomorfismo opera construyendo programas a partir de pruebas de teoremas basados en la lógica constructivista que habíamos mencionado. En tanto que analizándolo en el otro sentido, nos encontramos con un mecanismo capaz de generar pruebas a partir de un programa (del cual estamos seguros de su correctitud por la manera en que fue elaborado). Claro que para lograr una mayor expresividad de estos programas, se requiere de teoría de tipos que así lo permitan, y es allí en donde los *PTS* que habíamos definido entran en juego.

Ejemplo 3.1 ($A \Rightarrow A$)**:**
Aplicado a la semántica de *Heyting* que dimos anteriormente, podemos ver por ejemplo que la sentencia $A \Rightarrow A$ es demostrada por la función identidad:

$$\lambda x : A \,.\, A$$

ya que está asocia a cada prueba p de A exactamente la misma prueba. $*$

Ejemplo 3.2 ($\beta \Rightarrow \alpha$)**:**
La interpretación como tipos de la proposición $\beta \Rightarrow \alpha$ resulta ser $\beta \rightarrow \alpha$. Claramente, no existe ningún término cerrado que satisfaga dicho tipo, ya que deberíamos definir una función que recibe como argumento un valor de tipo β y retornase un valor de tipo α, completamente diferente al recibido como parámetro. Este hecho, analizado en el entorno del isomorfismo de *Curry-Howard*, nos está indicando que $\beta \Rightarrow \alpha$ no es un teorema de la lógica proposicional, lo cual claramente resulta verdadero. $*$

De todas maneras, la recíproca no resulta verdadera. Es decir, dado un teorema lógico, no siempre va a resultar posible hallar un término cerrado que habite el tipo que se corresponde con dicha proposición.

Con lo expuesto hasta aquí nos alcanza para deducir que no siempre será posible probar una afirmación lógica bajo la teoría intuicionista. Sin embargo, si logramos hallar un término cerrado que habite en el tipo que se corresponde con dicha afirmación, habremos demostrado la veracidad de dicha proposición. Más aún, el término hallado será un programa funcional que nos brinda una manera algorítmica de probar tal enunciado. Éste es el principio que rige el funcionamiento de nuestro prototipo de herramienta.

3.2 Teoría de Tipos Intuicionista

La *Teoría de Tipos Intuicionista* consiste en un sistema lógico y una teoría de conjuntos que permiten trabajar sobre las bases de la matemática constructivista. Esta nueva teoría fue introducida por el matemático sueco Martin-Löf [ML84] y se sienta sobre

los resultados obtenidos por el isomorfismo de Curry-Howard que vimos. Si bien dicha correspondencia en un principio fue formulada para la lógica proposicional y el cálculo lambda simple con tipos, gracias a los aportes de los tipos dependientes que brinda la teoría de tipos intuicionista, es posible extender el isomorfismo hacía otros sistemas lógicos, como por ejemplo la lógica de predicados.

3.2.1 Tipos dependientes

Como dijimos, resulta factible ampliar el alcance del isomorfismo de Curry-Howard hacia otros sistemas lógicos más complejos al trabajar con tipos dependientes. Anteriormente, cuando introdujimos el sistema $\lambda\omega$ en la sección 2.1.4, hicimos notar cierta interdependencia entre términos y tipos del cálculo lambda. Lo que haremos ahora será dar con un poco más de detalle los constructores de tipos que tenemos en la teoría de tipos intuicionista y que, tal como su nombre lo indica, nos permitirán construir nuevos tipos empleando tipos ya conocidos. Entre los tipos dependientes mas comúnmente empleados se encuentran:

Tipos finitos Como esta teoría se encuentra definida sobre la teoría de conjuntos, no resulta extraño encontrarnos con los tipos que pertenecen a un conjunto finito. Ejemplos de éstos son el tipo vacío (0), el tipo unidad (1[1]), o el tipo de los Booleanos (2). Empleando esta clase de tipos, es posible escribir la negación como:

$$\neg P = P \to 0$$

Tipos de equivalencia Dados $a, b : A$, el tipo $a = b$ es aquél que denota las pruebas que establecen que a es igual a b. En realidad existe un único término que habita en este tipo y dicho término en el isomorfismo representa la prueba de la reflexividad:$\Pi a : A \,.\, a = a$

Tipos Π Ya habíamos realizado una introducción a éstos tipos cuando dimos la noción de producto dependiente en la definición 2.17. Como dijimos entonces, éstos denotan el tipo de las funciones cuyo valor de retorno puede depender del valor que reciben como argumento. De esa manera, podríamos expresar el tipo de los vectores de valores reales n-dimensionales como:

$$\Pi n : \mathbb{N} \,.\, Vec(n, \mathbb{R})$$

mientras que las funciones clásicas en las que el tipo del valor de retorno no dependen del tipo del valor recibido se denotan como:

$$\mathbb{N} \to \mathbb{R} \quad = \quad \Pi x : \mathbb{N} \,.\, \mathbb{R}$$

siguiendo el isomorfismo de Curry-Howard, los tipos Π nos son útiles a la hora de modelar la implicación y la cuantificación universal, por ejemplo, un término de tipo:

$$\Pi m : \mathbb{N} \,.\, \Pi n : \mathbb{N} \,.\, m + n = n + m$$

resulta ser una función que dados dos números naturales cualesquiera, retorna una prueba de que la suma para ese par de números resulta conmutativa. También se lo emplea para definir la cuantificación universal:

$$\forall x : A.P \quad \equiv \quad \Pi x : P \,.\, Q(x)$$

[1]Los familiarizados con Haskell pueden reconocer este tipo como ()

y la cuantificación universal, que cuando $Q(x)$ resulta ser el mismo tipo para cualquier valor x de A, queda definida como:

$$P \Rightarrow Q \quad \equiv \quad \Pi x : P \,.\, Q(x)$$

Tipos Σ Éstos tipos generalizan el producto cartesiano para modelar pares en donde la segunda componente depende de la primera. Por ejemplo el tipo:

$$\Sigma n : \mathbb{N}.Vec(n, \mathbb{R})$$

denota el tipo de los pares en donde la primer componente es un número natural y la segunda componente es un vector de valores reales cuya longitud es igual al primer elemento del par. El producto cartesiano tradicional se obtiene cuando la segunda componente no depende en nada de la primera. Del mismo modo, se emplean los tipos Σ para denotar el concepto de unión disjunta y bajo el isomorfismo de Curry-Howard resulta factible modelar la conjunción:

$$A \wedge B \quad \equiv \quad A \times B \quad \equiv \quad \Sigma x : A \,.\, B(x)$$

cuando $B(x)$ resulta ser el mismo objeto para todo elemento x que vive en A, y la cuantificación existencial de la forma:

$$\exists x : A.P \quad \equiv \quad \Sigma x : A \,.\, P(x)$$

Unión disjunta Si A y B son dos tipos, entonces la unión disjunta denotada por $A + B$ también es un tipo. Éste resulta ser el tipo de objetos de la forma $i(a)$, donde a es de tipo A o $j(b)$, donde b es de tipo B e i, j denotan las inyecciones canónicas. Trasladándonos al isomorfismo, cuando A y B representan fórmulas lógicas, tenemos que:

$$A \vee B \quad \equiv \quad A + B$$

Tipos inductivos Uno de los ejemplos clásicos de tipos inductivos es el de los números naturales $\mathbb{N}$. Este tipo es generado por los constructores:

$$\begin{array}{rcl} 0 & : & \mathbb{N} \\ succ & : & \mathbb{N} \to \mathbb{N} \end{array}$$

al trabajar con la idea de que las proposiciones pueden ser consideradas tipos, resultará interesante contar con la recursión primitiva y la inducción, donde una prueba inductiva será un termino de tipo:

$$P(0) \to (\Pi n : \mathbb{N} \,.\, P(n) \to P(succ\ n)) \to \Pi n : \mathbb{N} \,.\, P(n)$$

Los tipos inductivos nos permitirán definir relaciones inductivas entre proposiciones. Es decir, por ejemplo, podemos definir:

$$Even \ : \ \Pi n : \mathbb{N} \,.\, \mathbb{S}\approx$$

utilizando:

$$\begin{array}{rcl} ev_0 & = & Even\ 0 \\ ev_s & = & \Pi n : \mathbb{N} \,.\, Even\ n \to Even\ (succ(succ\ n)) \end{array}$$

Resumiendo, hasta aquí hemos visto, por un lado, las analogías que establece el isomorfismo de Curry-Howard. Luego vimos que las construcciones más frecuentes de la lógica de predicados pueden ser representadas a partir de tipos dependientes. Al mismo tiempo dimos a entender que los sistemas de tipos puros que habíamos definido en el capítulo anterior permitían un cierto grado de interdependencia entre términos y tipos, y que los mismos nos brindarían la gran expresividad que requeríamos para sacarle el máximo provecho a los conceptos aportados por la teoría de tipos intuicionista. Ahora sólo nos resta establecer cual será el aporte concreto que efectuarán los *PTS* y es de este tema que nos encargaremos a partir de este punto.

3.3 El aporte de los *PTS*

Para entender ya definitivamente el aporte que efectúan los sistemas de tipos puros comenzaremos por introducir algunos sistemas lógicos que no hemos dado hasta ahora. No los definiremos formalmente, sino que sólo los mencionaremos y explicaremos como éstos se relacionan con la noción de sistemas de tipos puros. Esto nos mostrará de qué manera enunciados lógicos pueden ser mapeados a estructuras de *PTS* y luego emplear nuestro prototipo de herramienta para lograr una prueba de las mismas.

3.3.1 Conectando sistemas de tipos con sistemas lógicos

Para comenzar, introduciremos ocho sistemas lógicos que forman parte de la teoría intuicionista. Cuatro de ellos se corresponden a la lógica proposicional y los otro cuatro a la lógica de predicado. Éstos son:

PROP	Lógica proposicional.
*PROP*2	Lógica proposicional de segundo orden.
$PROP\underline{\omega}$	Lógica proposicional débil de alto orden.
$PROP\omega$	Lógica proposicional de alto orden.
PRED	Lógica de predicados.
*PRED*2	Lógica de predicados de segundo orden.
$PRED\underline{\omega}$	Lógica de predicados débil de alto orden.
$PRED\omega$	Lógica de predicados de alto orden.

Estamos asumiendo que todos éstos sistemas son mínimos en el sentido de que sólo se encuentran definidos por operadores $\Rightarrow$ y $\forall$. Aún así, para los sistemas de segundo y alto orden los operadores $\neg$, $\wedge$, $\vee$, $\exists$ e incluso la equivalencia de Leibniz's pueden definirse en función de esos dos operadores mínimos como se detalla en [Bar92]. Los sistemas $PROP\underline{\omega}$ y $PREP\underline{\omega}$ cuentan con variables para proposiciones y predicados de alto orden respectivamente, pero no se pueden efectuar cuantificación sobre éstas. Siguiendo esta definición, si relacionamos a estos ocho sistemas lógicos a través de la relación de inclusión, podemos representarlo a través de un cubo, al que se suele denominar *cubo lógico*, tal y como se puede apreciar en la figura 3.3.1.

Si observamos la representación gráfica del sistema $\lambda-cubo$ obtenida en la figura 2.5, notaremos que éste mantiene una correspondencia muy cercana con el cubo conformado por sistemas lógicos dado en la figura 3.3.1. Cada sistema L_i que ocupa un vértice del *cubo lógico* se corresponde con el sistema λ_i del $\lambda-cubo$. Esta nueva representación gráfica nos da una idea visual de como se puede llegar a representar el concepto de *proposiciones como tipos*. A partir de la idea propuesta por el *cubo lógico*, Berardi [Ber90] observó que los ocho sistemas lógicos que lo componen podían ser

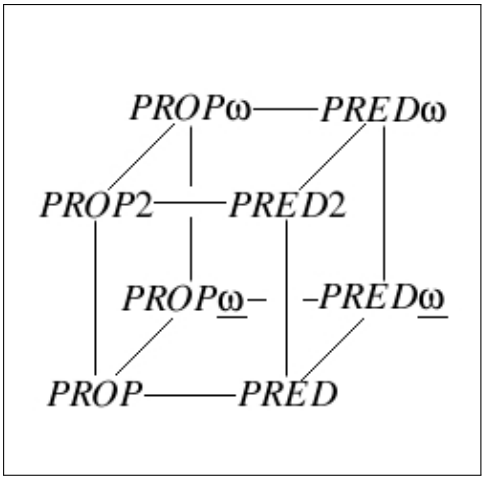

Figura 3.1: Representación gráfica del *cubo lógico*

descriptos como *PTS*, de manera que la idea de las *proposiciones como tipos* se logra expresar de una manera canónica.

Entonces, si establecemos el mapeo que nos lleve de sistemas lógicos a sistemas de tipos, estaremos en condiciones de representar formulas lógicas del sistema L_i a través de un tipo en el sistema λ_i, siendo dicha función transformadora la que nos define la idea de *proposiciones como tipos*. De manera análoga a lo que comentamos al introducir la noción del *isomorfismo de Curry-Howard* en la sección 3.1.2, si llamamos $f_{L\to T}$ a la función que nos lleva de sistemas lógicos a sistemas de tipos resulta cierto para sistemas como los definidos en el *cubo lógico*[2] que si una fórmula A es demostrable en el sistema L_i, entonces el tipo $f_{L\to T}(L_i)$ se encuentra habitado. Más aún, un término que habita dicho tipo puede ser hallado de manera canónica a partir de una prueba de A, por lo que diferentes pruebas de A se interpretan como distintos términos del tipo $f_{L\to T}(L_i)$.

Desde que se definió esta correspondencia para los sistemas del *cubo lógico*, se ha logrado demostrar que muchos otros sistemas satisfacen esta misma propiedad, tal y como se detalla en [ML84], [Ste72] y [Luo90]. Así mismo, en la sección 3.1.2 afirmábamos que si un tipo se encontraba habitado, entonces la fórmula lógica que se encuentra representada por dicho tipo también resultaba demostrable. En lo que a sistemas del *cubo lógico* respecta, esta afirmación también se satisface, tal y como se demuestra en [Mar] y [Ber90] para el sistema *PRED* y en [Geu89] para el sistema *PRED*ω.

No enunciaremos una prueba formal, pero si daremos una idea de como éstos sistemas lógicos pueden llegar a ser expresados a través de los *PTS*. Para ello, lo primero que deberemos hacer es definir los *sorts* que emplearemos en la especificación de los *PTS*. Éstos son:

- $\star^s$ representará a los conjuntos.
- $\star^p$ representará a las proposiciones, de manera que las fórmulas lógicas serán elementos de $\star^p$.

[2] Una prueba de ello puede encontrarse en [Geu93]

- $\star^f$ representará a las funciones de primer orden entre conjuntos de $\star s$.
- $\Box^s$ contiene a $\star^s$.
- $\Box^p$ contiene a $\star^p$.

Empleando ahora estos *sorts* podemos definir un sistema de tipos puro que represente los predicados definibles en el sistema lógico *PRED*. A este nuevo *PTS* le daremos el nombre de $\lambda PRED$ y su especificación estará dada por:

$$\begin{array}{lcl} S & = & \{\star^s, \star^p, \star^f, \Box^s, \Box^p\} \\ A & = & \{(\star^s, \Box^s),\ (\star^p, \Box^p\} \\ R & = & \{(\star^p, \star^p, \star^p),\ (\star^s, \star^p, \star^p),\ (\star^s, \Box^p, \Box^p),\ (\star^s, \star^s, \star^f),\ (\star^s, \star^f, \star^f)\} \end{array}$$

Puede que no resulte evidente a primera vista, pero al especificar el sistema $\lambda PRED$ de esta manera, estamos simulando la lógica del sistema *PRED*. Siguiendo los lineamientos del mapeo de expresiones lógicas a tipos dependientes que dimos en la sección 3.2.1, tenemos por ejemplo:

- La regla de $(\star^p, \star^p, \star^p)$ nos permite construir la implicación entre dos fórmulas. Es decir, dadas las proposiciones $\varphi, \psi : \star^p$, tenemos que:

$$(\varphi \to \psi) \quad \equiv \quad (\Pi x : \varphi \,.\, \psi) : \star^p$$

- La regla de $(\star^s, \star^p, \star^p)$ nos permite definir la cuantificación sobre conjuntos. Si $A : \star^s$ es un conjunto y $\varphi : \star^p$ es una proposición, tenemos que:

$$(\forall x : A \,.\, \varphi) \quad \equiv \quad (\Pi x : A \,.\, \varphi) : \star^p$$

- La regla de $(\star^s, \Box^p, \Box^p)$ nos permite la construcción de predicados de primer orden. Es decir, si $A : \star^s$ es un conjunto, podemos definir:

$$(A \to \star^p) \quad \equiv \quad (\Pi x : A \,.\, \star^p) : \Box^p$$

  y por lo tanto, si $P : A \to \star^p$ es el nuevo constructor de predicados del tipo que acabamos de definir arriba y a es un elemento de A, entonces tenemos que la aplicación de los mismos genera una nueva proposición, es decir que:

$$P\ a\ :\ \star^p$$

- La regla de $(\star^s, \star^s, \star^f)$ nos permite la formación de funciones entre los conjuntos que viven en $\star^s$. Dicho de otro modo, si $A, B : \star^s$ son dos conjuntos, entonces:

$$(A \to B)\ :\ \star^f$$

- La regla de $(\star^s, \star^f, \star^f)$ es la que nos permite la construcción de funciones *currificadas* que reciben como argumento conjuntos. Esto es, si $A_i, A_j, A_k : \star^s$, y $f : A_j \to A_k$ ya es una función, tenemos que:

$$(A_i \to (A_j \to A_k)) \quad \equiv \quad (\Pi x : A_i \,.\, f) : \star^f$$

Ejemplo 3.3 (especificación de sistema lógicos como *PTS*):
Así como definimos el sistema *PRED* como un *PTS*, resulta posible también razonar de manera análoga para el resto de los sistemas que conforman el *cubo lógico*. Éstos son:

La lógica de proposiciones se encuentra representada por el *PTS* $\lambda PROP$:

$$\begin{array}{lcl} S & = & \{\star^p, \Box^p\} \\ A & = & \{(\star^p, \Box^p\} \\ R & = & \{(\star^p, \star^p, \star^p)\} \end{array}$$

La lógica proposicional de segundo orden se representa a través del sistema $\lambda PROP2$:

$$\begin{array}{lcl} S & = & \{\star^p, \Box^p\} \\ A & = & \{(\star^p, \Box^p\} \\ R & = & \{(\star^p, \star^p, \star^p),\ (\Box^p, \star^p, \star^p)\} \end{array}$$

La lógica proposicional débil de alto orden se identifica con el sistema $\lambda PROP\underline{\omega}$:

$$\begin{array}{lcl} S & = & \{\star^p, \Box^p\} \\ A & = & \{(\star^p, \Box^p\} \\ R & = & \{(\star^p, \star^p, \star^p),\ (\Box^p, \Box^p, \Box^p)\} \end{array}$$

La lógica proposicional de alto orden está dada por el *PTS* $\lambda PROP\omega$:

$$\begin{array}{lcl} S & = & \{\star^p, \Box^p\} \\ A & = & \{(\star^p, \Box^p\} \\ R & = & \{(\star^p, \star^p, \star^p),\ (\Box^p, \star^p, \star^p),\ (\Box^p, \Box^p, \Box^p)\} \end{array}$$

La lógica de predicados de segundo orden puede representarse con el sistema $\lambda PRED2$:

$$\begin{array}{lcl} S & = & \{\star^s, \star^p, \star^f, \Box^s, \Box^p\} \\ A & = & \{(\star^s, \Box^s),\ (\star^p, \Box^p\} \\ R & = & \{(\star^p, \star^p, \star^p),\ (\star^s, \star^p, \star^p),\ (\star^s, \Box^p, \Box^p), \\ & & (\star^s, \star^s, \star^f),\ (\star^s, \star^f, \star^f),\ (\Box^p, \star^p, \star^p)\} \end{array}$$

La lógica de predicados débil de alto orden está dada por $\lambda PRED\underline{\omega}$:

$$\begin{array}{lcl} S & = & \{\star^s, \star^p, \star^f, \Box^s, \Box^p\} \\ A & = & \{(\star^s, \Box^s),\ (\star^p, \Box^p\} \\ R & = & \{(\star^p, \star^p, \star^p),\ (\star^s, \star^p, \star^p),\ (\star^s, \Box^p, \Box^p), \\ & & (\star^s, \star^s, \star^f),\ (\star^s, \star^f, \star^f),\ (\Box^p, \Box^p, \Box^p)\} \end{array}$$

La lógica de predicados de alto orden finalmente se identifica por el sistema $\lambda PRED\omega$:

$$\begin{array}{lcl} S & = & \{\star^s, \star^p, \star^f, \Box^s, \Box^p\} \\ A & = & \{(\star^s, \Box^s),\ (\star^p, \Box^p\} \\ R & = & \{(\star^p, \star^p, \star^p),\ (\star^s, \star^p, \star^p),\ (\star^s, \Box^p, \Box^p), \\ & & (\star^s, \star^s, \star^f),\ (\star^s, \star^f, \star^f),\ (\Box^p, \Box^p, \Box^p),\ (\star^s, \Box^p, \Box^p)\} \end{array}$$

Todos éstos sistemas, además del $\lambda PRED$ que ya habíamos enunciado, forman lo que se conoce como el *L-cubo*, que resulta ser una estructura idéntica al *cubo lógico*, con la salvedad de que para cada vértice de éste último, el sistema lógico L_i se reemplaza por el sistema λL_i. $*$

Antes de dar a conocer los sistemas de tipos del ejemplo anterior, habíamos dicho que cada sistema del *cubo lógico* se correspondía con un sistema del $\lambda-cubo$. No obstante hemos empleado sistemas de *PTS* en los que ocurren *sorts* que hasta entonces no habíamos empleado (como el $\star^p$, por ejemplo), de modo que claramente estos nuevos sistemas de tipos contienen más información que los pertenecientes al $\lambda-cubo$ original definido en la sección 2.2. De todas formas se puede demostrar, para los sistemas lógicos del *cubo lógico* al menos, que empleando un mapeo que va de $\{\star^s, \star^p, \star^f, \Box^s, \Box^p\}$ en $\{\star, \Box\}$ dado por:

$$\begin{array}{lcl} \star^s & \mapsto & \star \\ \star^p & \mapsto & \star \\ \star^f & \mapsto & \star \\ \Box^s & \mapsto & \Box \\ \Box^p & \mapsto & \Box \end{array}$$

los resultados y propiedades sobre los tipos de los sistemas se conservan, aunque claramente en el proceso estamos descartando parte de la información que contenían los *sorts* originales.

3.3.2 Construyendo tipos y definiendo operadores con los *PTS*

Cuando enunciamos la semántica de Heyting en la sección 3.1.1 dijimos que los conectores lógicos como la implicación, conjunción, disyunción y cuantificadores entre otros eran representados a través de estructuras de datos como los pares y la unión disjunta. Más tarde, cuando introdujimos la noción de tipos dependientes en la sección 3.2.1, observamos que algunos de ellos podían ser empleados para definir no sólo a estos operadores lógicos, sino además algunos tipos simples como por ejemplo el producto cartesiano.

Es de esperarse entonces que, dado el hecho de que los sistemas que conforman el *cubo lógico* tiene su representación como *PTS*, exista una manera de definir dichos tipos dependientes en cada uno de los componentes de *L-cubo* y de esa manera contar con un mecanismo para construir expresiones lógicas más complejas. En la sección 3.3.1 ya vimos como algunas de las reglas que definimos para el sistema *PRED* nos permitía modelar algunas particularidades del sistema. Lo que haremos ahora es dar una breve idea de como se pueden llegar a construir tipos simples para la lógica proposicional de segundo orden, es decir, para *PROP*2, empleando $\lambda 2$ en su versión como *PTS* (tal y como fue definida en el ejemplo 2.16 del capítulo anterior). Algunos de los tipos definibles en este sistema son:

Booleanos Definimos el tipo *Bool* por:

$$\Pi X : \star . X \to X \to X$$

donde:

$$\begin{array}{lcl} True & = & \lambda X : \star . \lambda x : X . \lambda y : X . x \\ False & = & \lambda X : \star . \lambda x : X . \lambda y : X . y \end{array}$$

más aún, si $u, v : U$ y $t : Bool$ el condicional if puede describirse como:

$$if\ u\ v\ t \quad = \quad t\ U\ u\ v$$

y resulta fácilmente demostrable, por ejemplo que:

$$\begin{array}{rcl} if\ u\ v\ True & = & (\lambda X : \star\,.\,\lambda x : X\,.\,\lambda y : X\,.x)\ U\ u\ v \\ & \rightsquigarrow & (\lambda x : U\,.\,\lambda y : U\,.x)\ u\ v \\ & \rightsquigarrow & (\lambda y : U\,.\,u)\ v \\ & \rightsquigarrow & u \end{array}$$

Producto Definimos el producto cartesiano de elementos U y V por:

$$U \times V \quad = \quad \Pi X : \star\,.\,(U \to V \to X) \to X$$

donde si $u : U$ y $v : V$, tenemos que:

$$\langle u, v \rangle \quad = \quad \lambda X : \star\,.\,\lambda x : (U \to V \to X)\,.x\ u\ v$$

y si $t : U \times V$, las proyecciones se encuentran definidas como:

$$\pi^1\ t\ =\ t\ U\ (\lambda x : U\,.\,\lambda y : V\,.x) \qquad \pi^2\ t\ =\ t\ V\ (\lambda x : U\,.\,\lambda y : V\,.y)$$

y trivialmente se puede corroborar ésto. Por ejemplo, su $u : U$ y $v : V$, tenemos que:

$$\begin{array}{rcl} \pi^1\ \langle u, v \rangle & = & (\lambda X : \star\,.\,\lambda x : (U \to V \to X)\,.x\ u\ v)\ U\ (\lambda x : U\,.\,\lambda y : V\,.x) \\ & \rightsquigarrow & (\lambda x : U \to V \to U\,.x\ u\ v)\ (\lambda x : U\,.\,\lambda y : V\,.x) \\ & \rightsquigarrow & (\lambda x : U\,.\,\lambda y : V\,.x)\ u\ v \\ & \rightsquigarrow & (\lambda y : V\,.\,u)\ v \\ & \rightsquigarrow & u \end{array}$$

cuando dimos la semántica de Heyting, comentamos que la conjunción podía expresarse como un producto cartesiano. De ese modo, bajo el sistema *PROP*2 del *L-cubo* asumiendo que A y B son proposiciones (esto es $A, B\ :\ \star^p$) podemos definir esta operación como:

$$A \wedge B \quad = \quad \Pi\gamma : \star^p\,.\,(A \to B \to \gamma) \to \gamma$$

Tipo vacío Podemos definir el tipo vacío como:

$$Empty \quad = \quad \Pi X : \star\,.\,X$$

donde:

$$\varepsilon_U\ t = t\ U$$

más aun, podemos emplear este tipo para definir el operador $\bot$ como:

$$\bot \quad = \quad \Pi\beta : \star^p\,.\,\beta$$

Tipo suma El tipo suma nos permite definir la unión disjunta. Esto es, si U, V son tipos, definimos:

$$U + V \quad = \quad \Pi X : \star \,.\, (U \to X) \to (V \to X) \to X$$

y para construir elementos de este tipo, si suponemos que $u : U$ y $v : V$, definimos $\iota^1\ u$ y $\iota^2\ v$ de tipo $U + V$ como:

$$\begin{array}{rcl} \iota^1\ u & = & \lambda X : \star \,.\, \lambda x : (U \to X) \,.\, \lambda y : (V \to X) \,.\, x\ u \\ \iota^2\ v & = & \lambda X : \star \,.\, \lambda x : (U \to X) \,.\, \lambda y : (V \to X) \,.\, y\ v \end{array}$$

Empleando esta noción de unión disjunta sobre el sistema *PROP*2, resulta factible definir el operador lógico $\vee$ sobre dos proposiciones $A, B \ : \ \star^p$ como:

$$A \vee B \quad = \quad \Pi \gamma : \star^p \,.\, (A \to \gamma) \to (B \to \gamma) \to \gamma$$

Tipo existencial Si V es un tipo y X es una variable de tipo, entonces podemos definir los tipos Σ como:

$$\Sigma X.V \quad = \quad \Pi Y : \star \,.\, (\Pi X : \star \,.\, V \to Y) \to Y$$

por lo que si U es un tipo y v es un término de tipo $V[U/X]$, entonces definimos a $\langle U, v \rangle$ de tipo $\Sigma X.V$ por:

$$\langle U, v \rangle \quad = \quad \lambda Y : \star \,.\, \lambda x : (\Pi x : X \,.\, V \to Y) \,.\, x\ U\ v$$

Nótese que estamos expresando la misma idea que denota el operador lógico $\exists$, ya que el término $\langle U, v \rangle$ primero recibe el tipo del valor de retorno, tras lo cual toma como argumento una función que toma un tipo (el tipo del cual puede depender la segunda componente V, motivo por el cual afirmamos que $v : V[U/X]$, es decir, las ocurrencias de X en el tipo V pasan a quedar fijadas por el argumento de la función x) y retorna una función que al ser aplicada a un término de tipo V (con las ocurrencias del tipo X sustituidas por el tipo U) devuelve un elemento que se corresponde con el tipo que habíamos tomado en primer lugar.

Debido a ello, asumiendo que $A : \star p$ y $S : \star^s$, resulta factible definir la cuantificación existencial bajo el sistema *PROP*2 como:

$$\exists x : s \,.\, A \quad = \quad \Pi \gamma : \star^p \,.\, (\Pi x : S \,.\, (A \to \gamma)) \to \gamma$$

Aquí sólo dimos una breve reseña de la manera en que se pueden definir los tipos necesarios para expresar algunas de las construcciones lógicas más usualmente empleadas en razonamientos de la lógica proposicional. De todas maneras, el poder de expresividad de los sistemas de tipos va un poco más allá y resulta posible definir otros tipos simples que, si bien no son fundamentales en la formación de conectores lógicos resultan de gran utilidad a la hora de trabajar con sistemas basados en el cálculo lambda.

Ejemplo 3.4 (tipos inductivos en el sistema $\lambda 2$):
Algunos de los tipos inductivos más comúnmente empleados son:

- Los números enteros pueden ser definidos como:

$$\text{Int} \quad = \quad \Pi X : \star . X \to (X \to X) \to X$$

y tomando las funciones básicas:

$$\begin{array}{rcl} 0 & : & X \\ \text{S} & : & X \to X \end{array}$$

podemos representar números enteros como por ejemplo:

$$\begin{array}{rcl} \overline{0} & = & \lambda X : \star . \lambda x : X . \lambda y : (X \to X) . x \\ \overline{3} & = & \lambda X : \star . \lambda x : X . \lambda y : (X \to X) . y(y(y\,x)) \end{array}$$

- El tipo lista de elementos de tipo U puede definirse como:

$$\text{List}\, U \quad = \quad \Pi X : \star . X \to (U \to X \to X) \to X$$

contando con dos funciones, una que representa la lista vacía y otra que dado un elemento y una lista construye una nueva lista anexando dicho elemento al inicio de lista recibida, podemos definir:

$$\begin{array}{rcl} \text{nil} & = & \lambda X : \star . \lambda x : X . \lambda y : (U \to X \to X) . x \\ \text{cons}\, u\, l & = & \lambda X : \star . \lambda x : X . \lambda y : (U \to X \to X) . y\, u\, (l\, X\, x\, y) \end{array}$$

de manera que la lista $[x_1, \ldots, x_n]$ se representa como:

$$\lambda X : \star . \lambda x : X . \lambda y : (U \to X \to X) . y\, x_1\, (y\, x_2 \ldots (y\, x_n\, x) \ldots)$$

- El tipo de los árboles binarios se puede especificar como:

$$\text{Bintree} \quad = \quad \Pi X : \star . X \to (X \to X \to X) \to X$$

donde si pensamos que contamos con una función que construye árboles vacíos y otra que dados dos arboles construye uno nuevo, podemos definir:

$$\begin{array}{rcl} \text{empty} & = & \lambda X : \star . \lambda x : X . \lambda y : (X \to X \to X) . x \\ \text{join}\, u\, v & = & \lambda X : \star . \lambda x : X . \lambda y : (X \to X \to X) . y\, (u\, X\, x\, y)\, (v\, X\, x\, y) \end{array} \quad *$$

Para el caso particular que acabamos de ver, el sistema que define tipos para $\lambda 2$ se conoce con el nombre de *Sistema F* [Gir71]. De manera análoga, existen sistemas similares para otros de los sistemas que componen el $\lambda-$*cubo*, como lo son:

- Para $\lambda\!\to$ tenemos el sistema de términos λ^{τ}
- Para λP contamos con los sistemas *AUT-QE* y *LF*
- Para $\lambda\underline{\omega}$ se encuentra definido el sistema *POLYREC*
- Para $\lambda\omega$ existe el sistema $F\omega$
- Para λC el sistema relacionado a éste es el *CC*

Si el lector se encuentra interesado en efectuar un estudio mas detallado de alguno de éstos sistemas, puede consultar las referencias citadas para cada sistema de tipos del capítulo 2. Con ésto ya damos por cerrado el estudio de las bases formales sobre las cuales se sustenta nuestro prototipo de herramientas. Resulta cierto que en la mayoría de los temas analizados en los últimos dos capítulos no efectuamos un análisis formal y detallado de los mismos, pero nuestra intención es simplemente la de brindar una mera noción de todos los conceptos que emplearemos de aquí en adelante y como éstos se relacionan entre sí. Ahora que ya hemos expuesto una idea general del área en la que se enmarca nuestro desarrollo, estamos en condiciones de avanzar en el estudio de la aplicación desarrollada, tema que abarcará nuestros próximos dos capítulos.

Capítulo 4

Hacia un asistente

En este capítulo nos ocuparemos de definir los componentes y algoritmos que rigen el comportamiento del asistente que se ha implementado. Para lograr dicho cometido, primero comenzaremos por construir los componentes básicos que conforman la lógica empleada en la herramienta. Esto incluye estructuras tales como las expresiones, contextos y juicios, las cuales mantienen una estrecha relación con las definiciones y resultados obtenidos en los capítulos 2 y 3.

Una vez que hayamos establecido los bloques principales, estaremos en condiciones de definir estructuras más complejas como lo son las derivaciones y procedimientos como el de generación de variables e incógnitas, para luego avanzar sobre la lógica del comportamiento del asistente que abarca temas como la eliminación de juicios o la generación y manejo de restricciones sobre pruebas.

Durante este capítulo, nuestro objetivo es el de dejar en claro los conceptos y estructuras que manipula nuestro prototipo de asistente. Si bien la implementación de la herramienta sigue los lineamientos de las ideas que daremos a continuación, el lector no debería esperar encontrar código de nuestra implementación en éste capítulo, sino más bien las ideas que condujeron al desarrollo del mismo.

Un punto que vale la pena destacar es que la mayor parte del tiempo estaremos manipulando expresiones incompletas. Es decir, si nuestro trabajo se limitase sólo a un problema de *type checking*, por ejemplo, no tendríamos más que verificar que las reglas de tipado están correctamente aplicadas a expresiones completas. Pero como nos enfrentamos a la tarea de construir derivaciones al mismo tiempo que verificamos su consistencia, la mayor parte del tiempo estaremos intentando verificar la corrección de derivaciones incompletas, de modo que debemos tener en cuenta este inconveniente a medida que definimos los componentes que conformarán nuestra aplicación.

Ejemplo 4.1 (derivaciones incompletas):

- En el sistema $\lambda\!\rightarrow$, si intentaramos verificar si el tipo de la identidad $\alpha \rightarrow \alpha$ se encuentra habitado, resultaría lógico pensar que podemos encontrar un elemento de dicho tipo si empleamos la regla de *abstraction*, de modo que en un primer paso en la búsqueda de nuestra derivación nos encontraremos con algo similar a:

$$\frac{[\alpha : \star,\, x : \alpha] \vdash ?_1 \;:\; \alpha \qquad [\alpha : \star] \vdash\; \alpha \rightarrow \alpha \;:?_2}{[\alpha : \star] \vdash\; \lambda x : \alpha \,.\, ?_1 \;:\; \alpha \rightarrow \alpha}\;(\text{Ab})$$

y aunque aún no hemos determinado que valores tomarán $?_1$ y $?_2$, podemos estar seguros que $?_2$ debe ser un *sort*.

- Algo más radical podría ocurrir si nos embarcamos en la tarea de encontrar un término de tipo $\star$ bajo el sistema de tipos $\lambda\omega$. Si pensamos que podemos llegar a encontrar dicho término empleando la regla de *Product*, nos daríamos con que en un primer momento la derivación que estamos construyendo sería similar a:

$$\frac{[] \vdash ?_1 \ : ?_3 \qquad [x :?_1] \vdash ?_2 \ : ?_4}{[] \vdash \Pi x :?_1 \,.\, ?_2 \ : \star} \ (\mathrm{Pr})$$

  donde más tarde deberemos verificar que $?_3$ y $?_4$ son *sorts* y que además $(?_3, ?_4, \star)$ es una regla válida en $\lambda\omega$. $*$

4.1 Especificando los *PTS*

Lo primero que haremos será dejar en claro la noción de *PTS* que manejaremos en nuestra herramienta, la cual no variará demasiado de la dada en la sección 2.3.2. Pero antes de dar la definición de lo que consideraremos una especificación de *PTS*, debemos introducir una constante que nos será de gran utilidad más adelante.

Definición 4.1 (anysort).
Definimos un nuevo símbolo, $\diamond$, para denotar a una constante especial que denominaremos *anysort*. Dado un conjunto de constantes C, definimos $C_\diamond = C \cup \{\diamond\}$ †

Definición 4.2 (igualdad de anysorts).
Dado un conjunto de constantes C, definimos la relación de equivalencia $=_\diamond$ sobre $C_\diamond \times C_\diamond$ por:

$$c_1 =_\diamond c_2 \quad \Leftrightarrow \quad c_1 = c_2 \vee c_1 = \diamond \vee c_2 = \diamond$$ †

Notación 4.1:
Emplearemos el subíndice A para denotar elementos referidos al ámbito de nuestro asistente, para así poder compararlos y al mismo tiempo diferenciarlos de las definiciones que dimos para los *PTS*. A modo de ejemplo, las expresiones de los *PTS* las denotábamos por E, mientras que las definidas para nuestro prototipo de asistente serán E_A. $\diamondsuit$

Definición 4.3 (especificación de un *PTS*).
Una especificación de un Sistema de Tipos Puros es una tupla (S_A, A_A, R_A) donde:

- S_A es un subconjunto no vacío del conjunto de constantes C, tal que $\diamond \notin S_A$. A los elementos de este conjunto los denominaremos *sorts*.

- A_A es un subconjunto no vacío de $S_A \times S_A$, a cuyos elementos llamaremos *axiomas*.
- R_A es un subconjunto no vacío de $S_A \times S_A \times S_A$, a cuyos elementos llamaremos *reglas*. †

Notación 4.2:

- Dados los conjuntos S_A, A_A y R_A, a menudo denotaremos la especificación (S_A, A_A, R_A) por $\lambda_{S_A A_A R_A}$
- A los elementos de S_A los denotaremos por $s, s_1, s_2, \ldots$
- A los elementos de A_A los denotaremos por $a, a_1, a_2, \ldots$
- A los elementos de R_A los denotaremos por $r, r_1, r_2, \ldots$ ♢

Cuando avancemos en el estudio de las estructuras que manipula nuestra herramientas, estaremos interesados en consultar periódicamente si un *sort*, *axioma* o *regla* pertenece o no a la especificación de un *PTS* dado. Claro que como nos encontraremos manipulando expresiones parcialmente construidas, muy frecuentemente los sorts, axiomas o reglas por los que consultaremos no se encontraran completamente definidos. Por tal motivo, resulta de interés dejar en claro cuando consideraremos que un sort, axioma o regla pertenece (o puede pertenecer) a la especificación de un *PTS*.

Definición 4.4 ($\in_S$, $\in_A$, $\in_R$).

- Definimos la relación $\in_S$ sobre $C_\diamond \times (S_A, A_A, R_A)$ por:

$$c \in_S (S_A, A_A, R_A) \quad \Leftrightarrow \quad c \in S_A$$

- Definimos la relación $\in_A$ sobre $(C_\diamond, C_\diamond) \times (S_A, A_A, R_A)$ por:
  - $(\diamond, \diamond) \in_A (S_A, A_A, R_A), \forall S_A, A_A, R_A$.
  - $(c, \diamond) \in_A (S_A, A_A, R_A) \quad \Leftrightarrow \quad \exists s \in S_A$ tal que $(c, s) \in A_A$.
  - $(\diamond, c) \in_A (S_A, A_A, R_A) \quad \Leftrightarrow \quad \exists s \in S_A$ tal que $(s, c) \in A_A$.
  - $(c_1, c_2) \in_A (S_A, A_A, R_A) \quad \Leftrightarrow \quad (c_1, c_2) \in A_A$.

  donde $\{c, c_1, c_2\} \cap \{\diamond\} = \varnothing$.
- Definimos la relación $\in_R$ sobre $(C_\diamond, C_\diamond, C_\diamond) \times (S_A, A_A, R_A)$ por:
  - $(\diamond, \diamond, \diamond) \in_R (S_A, A_A, R_A), \forall S_A, A_A, R_A$.
  - $(c, \diamond, \diamond) \in_R (S_A, A_A, R_A) \quad \Leftrightarrow \quad \exists s_1, s_2 \in S_A$ tales que $(c, s_1, s_2) \in R_A$.
  - $(\diamond, c, \diamond) \in_R (S_A, A_A, R_A) \quad \Leftrightarrow \quad \exists s_1, s_2 \in S_A$ tales que $(s_1, c, s_2) \in R_A$.
  - $(\diamond, \diamond, c) \in_R (S_A, A_A, R_A) \quad \Leftrightarrow \quad \exists s_1, s_2 \in S_A$ tales que $(s_1, s_2, c) \in R_A$.
  - $(c_1, c_2, \diamond) \in_R (S_A, A_A, R_A) \quad \Leftrightarrow \quad \exists S_A \in s$ tales que $(c_1, c_2, s) \in R_A$.
  - $(c_1, \diamond, c_2) \in_R (S_A, A_A, R_A) \quad \Leftrightarrow \quad \exists S_A \in s$ tales que $(c_1, s, c_2) \in R_A$.
  - $(\diamond, c_1, c_2) \in_R (S_A, A_A, R_A) \quad \Leftrightarrow \quad \exists S_A \in s$ tales que $(s, c_1, c_2) \in R_A$.

– $(c_1, c_2, c_3) \in_R (S_A, A_A, R_A) \quad \Leftrightarrow \quad (c_1, c_2, c_3) \in R_A.$

donde $\{c, c_1, c_2, c_3\} \cap \{\diamond\} = \varnothing$. †

Además de indagar si un axioma o regla parcialmente formada se encuentra definida o no en una especificación de *PTS*, también estaremos interesados en determinar cuando una especificación se encuentra contenida dentro de otra. Para ello encontramos conveniente dar la siguiente definición.

Definición 4.5 (inclusión de especificaciones de *PTS*).
Definimos la inclusión de especificaciones de *PTS* como la relación $\subseteq_{PTS}$ definida sobre $(S_A, A_A, R_A) \times (S_A, A_A, R_A)$ tal que:

$$(S_A^1, A_A^1, R_A^1) \subseteq_{PTS} (S_A^2, A_A^2, R_A^2)$$

si y sólo si

$$\forall s \in S_A^1, a \in A_A^1, r \in R_A^1 \quad s \in S_A^2 \wedge a \in A_A^2 \wedge r \in R_A^2$$ †

Con ésto ya nos alcanza para trabajar con las especificaciones de *PTS* que manipulamos en nuestro prototipo de herramienta, de modo que ya nos encontramos en condiciones de comenzar a definir los componentes básicos que conformarán la base de nuestra aplicación.

4.2 Expresiones y contextos

En ésta sección nos abocaremos al estudio de los componentes básicos que conforman nuestro sistema y sobre los cuales se sustenta toda la lógica y algoritmos que expondremos más adelante. esto incluye fijar ideas sobre lo que consideraremos una variable, una expresión, un contexto y los distintos tipos de sustituciones con que trabajaremos.

4.2.1 Expresiones

Antes de dar a conocer nuestra definición de lo que consideraremos una expresión válida en nuestra herramienta, debemos fijar algunas ideas sobre elementos aún más simples, que serán empleados en futuras definiciones. Comenzaremos por dar una idea de lo que nosotros llamaremos, a partir de este momento, identificadores. Los identificadores serán las construcciones más pequeñas con que trabajaremos y nos permiten asignar un nombre a un valor dado. Nosotros consideraremos dos clases distintas de identificadores.

Definición 4.6 (variables).
Definimos el conjunto de identificadores creados por el usuario por I_U. Además, definimos un nuevo conjunto de identificadores I_G, tales que $I_U \cap I_G = \varnothing$. Diremos que los elementos de I_G son identificadores generados de manera automática por el sistema.

Definimos el conjunto de *variables* $V_A = I_U \cup I_G$. †

Nótese que al pedir que I_U y I_G sean disjuntos, nos estamos asegurando que los identificadores que requiera generar el sistema no ocurrirán en la derivación sobre la cual se está trabajando, ya que por un lado la herramienta puede conocer los identificadores que ya ha generado y por otro lado ninguno de los que genere coincidirá con uno definido por el usuario.

Ahora que ya contamos con una noción de lo que llamaremos identificadores y variables, nuestro siguiente paso será brindar una idea de lo que llamaremos expresiones. Una de las diferencias respecto a la definición 2.28 de expresiones para *PTS* que dimos en el capítulo 2, resulta en el hecho de que la definición de entonces nos resultaba útil para definir expresiones completas. Pero como establecimos al iniciar este capítulo, en todo instante estaremos usando expresiones incompletas y necesitaremos reflejar esto al dar nuestra definición de expresiones.

Definición 4.7 (expresiones).
El conjunto de expresiones E_A se construye a partir de la siguiente gramática:

$$E_A ::= V_A \mid S_A \mid E_A\ E_A \mid \lambda V_A : E_A \,.\, E_A \mid \Pi V_A : E_A \,.\, E_A \mid ?_i\big[[E_A/V_A]\big]$$

donde el subíndice i nos permite identificar cada una de las incógnitas, de modo que pedimos que $i \in \mathbb{N}$. Además $\big[[E_A/V_A]\big]$ denota una lista de sustituciones de variables por expresiones, de la forma:

$$\big[[{E_A}_1^1/x_1^1, \ldots, {E_A}_{n_1}^1/x_{n_1}^1], \ldots, [{E_A}_1^m/x_1^m, \ldots, {E_A}_{n_m}^m/x_{n_m}^m]\big] \qquad \dagger$$

Como se puede apreciar, la única diferencia que presenta esta nueva definición respecto a la 2.28, resulta en la aparición de una nueva clase de expresiones, las de la forma $?_i\big[[E_A/V_A]\big]$. Llamaremos a éstas expresiones *incógnitas* y serán las encargadas de denotar aquellos sectores de nuestras pruebas que aún no han sido completamente definidos.

Ejemplo 4.2 (unknown):
Si empleando la especificación de *PTS* del sistema $\lambda\!\rightarrow$ estuviéramos interesados en buscar un termino de tipo $\Pi x : \alpha \,.\, \alpha$, en un primer momento sólo contaríamos con el juicio:

$$\alpha : \star \vdash ?_0\big[[]\big] \ : \ \Pi x : \alpha \,.\, \alpha \qquad *$$

Viendo el ejemplo anterior, luego de analizar la situación, puede que deduzcamos que la incógnita puede ser suplantada por una abstracción y a partir de allí continuar con el resto de la prueba. De todas maneras, debido a la naturaleza del problema para el cual fue pensada la herramienta, en la mayoría de los casos *a priori* en primeros estadios del desarrollo de la prueba, varias de las expresiones involucradas resultaran desconocidas para el usuario, de allí parte nuestra necesidad de definir las *incógnitas*.

Otro aspecto a destacar es la presencia de la lista que acompaña a las incógnitas. Éstas en realidad no son más que un simple mecanismo que le permite a las *incógnitas* recordar cualquier sustitución de variable que se haya efectuado en el ámbito en el cual se encuentra. Puede que la necesidad de dicho artilugio no resulte claro en estos momentos, sin embargo su necesidad se hará evidente cuando definamos la sustitución de expresiones un poco más adelante.

Notación 4.3:
Emplearemos letras minúsculas $(b, c \ldots)$ para denotar elementos de V_A y letras mayúsculas $(A, B, \ldots)$ para detonar elementos de E_A. De todas maneras, como ya dijimos, reservaremos las letras s, a y r para denotar elementos de S_A, A_A y R_A respectivamente. ♢

Notación 4.4:
A menudo emplearemos la letra δ para denotar una sustitución y con $\vec{\delta}$ denotaremos una lista de sustituciones. ♢

También estamos interesados en contar con algunas operaciones que nos permitan manipular nuestras expresiones, y algunos de los conceptos que se tornarán recurrentes de aquí en adelante son los de variables libres y ligadas para una expresión de E_A.

Definición 4.8 (variables libres de expresiones).
Definimos la función que retorna las *variables libres* de una expresión como:

$$FV \ : \ E_A \to \mathscr{P}(V_A)$$

la cual queda determinada por:

$$\begin{array}{rcl} FV(v) & = & \{v\} \\ FV(s) & = & \varnothing \\ FV(M\,N) & = & FV(M) \cup FV(N) \\ FV(\lambda v : M\,.\,N) & = & FV(M) \cup (FV(N) - \{v\}) \\ FV(\Pi v : M\,.\,N) & = & FV(M) \cup (FV(N) - \{v\}) \end{array}$$

donde $i \in \mathbb{N}$, $v \in V_A$, $s \in S_A$, $M, N \in E_A$. †

Como una incógnita es un término que eventualmente va a definirse, por la forma de la definición de *variables libres*, puede deducirse que cuando una variable ocurre libre en un término con incógnitas, las mismas serán libres en el término cualquiera sea la eventual definición de sus incógnitas. Será conveniente entonces extender la notación de FV de modo que:

$$FV(?_i[\vec{\delta}]) \ = \ \varnothing$$

Esto nos permitirá seguir expresando que, por ejemplo, $v \in FV(v\ ?_k[\vec{\delta}])$. Pero esta notación no significa efectivamente que se sepa cuales son todas las variables libres de un término con incógnitas. Sólo sabemos que son al menos las retornadas por la aplicación de la función FV.

De manera análoga podemos definir una función que nos retorne las incógnitas que se encuentran presente en un elemento de E_A.

Definición 4.9 (incognitas en expresiones).
Definimos la función que retorna las *incógnitas presentes en una expresión* como:

$$UK \;:\; E_A \to \mathscr{P}(E_A)$$

la cual, asumiendo que:

$$subs = [M_1^1/v_1^1,\ldots,M_{n_1}^1/v_{n_1}^1],\ldots,[M_1^m/v_1^m,\ldots,M_{n_m}^m/v_{n_m}^m]$$

queda determinada por:

$$\begin{array}{rcl} UK(v) & = & \varnothing \\ UK(s) & = & \varnothing \\ UK(M\,N) & = & UK(M)\cup UK(N) \\ UK(\lambda v : M\,.\,N) & = & UK(M)\cup UK(N) \\ UK(\Pi v : M\,.\,N) & = & UK(M)\cup UK(N) \\ UK(?_i\big[subs\big]) & = & \{?_i\big[subs\big]\}\cup(\bigcup_{j=1}^{m}\ \bigcup_{k=1}^{n_j}\ UK(M_k^j)) \end{array} \qquad \dagger$$

Esta función también nos permitirá determinar cuándo una expresión M se encuentra completamente definida, lo que ocurrirá cuando $UK(M) = \varnothing$.

También estaremos interesados en computar una expresión lambda. Ésto es, intentar reducir la misma a una forma normal, digamos su *Weak Head Normal Form*.

Definición 4.10 (weak head normal form).
Sean A, E y E' expresiones lambdas. Una expresión de la forma $(\lambda x : A\,.\,E)\;E'$ es llamada un *redex*. Intuitivamente representa la aplicación de una función $\lambda x : A\,.\,E$ sobre el argumento E', lo que nos da la expresión E, donde las ocurrencias libres de x son sustituidas por E'.

Decimos que una expresión lambda se encuentra en su *Head Normal Form* si la misma no es ni un *redex* ni una abstracción lambda que contiene en su cuerpo una expresión reducible.

Decimos que una expresión lambda se encuentra en su *Weak Head Normal Form* si ésta se encuentra en su *Head Normal Form* o bien se trata de una abstracción lambda. Es decir, que en si misma no se trata de un *redex*. †

Al computar una expresión no estamos interesados en llegar a su forma canónica, sino más bien intentar reducir la misma de manera *lazy*. Esto es, sólo estaremos interesados en evaluar la expresión sin preocuparnos por las subexpresiones de las que podría ésta estar formada.

Definición 4.11 (ejecución de expresiones).
Denotamos la ejecución de una expresión de E_A a través de la función $\dashrightarrow$, la cual lleva elementos de E_A en E_A y se encuentra definida por:

$$\begin{array}{rcl} F\,A & \dashrightarrow & \begin{cases} E' & \text{cuando } F \dashrightarrow \lambda x : B\,.\,E \quad\wedge\quad E[A/x] \dashrightarrow E' \\ G\,A & \text{cuando } F \dashrightarrow G \quad\wedge\quad G \text{ no es una abstracción} \end{cases} \\ E & \dashrightarrow & E \quad \text{en caso contrario} \end{array} \qquad \dagger$$

De esta manera, ya estamos en condiciones de reducir expresiones cuándo así lo requiramos. Otro resultado sobre el cual estaremos particularmente interesados más adelante, será en determinar cuando dos expresiones son α-equivalentes o β-equivalentes.

Se suele denominar una α-conversión (en símbolos $\rightarrow_\alpha$) al renombre de una variable ligada en una abstracción lambda. De todas maneras, al efectuar una α-conversión debemos tener cuidado de no efectuar el renombre por una variable que ocurra en la expresión capturada por la abstracción, sin importar si ésta se encuentra libre o ligada. Dicho de otro modo:

$$\lambda x : M \,.\, N \;\rightarrow_\alpha\; \lambda y : M \,.\, N[y/x] \quad \text{donde } y \notin FV(N)$$

Es necesario pedir que la variable por la cual esta renombrando no ocurra en la expresión capturada porque de no ser así, podrían obtenerse resultados dispares a los esperados. Debido a este posible inconveniente, es que pediremos que la variable y sea nueva, ya que si la expresión N contiene incógnitas, entonces no nos alcanza con pedir que $y \notin FV(N)$.

Ejemplo 4.3 (α-conversión):
Supongamos tener la expresión:

$$\lambda x : A \,.\, y$$

donde claramente tenemos que $y \in FV(y)$. Luego si aplicamos una α-conversión que nos sustituya la variable ligada x por y (asumiendo que x e y tienen el mismo tipo), obtendríamos la expresión:

$$\lambda y : A \,.\, y$$

la cual evidentemente no es equivalente a la expresión original, ya que la primera se corresponde con la función constante que retorna siempre el valor y, miestras que en el segundo caso tenemos la función de identidad para elementos de tipo A. $*$

Notación 4.5:
Dadas dos expresiones P y Q, cuando Q se obtiene de P reemplazando una ocurrencia de $\lambda x : M \,.\, N$ por $\lambda y : M \,.\, N[y/x]$, decimos que Q se obtiene por α-conversión de P. En símbolos:

$$P \;\rightarrow_\alpha\; Q$$

Si Q se obtiene a partir de P aplicando 0 o más α-conversiones, diremos que:

$$P \;\rightarrow_\alpha^*\; Q \qquad \diamondsuit$$

Finalmente, diremos que dos expresiones M y N son α-equivalentes, denotado por $M =_\alpha N$, cuando $M \rightarrow_\alpha^* N$. Notemos que la relación $=_\alpha$ es simétrica, de modo que cuando digamos que $M =_\alpha N$, también estaremos diciendo que $N =_\alpha M$. Esta misma noción de α-equivalencia resulta necesaria en nuestra aplicación, de manera que damos la siguiente definición:

Definición 4.12 (caracterización de α-equivalencia para expresiones).
Para expresiones que contengan *incógnitas* no podemos efectuar caracterización alguna de la relación de α-equivalencia. Pero podemos observar que si las expresiones involucradas no presentan *incógnitas*, entonces podemos caracterizar la relación $=_\alpha$ a través de:

$$
\begin{array}{lcl}
v =_\alpha v & & \\
s =_\alpha s & & \\
M\,N =_\alpha M'\,N' & \Leftrightarrow & M =_\alpha M' \;\wedge\; N =_\alpha N' \\
\lambda x : M\,.\,N =_\alpha \lambda y : M'\,.\,M' & \Leftrightarrow & M =_\alpha N \;\wedge\; N[y/x] =_\alpha N' \\
\Pi x : M\,.\,N =_\alpha \Pi y : M'\,.\,M' & \Leftrightarrow & M =_\alpha N \;\wedge\; N[y/x] =_\alpha N'
\end{array}
\qquad \dagger
$$

De manera similar podemos definir la noción de β-reducción ($\rightarrow_\beta$). Ésta consiste en reducir un *redex*, obteniéndose como resultado la expresión capturada por la abstracción, donde las ocurrencias libres de la variable capturada son sustituidas por la expresión sobre la cual se encuentra aplicada la abstracción. Dicho de otro modo:

$$(\lambda x : A\,.\,M)\,N \quad \rightarrow_\beta \quad M[N/x]$$

Y tal y como ocurrió cuando definimos la α-reducción, si la expresión N se obtiene por 0 o más β-reducciones a partir de la expresión M, diremos que:

$$M \rightarrow_\beta^* N$$

Además denotaremos por $=_\beta$ a la clausura simétrica, reflexiva y transitiva de la relación $\rightarrow_\beta$. La noción de β-equivalencia que emplearemos en nuestra herramienta queda entonces dada por la siguiente definición.

Definición 4.13 (caracterización de β-equivalencia para expresiones).
Caracterizamos la relación de β-equivalencia sobre expresiones de E_A de acuerdo a las siguientes reglas. Cuando nos encontramos frente a un *redex*, la β-equivalencia se encuentra dada por:

$$(\lambda x : M\,.\,N)\,E =_\beta E' \quad \Leftrightarrow \quad N[E/x] =_\beta E'$$

mientras que cuando la expresión misma no es un *redex*, caracterizamos la β-equivalencia por:

$$
\begin{array}{lcl}
\lambda x : M\,.\,N =_\beta \lambda y : M'\,.\,N' & \Leftrightarrow & M =_\beta M' \;\wedge\; N[y/x] =_\beta N' \\
M\,N =_\beta M'\,N' & \Leftrightarrow & M =_\beta N \;\wedge\; M' =_\beta N'
\end{array}
\qquad \dagger
$$

Ahora que ya hemos definido las expresiones con las que trabajaremos en nuestra herramienta junto a algunas de las operaciones más comunes que emplearemos para manipularlas, estamos en condiciones de proceder con el estudio de otro elemento básico que formará parte de nuestras derivaciones. Nos estamos refiriendo a los contextos.

4.2.2 Contextos

Los contextos nos permiten establecer los supuestos que podemos considerar válidos al momento de chequear la correctitud de un juicio que forma parte de una derivación.

Definición 4.14 (contextos).
El conjunto C_A de *contextos* es el conjunto de todas las listas de la forma:

$$[x_1 : E_1 , \ldots , x_n : E_n]$$

donde tenemos que

$$E_1 , \ldots , E_n \ \in \ E_A$$

$$x_1 , \ldots , x_n \ \in \ V_A$$

Además pediremos que $\forall i, j$ tal que $i \neq j$ se satisface que $x_i \neq x_j$. De acuerdo a lo expuesto en el lema 2.1, diremos que un contexto $[x_1 : E_1 , \ldots , x_n : E_n]$ se encuentra *bien formado* cuando:

$$\forall v \ \in \ FV(E_i) \ : \ v \in \{x_1, \ldots, x_{i-1}\}$$

Claro que si algún E_i contiene una incógnita, sólo estamos diciendo que las variables libres de términos completos se encuentran previamente introducidas por el contexto. Esto no impide que un contexto *bien formado* con incógnitas pase a encontrarse *mal formado* luego de efectuar la asignación de un valor a una incógnita. †

Muy a menudo diremos que una variable pertenece o no al dominio de un contexto. Por tal motivo, aunque resulte bastante intuitivo, damos la siguiente definición.

Definición 4.15 (dominio de contextos).
Dado el contexto $\Gamma = [x_1 : E_1 , \ldots , x_n : E_n]$ perteneciente a C_A, definimos el dominio del contexto Γ como $dom(\Gamma) = \{x_1, \ldots, x_n\}$
Además diremos que $(x : E) \ \in \ \Gamma$ cuando $\exists i \ \in \ \{1, \ldots, n\} \ : \ x = x_i \wedge E = E_i$ †

Diremos que un contexto $\Gamma = [x_1 : E_1 , \ldots , x_n : E_n]$ se encuentra *completamente definido* cuando se satisfaga que:

$$\forall E \ \in \ \{E_1, \ldots, E_n\} \ : \ UK(E) = \varnothing$$

Así mismo, estaremos interesados en determinar cuándo un contexto se encuentra contenido dentro de otro. Esto se debe a que si logramos demostrar algo en un contexto, entonces será igualmente posible efectuar el mismo supuesto sobre un contexto idéntico, pero que defina algunos elementos más que el contexto original. En nuestra definición de inclusión de contextos somos un poco menos estrictos y no pedimos que los elementos definidos en un contexto ocurran en el otro, sino que tan solo pedimos que en el contexto mayor puedan hacerse las mismas deducciones que son posibles en el contexto más pequeño.

Definición 4.16 (inclusión de contextos).
Dados dos contextos $\Gamma = [x_1 : M_1, \ldots, x_k : M_k]$ y $\Delta = [y_1 : N_1, \ldots, y_l : N_l]$, diremos que el contexto Γ se encuentra incluido en el contexto Δ, $\Gamma \subseteq \Delta$, si y sólo si se cumplen las siguientes condiciones:

- $k \leq l$
- $\forall i \in \{1,\ldots,k\}, \exists j \in \{1,\ldots l\} : x_i = y_j \wedge N_j \rightarrow_\beta M_i$ †

Lema 4.1 (conservación por inclusión de contextos):
Sean Γ y Δ dos contextos pertenecientes a C_A. Si $\Gamma \subseteq \Delta$, entonces se satisface que:

$$\Gamma \vdash M : N \quad \Rightarrow \quad \Delta \vdash M : N$$ ♠

Prueba Supongamos que Γ y Δ son de la forma:

$$\begin{array}{ll} \Gamma & = [x_1 : A_1, \ldots, x_n : A_n] \\ \Delta & = [y_1 : B_1, \ldots, y_m : B_m] \end{array}$$

Por la definición de inclusión de contextos, como $\Gamma \subseteq \Delta$, sabemos que existen n supuestos del contexto Δ de la forma $\tilde{y_i} : \tilde{B_i}$ tales que:

$$\forall i \in \{1,\ldots,n\} : x_i = \tilde{y_i} \quad \wedge \quad \tilde{B_i} \rightarrow_\beta^* A_i$$

Asumiendo que $\Gamma \vdash M : N$ y aplicando el lema 2.4, si logramos verificar que $\Delta \vdash \Gamma$, entonces podemos deducir que $\Delta \vdash M : N$, que es lo que queremos.
Recordemos que verificar que $\Delta \vdash \Gamma$ significa verificar que:

$$\begin{array}{c} \Delta \vdash x_1 : A_1 \\ \vdots \\ \Delta \vdash x_n : A_n \end{array}$$

Pero además, el lema 2.5 nos dice que:

$$\Delta \vdash x_i : \tilde{B_i} \quad \wedge \quad \tilde{B_i} \rightarrow_\beta^* A_i \qquad \Rightarrow \qquad \Delta \vdash x_i : A_i$$

Sabemos que $\forall i \in \{1,\ldots,n\}\ \Delta \vdash x_i : \tilde{B_i}$ por el lema 2.3. Pero además, por la definición de inclusión tenemos que $\tilde{B_i} \rightarrow_\beta^* A_i$. Luego tenemos que:

$$\forall i \in \{1,\ldots,n\}\ \Delta \vdash x_i : A_i$$

y $\therefore$ $\Delta \vdash \Gamma$

y $\therefore$ $\Delta \vdash M : N$ ■

De esta manera damos por concluido nuestro análisis sobre los contextos y las operaciones que se encuentran definidas sobre éstos.

4.2.3 Sustituciones

La noción de sustitución resulta bastante natural en el área de trabajo del cálculo lambda. Muy a menudo se necesita reemplazar alguna variable por otra expresión y es el mecanismo de sustitución el que hace posible llevar a cabo esta tarea. Nosotros ya hemos hecho uso de la misma en capítulos anteriores sin haberla definido.

De hecho, en este mismo capítulo ya hemos introducido sustituciones en el mismo momento en que definimos las *incógnitas*. No obstante, no resulta del todo claro cuál debería ser el resultado de aplicar una sustitución a una *incógnita* o incluso si existe algún mecanismo para definir una expresión aún desconocida.

Para lograr ambos cometidos, lo que haremos ahora sera definir las dos clases de sustituciones que manejaremos con nuestro asistente. La primera de ellas es la sustitución tradicional, es decir, aquella en la que aplicada a una expresión, reemplazamos todas las ocurrencias libres de ésta por alguna otra expresión. Diremos que éstas efectúan el reemplazo de identificadores.

Definición 4.17 (sustitucion de identificadores).
Diremos que una *sustitución de identificadores* es una lista de la forma:

$$[E_1/x_1, \ldots, E_n/x_n]$$

La cual nos indica por cual expresión debe reemplazarse cada una de las ocurrencias libres de los identificadores dados en la lista. Por lo general emplearemos la letra δ para denotar sustituciones. Supongamos tener la sustitución:

$$\delta = [E_1/x_1, \ldots, E_n/x_n]$$

definimos la aplicación de la misma a una expresión de acuerdo a la siguiente regla:

$$\begin{array}{rcl}
v/\delta & = & \begin{cases} E_i & \text{si } v = x_i \text{ para algún } i \in \{1,\ldots,n\} \\ v & \text{c.c.} \end{cases} \\
s/\delta & = & s \\
(M\,N)/\delta & = & (M/\delta)\,(N/\delta) \\
(\lambda x : M\,.\,N)/\delta & = & \lambda y : (M/\delta)\,.\,(N/[E_1/x_1, \ldots, E_n/x_n, y/x]) \text{ con } y \text{ nueva.} \\
(\Pi x : M\,.\,N)/\delta & = & \Pi y : (M/\delta)\,.\,(N/[E_1/x_1, \ldots, E_n/x_n, y/x]) \text{ con } y \text{ nueva.} \\
?_i[\delta_1,\ldots,\delta_n] & = & ?_i[\delta_1,\ldots,\delta_n,\delta]
\end{array}$$

Definimos además el conjunto de sustituciones de identificadores *Subst* por:

$$Subst \;=\; \{[E_1/x_1, \ldots\, E_n/x_n] \;:\; E_1,\ldots,E_n : E_A \;\wedge\; x_1,\ldots,x_n : V_A\} \qquad \dagger$$

Cuando dimos la definición de sustitución para el caso de la abstracción y el producto cartesiano, pedimos que la variable y fuera *"nueva"*. Ésto significa que la variable debe ser totalmente nueva en el entorno en el cual se está trabajando. Esto se debe a que, como ya dijimos, no nos alcanza con pedir que:

$$y \notin \bigcup_{w \in FV(N)-\{x\}} FV\,(w/\delta)$$

debido a que el comportamiento de la función FV dada en la definición 4.8 no nos garantiza que la elección de y nos capture la ocurrencia de alguna otra variable en

la expresión sobre la cual estamos sustituyendo. Esto se debe a que o bien N o bien alguna de las expresiones E_i pueden contener *incógnitas*, lo que nos restringe el uso de la función FV.

Como se puede apreciar, la única innovación respecto a la sustitución tradicional del cálculo lambda, es la que ocurre al momento de aplicar una sustitución a una expresión desconocida. Dado que aun no conocemos la forma que tomará una *incógnita*, lo único que podemos hacer es almacenar la sustitución que fue aplicada, de manera que una vez que la *incógnita* quede determinado, podremos aplicar a dicha expresión todas las sustituciones que debieran haber sido aplicadas a la misma si ésta se hubiese encontrado definida desde un principio.

Pero como mencionamos anteriormente, también contamos con una segunda clase de sustituciones: las que nos determinan los reemplazos de las *incógnitas* por expresiones.

Definición 4.18 (sustitucion de incógnitas).
Diremos que una *sustitución de incógnitas* es una lista de la forma:

$$[E_1/i_1, \ldots, E_n/i_n]$$

donde $i_1, \ldots, i_n$ se corresponden a números naturales que denotan identificadores de incógnitas. Recordemos que una expresión desconocida la denotábamos como $?_i[subs]$, es ése subíndice i el cual identifica unívocamente a cada incógnita. Por lo general emplearemos la letra $\delta^?$ para denotar sustituciones de incógnitas. Supongamos tener la sustitución:

$$\delta^? = [E_1/i_1, \ldots, E_n/i_n]$$

definimos la aplicación de la misma a una expresión de acuerdo a la siguiente regla:

$$\begin{array}{rcl}
v/\delta^? & = & v \\
s/\delta^? & = & s \\
(M\,N)/\delta^? & = & (M/\delta^?)\,(N/\delta^?) \\
(\lambda x : M\,.\,N)/\delta^? & = & \lambda x : (M/\delta^?)\,.\,(N/\delta^?) \\
(\Pi x : M\,.\,N)/\delta^? & = & \Pi x : (M/\delta^?)\,.\,(N/\delta^?) \\
(?_k[\delta_1,\ldots,\delta_n])/\delta^? & = & \begin{cases} (\ldots((E_j/\hat{\delta_1})/\hat{\delta_2})\ldots)/\hat{\delta_n} & \text{si } k = i_j \text{ para algún } j \in \{1,\ldots,n\} \\ (\ldots((?_k[]/\hat{\delta_1})/\hat{\delta_2})\ldots)/\hat{\delta_n} & \text{c.c.} \end{cases}
\end{array}$$

donde si:

$$\delta_i \quad = \quad [E^i_1/x^i_1, \ldots, E^i_{n_i}/x^i_{n_i}]$$

tenemos que:

$$\hat{\delta_i} \quad = \quad [(E^i_1/\delta^?)/x^i_1, \ldots, (E^i_{n_i}/\delta^?)/x^i_{n_i}]$$

Además definimos el conjunto de sustituciones de incógnitas $Subst^?$ por:

$$Subst^? \;=\; \{[E_1/i_1, \ldots\, E_n/i_n] \;:\; E_1,\ldots,E_n : E_A \wedge i_1,\ldots,i_n : Int\} \qquad \dagger$$

Notación 4.6 (aplicación de listas de sustituciones):
Dado un elemento e para el cual se encuentre definida la aplicación de sustituciones y una lista de sustituciones:

$$[\delta_1, \ldots, \delta_n]$$

a menudo emplearemos la notación:

$$e \,/\, [\delta_1, \ldots, \delta_n]$$

para significar:

$$(\ldots((e \,/\, \delta_1) \,/\, \delta_2)\ldots) \,/\, \delta_n$$

De igual manera, dados elementos $e_1, \ldots, e_n$ para los cuales se encuentra definida la aplicación de sustituciones y sustituciones $\delta, \delta_1, \ldots, \delta_m$, emplearemos la notación:

$$\{e_1, \ldots, e_n\} \,/\delta$$

$$[e_1, \ldots, e_n] \,/\delta$$

$$\{e_1, \ldots, e_n\} \,/[\delta_1, \ldots.\delta_m]$$

$$[e_1, \ldots, e_n] \,/[\delta_1, \ldots.\delta_m]$$

para significar:

$$\{e_1/\delta, \ \ldots\ , e_n/\delta\,\}$$

$$[e_1/\delta, \ \ldots\ , e_n/\delta]$$

$$\{e_1/[\delta_1, \ldots.\delta_m], \ \ldots\ , e_n/[\delta_1, \ldots.\delta_m]\}$$

$$[e_1/[\delta_1, \ldots.\delta_m], \ \ldots\ , e_n/[\delta_1, \ldots.\delta_m]]$$

respectivamente. ♢

No obstante, al efectuar el reemplazo de una incógnita por una expresión, puede ser factible que ocurrencias distintas de la misma incógnita resulten en expresiones diferentes, ya que la secuencia de sustituciones que se debería aplicar en cada caso puede variar de una ocurrencia a la otra.

Ejemplo 4.4 (sustituir incógnitas puede no resultar en expresiones iguales): Supongamos tener la sustitución de incógnitas:

$$\delta^? = [x/1]$$

luego al aplicar dicha sustitución a una *incógnita* con identificador 1, la expresión resultante puede variar de acuerdo a la lista de sustituciones que tenga cada ocurrencia de la *incógnita*. De ese modo, tenemos por ejemplo que:

$$\begin{aligned}
?_1\big[\big]/\delta^? &= x \\
?_1\big[[w/y]\big]/\delta^? &= x \\
?_1\big[[w/x]\big]/\delta^? &= w \\
?_1\big[[w/x][y/w]\big]/\delta^? &= y \\
?_1\big[[w/y,\ y/x], [t/y], [?_1\big[[r/x]\big]/t,\ s/w]\big]/\delta^? &= r
\end{aligned}$$ ∗

Ya con lo visto hasta este momento acerca de expresiones, contextos y sustituciones nos resulta suficiente para comprender las estructuras más complejas que compondrán nuestra herramienta.

4.3 Formulando derivaciones

En esta sección emprenderemos el estudio de las derivaciones, uno de los conceptos principales que manejamos en nuestra herramienta. De todas maneras, antes de poder efectuar tal definición, debemos introducir algunas estructuras y conceptos que nos permitirán lograrlo. Esto se debe a que, dada la naturaleza de nuestro prototipo, la noción usual de derivación que se emplea en el calculo lambda no nos es suficiente. Por tal motivo, primero procederemos a definir algunos conceptos y componentes auxiliares para después ya sí, definir de manera precisa lo que nosotros consideraremos una derivación.

4.3.1 Juicios

En toda derivación del cálculo lambda, las unidades básicas que componen el árbol de derivaciones son los juicios. Éstos pueden ser considerados como una relación sobre $C_A \times E_A \times E_A$, denotando el tipo de una expresión bajo los supuestos de un contexto dado. Definimos el conjunto de juicios que maneja nuestra aplicación como el conjunto J_A y la definición de *juicio* que nosostros manejaremos será la usual. De todas formas, daremos algunas definiciones vinculadas a éstos.

Notación 4.7:
A menudo emplearemos j para denotar elementos de J_A. Frecuentemente también usaremos la letra p para denotar juicios que son premisas de algún otro juicio. ♢

Definición 4.19 (juicio completamente definido).
Dado el juicio $J = \Gamma \vdash A : B$, diremos que J se encuentra completamente definido si se satisfacen:

- $UK(A) = UK(B) = \varnothing$
- $\forall (x:E) \in \Gamma : UK(E) = \varnothing$ †

Con nuestra definición de completamente definido, estamos queriendo significar que dicho juicio no contiene ninguna incógnita, y por lo tanto, está compuesto por expresiones completamente determinadas. De todas formas, que un juicio se encuentre completamente definido no significa que el mismo sea un juicio válido. Para que un juicio nos resulte útil a la hora de trabajar con un *PTS* necesitamos pedir algunas restricciones más, por lo que introducimos el concepto de juicio bien formado de acuerdo a la siguiente definición.

Definición 4.20 (juicio bien formado).
Diremos que un juicio $J = \Gamma \vdash A : B$ se encuentra *bien formado* cuando se satisfaga que:

- El contexto Γ se encuentre bien formado, de acuerdo a lo indicado en la definición 4.14
- $FV(A) \cup FV(B) \subseteq dom(\Gamma)$ †

Además, de manera análoga a lo ocurrido con las expresiones, podemos dar una noción de lo que significa aplicar una sustitución a un juicio.

Definición 4.21 (sustitución aplicada a juicios).
Dado el juicio $\Gamma \vdash A : B$, definimos la aplicación de una sustitución de incógnitas sobre tal juicio como:

$$(\Gamma \vdash A : B)/\delta^? \quad = \quad (\Gamma/\delta^?) \vdash (A/\delta?) : (B/\delta^?)$$

donde si $\Gamma = [x_1 : E_1, \ldots, x_n : E_n]$, $(\Gamma/\delta^?)$ denota al contexto:

$$[x_1 : (E_1/\delta^?), \ldots, x_n : (E_n/\delta^?)] \qquad \dagger$$

Cuando estamos sustituyendo incógnitas, el reemplazo debe efectuarse en todos los elementos que forman parte del juicio. Es decir, resulta evidente que a medida que desarrollamos una derivación, las incógnitas se pueden presentar en cualquier parte de la misma, e incluso, como todas las incógnitas representan la misma expresión aún no determinada, cuando decidimos fijar un valor para una incógnita, dicho reemplazo debe efectuarse en todas las ocurrencias de dicha *incógnita* dentro del árbol de derivaciones.

4.3.2 Restricciones

Aunque aún no hemos dado ninguna definición de lo que consideraremos una derivación, resulta lógico que a medida que construimos una prueba, comienzan a surgir algunas condiciones que deben ser satisfechas con el fin de poder concluir con la demostración. Éstas condiciones surgen de manera natural ante cualquier desarrollo que involucra una prueba formal, y nuestro caso no es ajeno a esta situación.

Nuestro prototipo trabaja bajo el supuesto de que, dado un tipo válido para una especificación de *PTS*, resulta posible encontrar un término que habite dicho tipo aplicando las reglas de tipado para *PTS*, hasta conseguir una derivación que nos conduzca a dicho término, partiendo desde los axiomas definidos para dicho *PTS*. Si se recuerdan las reglas de tipado para *PTS* introducidas en la figura 2.7, veremos que algunas de las reglas establecen ya por sí mismas, alguna condición que debe ser verificada para que dicha regla pueda aplicarse. Un ejemplo de ello es la regla de *conversion* que exige que tanto el juicio resultante como una de sus premisas posean tipos β-equivalentes.

De todas maneras, nuestro interés por las condiciones impuestas en el proceso de construcción de la prueba no sólo se limita a las introducidas por las reglas mismas. Como establecimos al definir el conjunto de expresiones que manipula nuestra herramienta en la sección 4.2.1, existen algunas expresiones que pueden no estar completamente definidas. No obstante, aún si éstas representen incógnitas en nuestra demostración, muy a menudo no resultará posible sustituir las mismas por cualquier expresión arbitraria. Dicho de otro modo, aunque no conozcamos el valor que tomará una incógnita, existen algunos supuestos referidos a la misma.

Estas condiciones pueden referirse a varios aspectos. Por ejemplo, podríamos estar ante una *incógnita* que debería tomar la forma de una variable, un sort o formar parte de un axioma o regla. O también se podría dar el caso en que necesitamos que una incógnita tome la forma de una abstracción, aplicación o producto dependiente, sin conocer completamente las expresiones que formarán parte de la misma. Todas estas condiciones las podemos expresar a través de lo que definimos como *restricciones*.

Definición 4.22 (restricciones).
Dadas las expresiones E, E_1, E_2, E_3 pertenecientes a E_A, definimos el conjunto de restricciones aceptadas por el sistema como las que se corresponden con alguna de las siguientes:

- $\in_S^R (E)$: Pertenencia al conjunto de sorts.
- $\in_A^R (E_1, E_2)$: Pertenencia al conjunto de axiomas.
- $\in_R^R (E_1, E_2, E_3)$: Pertenencia al conjunto de reglas.
- $\in_{V \cup S}^R (E)$: Pertenencia al conjunto de variables o sorts.
- $=^R (E_1, E_2)$: Igualdad sintáctica de expresiones salvo α-conversión.
- $=_\beta^R (E_1, E_2)$: β-igualdad de expresiones.

Definiremos además el conjunto de restricciones *Constr* como:

$$\begin{aligned} Constr \quad = \quad & \{\in_S^R (E) \ : \ E \in E_A\} \cup \\ & \{\in_A^R (E_1, E_2) \ : \ E_1, E_2 \in E_A\} \cup \\ & \{\in_R^R (E_1, E_2, E_3) \ : \ E_1, E_2, E_3 \in E_A\} \cup \\ & \{\in_{V \cup S}^R (E) \ : \ E \in E_A\} \cup \\ & \{=^R (E_1, E_2) \ : \ E_1, E_2 \in E_A\} \cup \\ & \{=_\beta^R (E_1, E_2) \ : \ E_1, E_2 \in E_A\} \end{aligned}$$ †

Aunque se desprende de la definición misma, las restricciones no necesariamente involucran a expresiones desconocidas. Es decir, claramente podríamos tener la restricción de que $v = v$, la cual se satisface trivialmente, pero no deja de denotar una condición válida.

Otro aspecto que debería quedar claro es que las restricciones no representan propiedades que satisface una expresión, sino más bien una condición que se espera que satisfaga la misma una vez que ésta se encuentre completamente definida. Dicho de otro modo, al decir que $E_1 = E_2$, estamos diciendo que las expresiones E_1 y E_2 son idénticas, algo que se satisface incluso si $E_1 = E_2 = ?_1\big[[y/x]\big]$. En cambio, al afirmar que $E_1 =^R E_2$, estamos interesados en que ambas expresiones sean idénticas una vez que ya contienen ocurrencias de *incógnitas*. Nótese que para que $E_1 =^R E_2$ se satisfaga no necesitamos ni siquiera que ambas sean la misma expresión.

Otro punto que vale la pena destacar es que, al momento de declarar restricciones no estamos para nada interesados en la especificación de *PTS* bajo la cual estamos trabajando. Es decir, querremos denotar que una expresión debería cumplir con algún requerimiento particular, pero el mismo resulta independiente del sistema que estemos empleando. Con esto queremos decir que una restricción establece una expresión de deseo, a la cual se le puede asignar un valor de verdad sólo cuando se provee una especificación contra la cual efectuar la evaluación.

Por tal motivo, estamos interesados en describir una función capaz de chequear la validez o no de un conjunto de restricciones frente a un *PTS* dado. Al definir dicha función, debemos tener en cuenta el rango de resultados posibles que se pueden obtener al evaluar una restricción sobre la especificación de un *PTS* en particular. Éstos son:

1. La restricción puede ser verificada como verdadera.
2. La restricción puede ser verificada como falsa.
3. No se puede determinar como verdadera o falsa, pero si se puede resolver de forma parcial, obteniéndose un nuevo conjunto de restricciones más simples.
4. No se puede verificar la restricción como verdadera o falsa, ni tampoco es posible reducir la misma a un conjunto de restricciones más simples.

El primer caso se da cuando se tienen restricciones de la forma:

- $\in^R_{V \cup S} (v)$
- $=^R (\lambda x : \star . x, \lambda x : \star . x)$
- $\in^R_R (\Box, \star, \star)$ bajo la especificación de *PTS* para el sistema λC.
- $=^R (?_1 [], ?_1 [])$

En tales situaciones lo único que deberíamos hacer es retirar la restricción del conjunto de condiciones que queremos verificar, ya que la misma puede ser verificada. La siguiente clase de condiciones son aquéllas que pueden ser verificadas como falsas. En cierta forma serían análogas a las vistas anteriormente, con la salvedad de que difieren en su valor de verdad. Algunos ejemplos de estas clases de restricciones son:

- $\in^R_S (\lambda x :?_1 [] . x)$
- $=^R (\lambda x :?_2 [] . ?_3 [], \Pi x :?_2 [] . ?_3 [])$
- $\in^R_A (\Box, ?_1 [])$ cuando $(\Box, \diamond) \notin_A (S_A, A_A, R_A)$ para la especificación de un *PTS*.

Si en algún momento nos topamos con un conjunto de restricciones donde una de ellas puede ser evaluada en falsa, entonces podemos concluir inmediatamente que tal conjunto nunca va a poder ser satisfecho completamente. La tercer clase de restricciones son aquéllas en las que no resulta factible decidir si son verdaderas o falsas, pero sí es posible reducir la misma a un conjunto de restricciones más simples, el cual claramente puede incluso resultar vacío. Más aún, cuando logramos resolver parcialmente una restricción de equivalencia ($=^R$ o $=^R_\beta$), obtendremos un conjunto de asignaciones a incógnitas que pudieron ser deducidas durante el proceso de verificación de condiciones. Tal vez este concepto nos resulte un poco más claro si vemos algunos ejemplos al respecto.

Ejemplo 4.5 (resolución parcial de restricciones):

- Uno de los casos mas simples de resolución de restricciones es el que se presenta ante la situación:

  $$=^R (?_1 [], x)$$

  El cual nos dice que la incógnita con identificador "1", debería tener el mismo valor que la variable x. Como x ya esta definido, podemos considerar la condición verificada obteniéndose un conjunto vacío de nuevas restricciones para ser chequeadas. Sin embargo, sí obtenemos una asignación que debería ser aplicada al resto de los elementos vinculados a la restricciones resuelta. Tal asignación es $[x/1]$, es decir la que establece que las ocurrencias de las incógnitas con identificador 1 deberían ser sustituidas por la variable x.

- Otra situación que se podría dar es que tengamos la asignación entre incógnitas o expresiones conformandas por incógnitas de la forma:

$$=^R (?_1\,[\,],?_2\,[\,])$$

$$=^R (?_1\,[\,],\lambda x :?_2\,[\,]\,.\,x)$$

  Restricciones de esta forma resultan en un conjunto vacío de nuevas restricciones, pero sí generan sustituciones de incógnitas que deberían ser aplicadas al entorno en el cual se definieron las condiciones. En nuestro caso particular, las sustituciones retornadas serían:

$$[?_1\,[\,]/2] \quad \text{o} \quad [?_2\,[\,]/1]$$

  y

$$[\lambda x :?_2\,[\,]\,.\,x/1]$$

  respectivamente.

- De todas maneras, no siempre obtendremos un conjunto vacío de nuevas restricciones. Un ejemplo de ello puede ser la restricción dada por:

$$=^R ((\lambda x :?_1\,[\,]\,.\,x)\;z,(\lambda x : A\,.\,x)\;?_2\,[\,])$$

  Viendo que la estructura de ambas expresiones son similares, la condición puede ser reducida a un conjunto mas simple dado por:

$$\{=^R (z,?_2\,[\,]),\; =^R (A,?_1\,[\,])\}$$

  sin obtenerse ninguna asignación. De todas formas, las asignaciones obvias que mapean la expresión A a la incógnita 1 y la variable z a la incógnita 2 se pueden obtener al intentar resolver las nuevas condiciones. $*$

Además de éstas tres posibilidades de resolución de restricciones, nos podemos topar con condiciones que no pueden verificarse, ni refutarse, ni ser reducidas a un conjunto de restricciones mas sencillas. esta situación se da en casos como:

- $\in^R_S (?_1\,[\,])$
- $\in^R_R (\Box,?_1\,[\,],?_2\,[\,])$ para la especificación del sistema λC
- $=^R (?_1\,[x/y]\,],x)$

En el primer caso, no podemos determinar el valor de verdad de la restricción debido a que no sabemos qué valor tomará la incógnita en cuestión. Más aún, no resulta posible reducirla a una expresión más simple debido a que no hay ninguna manera de proceder hasta tanto quede definido un valor para la incógnita 1.

En el segundo caso estamos frente a una situación similar. Tenemos la necesidad de que $(\Box,?_1\,[\,],?_2\,[\,])$ se encuentre definido como una regla en la especificación del sistema λC. Pero por un lado, esta información no nos resulta suficiente como para asignarle valores a ambas incógnitas, ya que mediante cualquiera de las asignaciones:

$$\begin{array}{ccccc} ?_1\,[\,] & = & ?_2\,[\,] & = & \star \\ ?_1\,[\,] & = & ?_2\,[\,] & = & \Box \end{array}$$

la restricción se satisface, pero obviamente ambas asignaciones no tendrán el mismo impacto en el entorno de la restricción analizada y claramente cualquier otra asignación provocará que la restricción resulte falsa. Más aún, ésta no puede ser reducida, ya que por ejemplo pedir que $\{\in^R_S (?_1 [\,]),\ \in^R_S (?_2 [\,])\}$ no expresa realmente lo mismo que la restricción original.

Finalmente, nuestro tercer ejemplo se parece mucho a una de las situaciones expuestas para el caso en que podíamos reducir una restricción a un conjunto simplificado que exprese exactamente la misma condición. De todas maneras existe una sutil diferencia entre esta situación y la dada anteriormente. En esta última, la incógnita posee una lista de sustituciones asociada, lo que nos restringe el campo de acciones que podemos llevar a cabo. Por un lado no podemos determinar la veracidad de la condición debido a que uno de los componentes que la integran es una incógnita. Por otro lado tampoco podemos reducir la expresión, debido a que no hay ninguna expresión que pueda serlo y menos extraer una asignación de la misma. Uno se sentiría tentado a decir que podemos reemplazar las ocurrencias de la incógnita 1 por la variable x, pero esto no es cierto, ya que tal vez la incógnita podría ser asignada a la variable y, de modo que para la ocurrencia de $?_1[[x/y]]$, se satisface que $=^R (?_1[[y/x]], x)$. Por tal motivo, no tenemos ninguna manera segura de proceder y sólo nos queda la posibilidad de que alguna otra deducción externa a la restricción que estamos analizando nos arroje algún resultado que nos permita verificar o reducir dicha condición.

Teniendo todas estas posibilidades presentes, procederemos a formalizar una relación que nos permita determinar si un conjunto de restricciones tiene o no solución o en su defecto puede al menos ser reducida a un conjunto más sencillo de restricciones. Pero antes de definirla, daremos algunas funciones auxiliares que establecerán el comportamiento que tendrá la misma ante cada una de las restricciones que hemos definido.

A partir de ahora comenzaremos a manejar funciones para las cuales surge la necesidad de denotar una falta de solución para algunos de los casos que analizan. Para representar este valor especial, introducimos una nueva constante.

Definición 4.23 ($\bowtie$).
Dada una función, cuando su definición le impida retornar un elemento de su imagen, diremos que no existe una solución para dicho caso. Por ese motivo, extendemos la imagen de dicha función, permitiéndole retornar también el símbolo $\bowtie$ que representa la ausencia de una solución en particular.

Dado un conjunto A, definimos $A_\bowtie = A \cup \{\bowtie\}$. †

Notación 4.8 (extensión de $\bowtie$ sobre funciones):
Dada $f : A \to B_\bowtie$ y $a \in A$, denotaremos:

$$f_\bowtie : A_\bowtie \to B_\bowtie$$

a la función dada por:

$$\begin{array}{lcl} f_\bowtie (a) & = & f (a) \\ f_\bowtie (\bowtie) & = & \bowtie \end{array}$$ ♢

El tipo del valor que retornarán las funciones auxiliares que operan para cada uno de los casos será $([Subst^?], \{Constr\})_{\bowtie}$. Esto puede ser visto como una unión disjunta, donde decimos que se retorna o bien un par de la forma $([Subst^?], \{Constr\})$ o bien el elemento $\bowtie$. Este último nos indicará que el conjunto de restricciones no puede ser satisfecho, debido a que una o más condiciones no son verdaderas. Por otro lado, si la función retorna un par, querrá decir que o bien el conjunto de restricciones se pudo satisfacer o bien se obtuvo un nuevo conjunto de condiciones que deberán ser verificadas. Para ser más precisos, los elementos que conforman dicho par representan:

- $[Subst^?]$: Se corresponde a una lista de sustituciones de incógnitas que se ha deducido en el proceso de resolución de restricciones. Pedimos que las sustituciones se retornen en una lista y no un conjunto porque estamos interesados en el orden en que se deben efectuar los reemplazos. Es decir, si tenemos la incógnita $?_1[]$ y el conjunto de asignaciones:

  $$\{[?_2[]/1],\ [x/2]\}$$

  dependiendo del orden en que aplicásemos las sustituciones terminaremos por obtener la expresión x o la expresión $?_2[]$. Es decir, a medida que analicemos un conjunto de restricciones podremos ir deduciendo valores que pueden ser reemplazados por incógnitas. El orden en que vamos efectuando dichas deducciones resulta importante debido a que, una vez que terminemos de analizar el conjunto de restricciones, estaremos interesados en plasmar la información adquirida a través de las deducciones sobre los juicios, expresiones o condiciones vinculadas al conjunto de restricciones analizado.

- $\{Constr\}$: Es el conjunto de restricciones que se obtuvieron al analizar las expresiones. Si la condición que vincula a las expresiones recibidas no se pudo reducir, entonces la misma restricción se retorna aquí para su posterior análisis. Si por el contrario, la restricción original pudo ser reducida, entonces el nuevo conjunto de condiciones derivado de la misma en retornado.

Antes de proceder con la definición de las funciones encargadas de efectuar el chequeo de restricciones, definiremos un nombre para denotar el conjunto de expresiones que representan incógnitas, ya que su uso se tornará recurrente de aquí en adelante.

Definición 4.24 ($?_{\text{SET}}$).
Definimos el conjunto de expresiones de E_A que denotan incógnitas por el conjunto $?_{\text{SET}}$, definido por:

$$?_{\text{SET}} = \left\{?_i[\delta_1, \ldots, \delta_n] \ :\ i \in \mathbb{N} \wedge \delta_1, \ldots, \delta_n \in Subst\right\} \qquad \dagger$$

Definición 4.25 (chequeo de restricciones $\in_S^R$).
Definimos la función:

$$check_{\in_S} \ :\ E_A \times (S_A, A_A, R_A) \to ([Subst^?], \mathscr{P}(Constr))_{\bowtie}$$

dada por:

$$check_{\in_S}(M,PTS) = \begin{cases} ([],\{=^R (M,s)\}) & \text{si } M \in ?_{\text{SET}} \wedge S_A = \{s\} \\ ([],\{\in_S^R (M)\}) & \text{si } M \in ?_{\text{SET}} \wedge S_A \neq \{s\} \\ ([],\varnothing) & \text{si } M \in S_A \\ \bowtie & \text{c.c.} \end{cases} \quad \dagger$$

La función $check_{\in_S}$ se aplicará a expresiones que se encuentren bajo la restricción $\in_S^R$. Si la expresión es una incógnita, entonces se la devuelve para que sea analizada más tarde. Si la expresión es un *sort* que se encuentra definido en la especificación del *PTS*, entonces la misma es verificada como verdadera. Resulta evidente que en cualquier otro caso la expresión no será un *sort* válido, y por lo tanto se retorna un $\bowtie$.

De manera similar, podemos definir las funciones que realizarán el chequeo de pertenencia de una expresión al conjunto de axiomas o reglas para una especificación de *PTS* dada. Pero antes de continuar daremos un mapeo que emplearemos en algunos de los resultados que veremos más adelante.

Definición 4.26 ($\rightarrow \diamond$).
Dada una especificación de *PTS* (S_A, A_A, R_A), definimos el mapeo $\rightarrow \diamond$ como:

$$\rightarrow_\diamond : (S_A \cup ?_{\text{SET}}) \rightarrow (S_A)_\diamond$$

y el cual queda definido por:

$$e_{\rightarrow_\diamond} = \begin{cases} e & \text{si } e \in_S S_A \\ \diamond & \text{c.c.} \end{cases} \quad \dagger$$

Ahora sí, podemos ver cómo resulta definida la función encargada de llevar a cabo el chequeo de restricciones referidas a la posibilidad de que un par de expresiones denoten un elemento definido en el conjunto de axiomas para un *PTS* dado.

Definición 4.27 (chequeo de restricciones $\in_A^R$).
Definimos la función:

$$check_{\in_A} : E_A \times E_A \times (S_A, A_A, R_A) \rightarrow ([Subst^?], \mathscr{P}(Constr))_\bowtie$$

dada por:

$$check_{\in_A}(E_1,E_2,PTS) = \begin{cases} ([],\varnothing) & \text{si } E_1,E_2 \in S_A \wedge (E_1,E_2) \in_A PTS \\ check'_{\in_A}(E_1,E_2) & \text{si } E_1,E_2 \in S_A \cup ?_{\text{SET}} \wedge \{E_1,E_2\} \cap ?_{\text{SET}} \neq \varnothing \\ \bowtie & \text{c.c.} \end{cases}$$

donde:

$$check'_{\in_A}(E_1,E_2) = \begin{cases} \bowtie & \text{si } sols = \varnothing \\ ([],\{=^R(E_1,s_1),\ =^R(E_2,s_2)\}) & \text{si } sols = \{(s_1,s_2)\} \\ ([],\{\in^R_A(E_1,E_2)\}) & \text{c.c.} \end{cases}$$

siendo:

$$sols\ =\ \{(s_1,s_2)\ :\ (s_1,s_2) \in A_A\ \wedge\ s_1 =_\diamond E_{1\to\diamond}\ \wedge\ s_2 =_\diamond E_{2\to\diamond}\} \qquad \dagger$$

Esta nueva función intenta verificar si dos expresiones pueden formar parte o no de un axioma para una especificación de *PTS* dada. Para llevar a cabo esta tarea se procede a efectuar un análisis por casos de las expresiones recibidas como argumento. Si las mismas son *sorts* y además éstas se encuentran definidas como un axioma en el *PTS*, entonces se determina que la restricción de que $\in^R_A(E_1,E_2)$ resulta verdadera.

Por otro lado, si las expresiones están formadas por *sorts* y al menos una incógnita, entonces hay tres nuevos casos por considerar, dependiendo de las posibilidades que existen de que el par de expresiones que estamos analizando pueda llegar a quedar definido como un elementos del conjunto de axiomas del *PTS* sobre el cual estamos trabajando. En efecto, el conjunto *sols* contiene todos los axiomas en que se pueden transformar nuestro par de expresiones cuando éstas resulten completamente definidas.

- Si dicho conjunto resulta vacío, entonces nunca podremos obtener un axioma a partir de las expresiones que estamos analizando y por lo tanto determinamos que la restricción no podrá ser satisfecha.
- Por otro lado, si el conjunto de posibles soluciones tiene un solo miembro, entonces podemos reescribir nuestra restricción como la necesidad de que las expresiones que estamos analizando se correspondan con los únicos *sorts* que pueden llegar a hacer que la condición sea verdadera.
- En cualquier otro caso, existe más de una posibilidad para hacer que nuestra restricción resulte verdadera (o incluso también falsa), de modo que solo podemos dejar nuestra restricción intacta, para que sea verificada más tarde cuando tengamos algo más de información disponible.

Claramente, si ocurre alguna otra situación distinta a las que ya hemos contemplado, podemos decir de inmediato que la restricción no se satisface.

De manera casi idéntica podemos definir la función que se encargará de efectuar el análisis de las restricciones vinculadas a la posibilidad de que una tripla de expresiones pueda identificar o no a una regla definida en la especificación de un *PTS*.

Definición 4.28 (chequeo de restricciones $\in^R_R$).
Definimos la función:

$$check_{\in_R}\ :\ E_A \times E_A \times E_A \times (S_A,A_A,R_A) \to ([Subst^?],\mathscr{P}(Constr))_\bowtie$$

dada por:

$$check_{\in_R}(E_1,E_2,E_3,PTS) = \begin{cases} ([],\varnothing) & \text{si } E_1,E_2,E_3 \in S_A \\ & \quad \wedge\, (E_1,E_2,E_3) \in_R PTS \\ check'_{\in_R}(E_1,E_2,E_3) & \text{si } E_1,E_2,E_3 \in S_A \cup ?_{\text{SET}} \\ & \quad \wedge\, \{E_1,E_2,E_3\} \cap ?_{\text{SET}} \neq \varnothing \\ \bowtie & \text{c.c.} \end{cases}$$

donde:

$$check'_{\in_R}(E_1,E_2,E_3) = \begin{cases} \bowtie & \text{si } sols = \varnothing \\ ([],\{=^R(E_1,s_1),\, =^R(E_2,s_2),\, =^R(E_3,s_3)\}) & \text{si } sols = \{(s_1,s_2,s_3)\} \\ ([],\{\in^R_R(E_1,E_2,E_3)\}) & \text{c.c.} \end{cases}$$

siendo:

$$sols \;=\; \{(s_1,s_2,s_3) \;:\; (s_1,s_2,s_3) \in R_A \;\wedge\; s_1 =_\diamond E_{1\to\diamond} \;\wedge\; s_2 =_\diamond E_{2\to\diamond} \;\wedge\; s_3 =_\diamond E_{3\to\diamond}\}$$

†

No daremos mayores detalles sobre esta última función debido a que su comportamiento es muy similar al de la función $check_{\in_A}$ que definimos anteriormente, ya que la única diferencia entre ambas radica en el hecho de que la primera opera sobre el conjunto de axiomas de una especificación de *PTS* mientras que la segunda lo hace sobre el conjunto de reglas.

De manera análoga a la función $check_{\in_S}$ que dimos en la definición 4.25 podemos también declarar la función que emplearemos para el análisis de las restricciones referidas a la pertenencia de una expresión al conjunto de variables o de *sorts* para un *PTS* específico.

Definición 4.29 (chequeo de restricciones $\in^R_{V\cup S}$).
Definimos la función:

$$check_{\in_{V\cup S}} \;:\; E_A \times (S_A,A_A,R_A) \to ([Subst^?],\mathscr{P}(Constr))_\bowtie$$

dada por:

$$check_{\in_{V\cup S}}(M,PTS) \;=\; \begin{cases} ([],\{\in^R_{V\cup S}(M)\}) & \text{si } M \in ?_{\text{SET}} \\ ([],\varnothing) & \text{si } M \in V_A \\ ([],\varnothing) & \text{si } M \in S_A \\ \bowtie & \text{c.c.} \end{cases}$$

†

Como se puede apreciar, la única diferencia que presenta esta nueva función respecto a $check_{\in_S}$ es que también verifica como verdadera la pertenencia de una expresión al conjunto de variables.

Luego de haber definido estas funciones auxiliares, sólo nos resta dar funciones que se encarguen de verificar las restricciones referidas a la igualdad y β-equivalencia de expresiones, y son éstas las que veremos a continuación.

Definición 4.30 (chequeo de restricciones $=^R$).
Definimos la función:

$$check_= \ : \ E_A \times E_A \times (S_A, A_A, R_A) \to ([Subst^?], \mathscr{P}(Constr))_{\bowtie}$$

dada por:

$$\begin{array}{lcl}
check_=(s_1, s_2, PTS) & = & \begin{cases} ([], \varnothing) & \text{si } s_1 = s_2 \\ \bowtie & \text{c.c.} \end{cases} \\
check_=(v_1, v_2, PTS) & = & \begin{cases} ([], \varnothing) & \text{si } v_1 = v_2 \\ \bowtie & \text{c.c.} \end{cases} \\
check_=(M_1\ N_1, M_2\ N_2, PTS) & = & ([], \{=^R (M_1, M_2), =^R (N_1, N_2)\}) \\
check_=(\lambda x_1 : M_1 \,.\, N_1, \lambda x_2 : M_2 \,.\, N_2, PTS) & = & \begin{cases} ([], set_1) & \text{si } UK(N_1) \neq \varnothing \\ ([], set_2) & \text{c.c.} \end{cases} \\
check_=(\Pi x_1 : M_1 \,.\, N_1, \Pi x_2 : M_2 \,.\, N_2, PTS) & = & \begin{cases} ([], set_1) & \text{si } UK(N_1) \neq \varnothing \\ ([], set_2) & \text{c.c.} \end{cases} \\
check_=(?_i[\,], ?_j[\,], PTS) & = & ([], \varnothing) \quad \text{si } i = j \\
check_=(?_i[\,], M, PTS) & = & ([[M/i]], \varnothing) \quad \text{si } ?_i[\delta_1, \ldots, \delta_n] \notin UK(M) \\
check_=(M, ?_i[\,], PTS) & = & ([[M/i]], \varnothing) \quad \text{si } ?_i[\delta_1, \ldots, \delta_n] \notin UK(M) \\
check_=(?_i[\delta_1, \ldots, \delta_n], M, PTS) & = & ([], \{=^R (?_i[\delta_1, \ldots, \delta_n], M)\}) \quad \text{si } M \neq ?_j[\,] \\
check_=(M, ?_i[\delta_1, \ldots, \delta_n], PTS) & = & ([], \{=^R (?_i[\delta_1, \ldots, \delta_n], M)\}) \quad \text{si } M \neq ?_j[\,] \\
check_=(M, N, PTS) & = & \bowtie \quad \text{para cualquier otro caso}
\end{array}$$

donde:

$$\begin{array}{lcl}
set_1 & = & \{=^R (M_1, M_2), =^R (N_1, N_2[x_1/x_2])\} \\
set_2 & = & \{=^R (M_1, M_2), =^R (N_1[x_2/x_1], N_2)\}
\end{array}$$ †

Veamos algunas consideraciones que deberemos tener en cuenta respecto a esta última definición. Algo que resulta evidente a primera vista es que solo tienen la posibilidad de ser consideradas verdaderas aquellas restricciones que relacionan dos expresiones similares a lo que su estructura respecta. Sin embargo existe un caso particular, el de las *incógnitas*, que además de chequear la equivalencia entre expresiones de la misma clase permite efectuar asignaciones de valores a incógnitas.

Siguiendo el razonamiento de nuestra definición, para determinar si dos variables o *sorts* son equivalentes, no debemos hacer más que chequear la equivalencia entre las mismas empleando la igualdad sintáctica.

Si estamos analizando dos expresiones que representan la aplicación de expresiones, podemos simplificar la condición de que ambas aplicaciones sean equivalentes

verificando si cada una de las componentes que las conforman son equivalentes entre sí.

Al intentar verificar la equivalencia entre dos abstracciones, no solo debemos corroborar que los tipos de las variables abstraídas sean equivalentes, sino que además los términos sobre los cuales estamos abstrayendo sean equivalentes. De todas maneras, dado que el nombre de las variables que estamos abstrayendo puede diferir de una a otra sin alterar el significado en sí de los términos, más que una igualdad sintáctica, estamos interesados en capturar la noción de expresiones α-equivalentes. Es por este motivo que una de las condiciones que se originan a partir de este caso pide que los términos abstraídos sean α-equivalentes, permitiendo el renombre de la variable abstraída. Más aún, debido a que la aplicación de sustituciones a expresiones desconocidas resulta en un anidamiento de sustituciones que dificulta la manipulación de tales *incógnitas*, resulta preferible aplicar el renombre de variables sobre un término completamente definido, si es que uno de los dos términos abstraídos cumple con esta condición. De allí a que decidamos sustituir uno u otro de los términos involucrados. Desde luego que en el caso de que ambas variables abstraídas tengan el mismo nombre, la sustitución que aplicamos se convierte en un renombre de variables trivial que no afecta en nada las expresiones involucradas. Para el caso del producto dependiente, operamos de manera semejante al caso de la abstracción.

Cuando nos enfrentamos al problema de decidir si dos expresiones desconocidas son equivalentes o no tenemos varios casos posibles. Si ambas incógnitas no presentan sustituciones asociadas a ellas y sus identificadores son idénticos, entonces claramente podemos determinar que son equivalentes. Por otro lado, si al menos una de las expresiones involucradas es una incógnita que no contiene sustituciones pendientes para ser aplicadas, entonces podemos descartar la restricción que estamos analizando y decidir que todas las ocurrencias de dicha incógnita pueden ser reemplazadas por la otra expresión que estamos contemplando. Claro que permitimos efectuar tal asignación solamente si la incógnita a la cual le estamos asignando el valor no ocurre normalmente en la expresión por la cual está siendo sustituida.

Evidentemente, si la incógnita que estamos comparando contiene alguna sustitución en espera, no podemos efectuar la asignación de manera directa como en el caso anterior. Es decir, no resulta cierto que la incógnita represente efectivamente a dicha expresión, ya que la presencia de las sustituciones está indicando que una vez que definamos un valor para la incógnita, recién despues de aplicar las sustituciones sobre dicho valor se obtendrá una expresión que debería ser equivalente a la expresión que estamos comparando con la incógnita.

Dicho de otro modo, si tenemos que $=^R (?_1\,[\![y/x]\!], y)$, no podemos asumir directamente que $?_1\,[\!] \equiv x$ o que $?_1\,[\!] \equiv y$, ya que ambas son ciertas. Por tal motivo, no nos queda más remedio que postergar el análisis de este tipo de restricciones. Para cualquier otro caso, asumimos que la restricción no es satisfacible.

Ahora sólo nos resta definir la función que se encargará de procesar las restricciones referidas a la β-equivalencia de expresiones, y es esta función la que veremos a continuación.

Definición 4.31 (chequeo de restricciones $=^R_\beta$).
Definimos la función:

$$check_{=_\beta} \;:\; E_A \times E_A \times (S_A, A_A, R_A) \to ([Subst^?], \mathscr{P}(Constr))_{\bowtie}$$

dada por:

$$
\begin{array}{lcl}
check_{=_\beta}(s_1, s_2, PTS) & = & \begin{cases} ([], \varnothing) & \text{si } s_1 \equiv s_2 \\ \bowtie & \text{c.c.} \end{cases} \\
check_{=_\beta}(v_1, v_2, PTS) & = & \begin{cases} ([], \varnothing) & \text{si } v_1 \equiv v_2 \\ \bowtie & \text{c.c.} \end{cases} \\
check_{=_\beta}(\lambda x_1 : M_1 \,.\, N_1, \lambda x_2 : M_2 \,.\, N_2, PTS) & = & \begin{cases} ([], set_1) & \text{si } UK(N_1) \neq \varnothing \\ ([], set_2) & \text{c.c.} \end{cases} \\
check_{=_\beta}(\Pi x_1 : M_1 \,.\, N_1, \Pi x_2 : M_2 \,.\, N_2, PTS) & = & \begin{cases} ([], set_1) & \text{si } UK(N_1) \neq \varnothing \\ ([], set_2) & \text{c.c.} \end{cases} \\
check_{=_\beta}(?_i[\,], ?_j[\,], PTS) & = & ([], \varnothing) \quad \text{si } i = j \\
check_{=_\beta}(?_i[\delta_1, \ldots, \delta_n], M, PTS) & = & ([], \{=_\beta^R (?_i[\delta_1, \ldots, \delta_n], M)\}) \\
check_{=_\beta}(M, ?_i[\delta_1, \ldots, \delta_n], PTS) & = & ([], \{=_\beta^R (M, ?_i[\delta_1, \ldots, \delta_n])\}) \quad \text{si } M \notin ?_{\text{SET}} \\
check_{=_\beta}(M\,N, M'\,N', PTS) & = & ([], \{=_\beta^R (M, M'), =_\beta^R (N, N')\}) \quad \text{si } M, M' \in S_A \cup V_A \\
 & & \qquad \cup \{(A\ B) : A, B \in E_A\} \\
check_{=_\beta}(A, M\,N, PTS) & = & ([], \{=_\beta^R (A, M\,N)\}) \\
check_{=_\beta}(M, N, PTS) & = & \bowtie \quad \text{para cualquier otro caso}
\end{array}
$$

donde:

$$
\begin{array}{lcll}
set_1 & = & \{=_\beta^R (M_1, M_2), =_\beta^R (N_1, N_2[x_1/x_2])\} & \\
set_2 & = & \{=_\beta^R (M_1, M_2), =_\beta^R (N_1[x_2/x_1], N_2)\} & \dagger
\end{array}
$$

A primera vista, esta nueva función resulta muy similar a la que dimos en la definición 4.30, pero sin embargo, presenta algunas sutiles diferencias que merecen ser tenidas en cuenta. Al comparar dos *sorts* o variables, diremos que estas satisfacen la condición de ser β-equivalentes si es que en efecto son los mismos *sorts* o variables respectivamente. Cuando comparamos abstracciones o productos dependientes, la manera de proceder es similar a la dada para la función $check_=$, con la salvedad de que las nuevas restricciones generadas reflejan la necesidad de que las expresiones involucradas resulten β-equivalentes y no sólo α-equivalentes, tal y como ocurría con anterioridad.

Las semejanzas con la función $check_=$ se extienden también al caso en que comparamos dos incógnitas sin sustituciones asociadas y cuyo identificador coincide, diciendo que son efectivamente β-equivalentes (ya que en efecto son las mismas incógnitas). No obstantes es hasta aquí que llegan las semejanzas entre estas dos funciones. Anteriormente, cuando comparábamos una incógnita sin sustituciones con una expresión cualquiera, podíamos deducir que dicha incógnita podía ser reemplazada por la expresión con la cual la estábamos comparando. Sin embargo, esto ya no es posible si trabajamos bajo el supuesto de que ambas son β-equivalentes. Esto se debe a que la incógnita en sí no tiene porque ser la expresión con la cual la estábamos analizando, sino que bien podría ser un β-redex que reduce a la expresión en cuestión. Dicho de

otro modo, al toparnos con una restricción de la forma $=^R_\beta (?_1 \,[\,], x)$, no podemos deducir que $?_1 \,[\,] \equiv x$, ya que bien podría resultar que $?_1 \,[\,] \equiv (\lambda y : A \,.\, y)\; x$ y la restricción sigue satisfaciéndose. Por este motivo, cuando nos topamos ante restricciones de esta forma, optamos por mantenerlas para que puedan ser analizadas más tarde, cuando se tenga algo mas de información que nos permita efectuar una elección unívoca del valor que asignaremos a la incógnita.

Para el caso de la aplicación no hacemos más que verificar que cada uno de los componentes que forman parte las aplicaciones sean β-equivalentes entre sí. Por un lado, pedimos que las expresiones sobre las cuales estamos efectuando la aplicación sea una variable, una incógnita o una aplicación. Esto nos restringe casos erróneos como en los que un *sort* o un producto cartesiano se encuentre aplicado a una expresión. Uno podría cuestionarse, sin embargo, porque estamos considerando que las expresiones $(\lambda x : s_1 \,.\, x)\; s_2$ y s_2 no son β-equivalentes cuando en realidad si lo son. Bueno, en efecto, este caso no está contemplado bajo la función chequeadora de $=^R_\beta$, porque los redexes ya son reducidos en el mismo momento en que se invoca a la función chequeadora que estamos analizando, tal y como veremos en la definición 4.34. De modo que el caso planteado sí está siendo considerado, con la salvedad de que la manipulación de las expresiones se lleva a cabo en dos lugares diferentes.

Más adelante necesitaremos aplicar sustituciones de incógnitas a expresiones que forman parte de un conjunto de restricciones. Si bien la aplicación de dichas sustituciones no representa una gran dificultad, puede que resulte conveniente fijar una idea de a qué nos estamos refiriendo.

Definición 4.32 (aplicación de sustituciones a restricciones).
Sea $\delta^?$ una sustitución de identificadores de incógnitas. Definimos la aplicación de tal sustitución a una restricción como:

$$\begin{array}{rcl}
\in^R_S (M)/\delta^? & = & \in^R_S (M/\delta^?) \\
\in^R_A (E_1, E_2)/\delta^? & = & \in^R_A (E_1/\delta^?, E_2/\delta^?) \\
\in^R_R (E_1, E_2, E_3)/\delta^? & = & \in^R_R (E_1/\delta^?, E_2/\delta^?, E_3/\delta^?) \\
\in^R_{V \cup S} (M)/\delta^? & = & \in^R_{V \cup S} (M/\delta^?) \\
=^R (E_1, E_2)/\delta^? & = & =^R (E_1/\delta^?, E_2/\delta^?) \\
=^R_\beta (E_1, E_2)/\delta^? & = & =^R_\beta (E_1/\delta^?, E_2/\delta^?)
\end{array}$$

Desde luego que esta definición también resulta extensible para los conjuntos de restricciones. Es decir, si c es un conjunto de restricciones de la forma:

$$c \quad = \quad \{c_1, \ldots, c_n\}$$

y $\delta^?$ es una sustitución de incógnitas, entonces definimos la aplicación de $\delta^?$ al conjunto c como:

$$c/\delta^? \quad = \quad \{c_1/\delta^?, \ldots, c_n/\delta^?\} \qquad \dagger$$

Ahora que ya hemos dado las definiciones de las funciones auxiliares que necesitamos, estamos en condiciones de definir una relación que nos permita determinar cuándo un conjunto de restricciones es válido o no. Pero antes de ello, definiremos una función que nos simplificará la descripción de tal relación.

Definición 4.33 (*mergeChk*).
Sean $\vec{\delta}$ y $\vec{\delta}'$ listas de sustituciones de identificadores de incógnitas. Definimos la función de aplicación de chequeo de restricciones:

$$mergeChk : ([Subst^?], \mathscr{P}(Constr)) \times ([Subst^?].\mathscr{P}(Constr))_{\bowtie} \to ([Subst^?].\mathscr{P}(Constr))_{\bowtie}$$

dada por:

$$\begin{array}{lcl} mergeChk\ ((\vec{\delta}, c),\ (\vec{\delta}', \tilde{c})) & = & (\vec{\delta}\ \#\ \vec{\delta}', (c/\vec{\delta}') \cup \tilde{c}) \\ mergeChk\ ((\vec{\delta}, c),\ \bowtie) & = & \bowtie \end{array}$$

donde # representa la concatenación de listas. †

Nuestra función *mergeChk* simplemente combina el resultado de dos análisis de chequeo de restricciones. El primer argumento que recibe corresponde a los valores de los cuales se parte y el segundo argumento identifica la nueva información que se desea agregar a la ya existente. Si este segundo parámetro representa información válida que puede ser anexada a la que ya tenemos, entonces se combinan ambos resultados. Si por el contrario, el segundo parámetro identifica la no satisfacción de una restricción, se retorna tal valor, denotando que la unificación de resultados conduce a la no satisfacción de algunas restricciones.

Definición 4.34 (chequeo de restricciones $\xrightarrow{\text{check}}$).
Sean $\delta_1^?, \ldots, \delta_n^?$ sustituciones de identificadores. Sea $\vec{\delta} = [\delta_1^?, \ldots, \delta_n^?]$. Dada una especificación *PTS*, definimos la relación de chequeo de restricciones bajo los supuestos de dicha especificación como:

$$\xrightarrow{\text{check}} : ([Subst^?], \mathscr{P}(Constr))_{\bowtie} \times ([Subst^?], \mathscr{P}(Constr))_{\bowtie}$$

la cual queda definida por:

$$\begin{array}{rcl} (\vec{\delta},\ \{c_1, \ldots, c_m, \in_S^R (M)\}) & \xrightarrow{\text{check}} & mergeChk\ ((\vec{\delta},\ \{c_1, \ldots, c_m\}),\ check_{\in_S}(M, PTS)) \\ (\vec{\delta},\ \{c_1, \ldots, c_m, \in_A^R (E_1, E_2, \})) & \xrightarrow{\text{check}} & mergeChk\ ((\vec{\delta},\ \{c_1, \ldots, c_m\}),\ check_{\in_A}(E_1, E_2, PTS)) \\ (\vec{\delta},\ \{c_1, \ldots, c_m, \in_R^R (E_1, E_2, E_3, \},)) & \xrightarrow{\text{check}} & mergeChk\ ((\vec{\delta},\ \{c_1, \ldots, c_m\}),\ check_{\in_R}(E_1, E_2, E_3, PTS)) \\ (\vec{\delta},\ \{c_1, \ldots, c_m, \in_{V \cup S}^R (M)\}) & \xrightarrow{\text{check}} & mergeChk\ ((\vec{\delta},\ \{c_1, \ldots, c_m\}),\ check_{\in_{V \cup S}}(M, PTS)) \\ (\vec{\delta},\ \{c_1, \ldots, c_m, =^R (E_1, E_2)\}) & \xrightarrow{\text{check}} & mergeChk\ ((\vec{\delta},\ \{c_1, \ldots, c_m\}),\ check_{=}(E_1, E_2, PTS)) \\ (\vec{\delta},\ \{c_1, \ldots, c_m, =_\beta^R (E_1, E_2)\}) & \xrightarrow{\text{check}} & mergeChk\ ((\vec{\delta},\ \{c_1, \ldots, c_m\}),\ check_{=_\beta}(E_1', E_2', PTS)) \\ (\vec{\delta},\ \varnothing) & \xrightarrow{\text{check}} & (\vec{\delta},\ \varnothing) \\ \bowtie & \xrightarrow{\text{check}} & \bowtie \end{array}$$

donde:

$$\begin{array}{l} E_1 \dashrightarrow E_1' \\ E_2 \dashrightarrow E_2' \end{array}$$ †

Tal vez resulte conveniente hacer algunas aclaraciones respecto a esta nueva relación. Básicamente lo que define es una relación entre los elementos que conforman el conjunto de posibles soluciones de un conjunto de restricciones. Así, si en un paso del proceso de verificación de restricciones el conjunto de condiciones que aún quedan por

ser verificadas es no vacío, podemos tomar un elemento de dicho conjunto y aplicarle la función chequeadora que corresponda. Si se determina que dicha restricción no se satisface, entonces podemos concluir que el conjunto de restricciones no es satisfacible. Si por el contrario al analizar la restricción obtenemos nuevos datos, añadimos los mismos a los que ya teníamos y así obtenemos un nuevo par que puede seguir siendo estudiado. Si en algún momento el conjunto de restricciones resulta vacío, entonces ya habremos concluido con el análisis de los mismos. Por otro lado, si partimos de una solución no válida ($\bowtie$), el único valor que podemos obtener es el mismo.

Esta nueva relación nos permite determinar cuándo un conjunto de restricciones resulta verdadero, falso o al menos parcialmente reducible para un *PTS* dado. Es decir, dado un conjunto de restricciones $\{c_1, \ldots, c_n\}$, diremos que bajo una especificación *PTS* dicho conjunto de restricciones es:

Verdadero: Si se satisface que:

$$([], \{c_1, \ldots, c_n\}) \quad \xrightarrow{\text{check}}{}^{*} \quad ([\delta_1^?, \ldots \delta_k^?], \varnothing)$$

resultando ser $[\delta_1^?, \ldots \delta_k^?]$ la lista de sustituciones de identificadores de incógnitas que hemos podido deducir en el proceso de verificación de restricciones y por lo tanto, deberían ser aplicados a los juicios y expresiones involucradas en la formulación original de las restricciones. Es decir que resulta posible verificar todas las condiciones del conjunto recibido si es que las incógnitas listadas en la lista de sustituciones son reemplazadas por las expresiones indicadas. En tal caso, si $\vec{\delta} = [\delta_1^?, \ldots \delta_k^?]$, decimos que $\vec{\delta}$ es la solución hayada.

Falso: Si se satisface que:

$$([], \{c_1, \ldots, c_n\}) \quad \xrightarrow{\text{check}}{}^{*} \quad \bowtie$$

lo que claramente nos está diciendo que al intentar analizar algunas de las restricciones, ésta se pudo determinar como falsa y por lo tanto, sin importar lo que resultase de analizar las demás restricciones hemos podido determinar que el conjunto de restricciones en sí mismo no puede ser satisfecho. Dicho de otro modo, el conjunto de restricciones no tiene solución, de modo que no existe un $\vec{\delta}$ que satisfaga las restricciones $c_1, \ldots, c_n$.

Reducible: Si no podemos determinar la veracidad o falsedad de las condiciones incluidas en el conjunto de restricciones, podemos al menos intentar obtener un nuevo conjunto que contenga condiciones más simples de verificar. Diremos que esto ocurre cuando:

$$([], \{c_1, \ldots, c_n\}) \quad \xrightarrow{\text{check}}{}^{*} \quad ([\delta_1^?, \ldots \delta_k^?], \{\tilde{c_1}, \ldots, \tilde{c_m}\})$$

y además $\forall sol$ tal que:

$$([\delta_1^?, \ldots \delta_k^?], \{\tilde{c_1}, \ldots, \tilde{c_m}\}) \quad \xrightarrow{\text{check}}{}^{i} \quad sol$$

se satisface que:

$$sol \quad = \quad ([\delta_1^?, \ldots \delta_k^?], \{\tilde{c_1}, \ldots, \tilde{c_m}\})$$

cuando se dé esta situación, diremos que el conjunto de restricciones original pudo ser reducido a uno más simple (que claramente podría ser incluso el mismo

desde el cual partimos), y durante el proceso de reducción se lograron deducir las asignaciones a incógnitas dadas por la lista $[\delta_1^?, \ldots \delta_k^?]$. Dicho de otro modo, hemos podido construir una solución parcial, la cual aún no somos capaces no poder determinar si tal solución existe o no.

Para establecer la diferencia entre los distintos resultados posibles que se pueden obtener al emplear la relación $\xrightarrow{\text{check}}$, establecemos una nueva relación que nos determina el punto hasta el cuál resulta factible operar sobre una posible solución de un conjunto de restricciones.

Definición 4.35 ($\overset{\text{check}}{\Longrightarrow}$).
Para definir la relación $\overset{\text{check}}{\Longrightarrow}$ sobre elementos de $([Subst^?], \mathscr{P}(Constr))$, diremos que:

$$([\delta_1, \ldots, \delta_n], \{c_1, \ldots, c_m\}) \overset{\text{check}}{\Longrightarrow} ([\hat{\delta_1}, \ldots, \hat{\delta_k}], \{\hat{c_1}, \ldots, \hat{c_l}\})$$

si y sólo si:

$$([\delta_1, \ldots, \delta_n], \{c_1, \ldots, c_m\}) \xrightarrow{\text{check}}{}^{*} ([\hat{\delta_1}, \ldots, \hat{\delta_k}], \{\hat{c_1}, \ldots, \hat{c_l}\})$$

y además $\nexists\, \hat{\delta_1}', \ldots, \hat{\delta_k}', \hat{c_1}', \ldots, \hat{c_l}'$ tales que:

$$\hat{\delta_i} \neq \hat{\delta_i}' \qquad \text{para algún } i \in \{1, \ldots, k\}$$

$$\hat{c_i} \neq \hat{c_i}' \qquad \text{para algún } i \in \{1, \ldots, l\}$$

que satisfagan que:

$$([\hat{\delta_1}, \ldots, \hat{\delta_k}], \{\hat{c_1}, \ldots, \hat{c_l}\}) \xrightarrow{\text{check}} ([\hat{\delta_1}', \ldots, \hat{\delta_k}'], \{\hat{c_1'}, \ldots, \hat{c_l'}\}) \qquad \dagger$$

Veamos algunos ejemplos que muestren el comportamiento de la relación $\xrightarrow{\text{check}}$ en el proceso de verificación de restricciones.

Ejemplo 4.6 (uso de la relación $\xrightarrow{\text{check}}$):
Como dijimos, un conjunto de restricciones puede ser considerado verdadero, falso o reducible. A continuación damos algunos ejemplos que ilustran cada una de estas situaciones posibles.

Restricciones verdaderas Sea el conjunto original de restricciones que queremos verificar el dado por:

$$\{=^R ((\lambda x :?_1 [\,] \,.\, x)\; z, (\lambda x : A \,.\, x)\; ?_2 [\,])\}$$

luego empleando la relación $\xrightarrow{\text{check}}$ obtenemos que:

$$\begin{array}{ll}
([], \{=^R ((\lambda x :?_1 [\,] \,.\, x)\; z, (\lambda x : A \,.\, x)\; ?_2 [\,])\}) & \xrightarrow{\text{check}} \\
([], \{=^R (z, ?_2 [\,]), =^R (A, ?_1 [\,])\}) & \xrightarrow{\text{check}} \\
([[z/2]], \{=^R (A, ?_1 [\,])\}) & \xrightarrow{\text{check}} \\
([[z/2], [A/1]], \varnothing) &
\end{array}$$

lo cual nos dice que las restricciones se satisfacen y que además se ha deducido los valores que deberían tomar las incógnitas con identificadores 1 y 2.

Otro ejemplo se puede dar bajo la especificación del sistema $\lambda\!\rightarrow$. Supongamos tener la restricción $\in^R_R (?_1[],?_2[],?_3[])$, luego empleando la relación $\xrightarrow{\text{check}}$ podemos observar que:

$$\begin{array}{ll}
([], \{\in^R_R (?_1[],?_2[],?_3[])\}) & \xrightarrow{\text{check}} \\
([], \{=^R (?_1[],\star), =^R (?_2[],\star), =^R (?_3[\star],\})) & \xrightarrow{\text{check}} \\
([[\star/1]], \{=^R (?_2[],\star), =^R (?_3[\star],\})) & \xrightarrow{\text{check}} \\
([[\star/1],[\star/2]], \{=^R (?_3[\star],\})) & \xrightarrow{\text{check}} \\
([[\star/1],[\star/2],[\star/3]], \varnothing) &
\end{array}$$

Lo cual nos esta diciendo que, debido a que la única regla definida en la especificación del sistema $\lambda\!\rightarrow$ es la de $(\star,\star,\star)$, a partir de la restricción original podemos determinar que la misma puede ser satisfecha y que además durante el proceso de verificación fuimos capaces de determinar los valores de las tres incógnitas involucradas.

Restricciones falsas Supongamos tener el siguiente conjunto de restricciones:

$$\{=^R (\lambda x : \Box\,.\,x, \lambda y : ?_2[]\,.\,y), \in^R_R (?_1[],?_2[],?_3[])\}$$

bajo la especificación del sistema $\lambda 2$, empleando la relación $\xrightarrow{\text{check}}$ nos encontramos con que:

$$\begin{array}{ll}
([], \{=^R (\lambda x : \Box\,.\,x, \lambda y : ?_2[]\,.\,y), \in^R_R (?_1[],?_2[],?_3[])\}) & \xrightarrow{\text{check}} \\
([], \{=^R (\Box, ?_2[]), =^R (x[y/x], y), \in^R_R (?_1[],?_2[],?_3[])\}) & \xrightarrow{\text{check}} \\
([[\Box/2]], \{=^R (y,y), \in^R_R (?_1[],\Box,?_3[])\}) & \xrightarrow{\text{check}} \\
([[\Box/2]], \{\in^R_R (?_1[],\Box,?_3[])\}) & \xrightarrow{\text{check}} \\
\bowtie &
\end{array}$$

Lo cual significa que el conjunto de restricciones no puede ser satisfecho. Esto ocurre debido a que cuando asignamos la expresión $\Box$ a la incógnita con identificador 2, obtenemos una posible regla que no se condice con ninguna de las que existen en el sistema de tipos bajo el cual estamos trabajando y por lo tanto el conjunto de condiciones resulta falso.

Restricciones reducibles Supongamos estar trabajando con la especificación que corresponde al sistema λC. Si quisiéramos verificar el conjunto de restricciones dado por:

$$\{\in^R_R (?_1[],?_2[],?_3[]), =^R (?_2[],\Box), =^R (?_3[],\Box)\}$$

luego, empleando la relación $\xrightarrow{\text{check}}$ podemos determinar que:

$$\begin{array}{ll}
([], \{\in^R_R (?_1[],?_2[],?_3[]), =^R (?_2[],\Box), =^R (?_3[],\Box)\}) & \xrightarrow{\text{check}} \\
([[\Box/2]], \{\in^R_R (?_1[],\Box,?_3[]), =^R (?_3[],\Box)\}) & \xrightarrow{\text{check}} \\
([[\Box/2],[\Box/3]], \{\in^R_R (?_1[],\Box,\Box)\}) &
\end{array}$$

Lo que ocurre aquí es que una vez que hemos efectuado las sustituciones de las incógnitas 2 y 3, dado el sistema de tipos que estamos empleando, aun queda más de una solución posible a nuestro problema. Es decir, la incógnita con identificador 1 podría ser asignada tanto al valor $\star$ como al valor $\Box$ siendo en ambos casos una regla verdadera para el sistema λC. Debido a ello, es que no podemos continuar con el análisis de la restricción resultante hasta tanto tengamos mas información que nos permita verificarla o descomponerla en restricciones más sencillas. $*$

Con esto nos basta por el momento para tener un entendimiento de cuáles son las restricciones que manipula nuestra herramienta. Más adelante, en la sección 4.4.1 veremos cómo podemos generar restricciones para cuando así lo requiramos. Pero antes de poder analizar los mecanismos de generación deberemos comprender la noción de derivación y los mecanismos que existen para manipular este tipo de estructura.

4.3.3 Derivaciones

Ahora que ya hemos fijado una idea de lo que consideraremos un juicio o una restricción, junto a algunas de las operaciones más importantes que necesitaremos para operar sobre los mismos, estamos en condiciones de determinar la manera en que se relacionarán todos ellos para conformar una derivación. Comenzaremos por definir lo que llamaremos un árbol de derivación.

Definición 4.36 (paso de derivación).
Llamaremos un *paso de derivación* a un par de la forma (J,R), donde:

- J es un juicio.
- R es una constante perteneciente al conjunto de constantes *Rules*, dado por:
  $\{Axiom, Start, Weakening, Product, Abstraction, Application, Conversion, NoRule\}$

A menudo emplearemos la letra r para referirnos a elementos de *Rules*. †

Aunque no lo hemos dicho explicitamente, las constantes que conforman el conjunto dado en esta definición, se corresponden con las reglas de tipado que consideraremos válidas en nuestra aplicación. Más aún cada una de las constantes se corresponde con la regla de tipado homónima introducida en la figura 2.31 al definir los sistemas de tipos puros, excepto la constante *NoRule* que indicará la ausencia de regla alguna.

Definición 4.37 (árbol de derivación).
Definimos un *árbol de derivación* como un árbol finitario en donde cada nodo contiene un *paso de derivación*. Dado el nodo que contiene el paso (J,R), si éste contiene n hijos, diremos que el juicio J puede ser obtenido aplicando la regla R tomando como premisas los juicios contenidos en los nodos hijos de dicho nodo. †

Con la definición anterior, lo único que estamos tratando de dejar en claro es una idea de lo que formará parte de nuestras derivaciones. Claramente nuestra definición se

ajusta bastante a la noción de derivación que usualmente se tiene al trabajar en teoría de tipos del cálculo lambda. De hecho, de acuerdo a nuestra definición, si tomamos por ejemplo la prueba de la identidad monomórfica para el sistema $\lambda\!\rightarrow$ especificado en forma de *PTS*, veremos que ésta toma la forma de:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[] \vdash \star : \Box}\,(\mathrm{Ax})}{[k:\star] \vdash k : \star}\,(\mathrm{St})}{[k:\star,\, x_1:k] \vdash x_1 : k}\,(\mathrm{St}) \qquad \dfrac{\dfrac{\dfrac{}{[] \vdash \star : \Box}\,(\mathrm{Ax})}{[k:\star] \vdash k : \star}\,(\mathrm{St}) \qquad \dfrac{\dfrac{\dfrac{}{[] \vdash \star : \Box}\,(\mathrm{Ax})}{[k:\star] \vdash k : \star}\,(\mathrm{St}) \qquad \dfrac{\dfrac{}{[] \vdash \star : \Box}\,(\mathrm{Ax})}{[k:\star] \vdash k : \star}\,(\mathrm{St})}{[k:\star,\, x:k] \vdash k : \star}\,(\mathrm{Wk})}{[k:\star] \vdash \Pi x : k\,.\,k : \star}\,(\mathrm{Pr})}{[k:\star] \vdash \lambda x_1 : k\,.\,x_1 \;:\; \Pi x : k\,.\,k}\,(\mathrm{Ab})$$

Correspondiéndose ésta, de acuerdo a nuestra definición de *árbol de derivación* con la siguiente estructura:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{([] \vdash \star : \Box, Ax)}}{([k:\star] \vdash k : \star, St)}}{([k:\star,\, x_1:k] \vdash x_1 : k, St)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{([] \vdash \star : \Box, Ax)}}{([k:\star] \vdash k : \star, St)}}{([k:\star,\, x:k] \vdash k : \star, Wk)} \qquad \dfrac{\dfrac{}{([] \vdash \star : \Box, Ax)}}{([k:\star] \vdash k : \star, St)} \qquad \dfrac{\dfrac{}{([] \vdash \star : \Box, Ax)}}{([k:\star] \vdash k : \star, St)}}{([k:\star] \vdash \Pi x : k\,.\,k : \star, Pr)}$$

$$([k:\star] \vdash \lambda x_1 : k\,.\,x_1 \Pi x : k\,.\,k : , Ab)$$

De todas maneras, por comodidad y simpleza, emplearemos la primer notación para denotar un árbol de derivación. De todas maneras, más adelante estaremos interesados en manipular una expresión que represente un árbol como el dado anteriormente. Para ello introducimos la siguiente notación.

Notación 4.9 (árboles):
Para representar un árbol finitario a veces emplearemos un par, donde el primer componente representa el valor almacenado por un nodo del árbol y el segundo componente es una lista de los árboles hijos del nodo que hemos definido. Así, por ejemplo, el árbol:

$$\dfrac{B_1 \quad \ldots \quad B_n}{A}$$

estará representado a través del par:

$$(A,\ [B_1, \ldots, B_n]) \qquad \diamondsuit$$

Desde luego que nuestra notación para representar árboles no nos permite describir un árbol vacío. Pero como de todas formas no necesitaremos describir un árbol vacío, nuestra notación no presentará mayores inconvenientes para los desarrollos que efectuaremos más adelante.

El lector acostumbrado a trabajar con expresiones correspondientes al cálculo lambda notará que nuestra definición de *árbol de derivación* se corresponde con lo que

comúnmente denominamos derivaciones. Sin embargo, dada la naturaleza de nuestro problema, un árbol de derivaciones no nos provee de toda la información requerida. Por lo tanto, damos nuestra propia definición de lo que consideraremos una derivación.

Definición 4.38 (derivación).
Definimos una derivación como un par de la forma (DT, CS), donde:

- DT es un *árbol de derivación* de acuerdo a nuestra definición 4.37.
- CS es un conjunto de restricciones tal y como las definimos en la sección anterior. †

Lo que estamos representando a través de nuestra definición de *derivación* es que, a los fines prácticos del desarrollo de nuestra herramienta, la información que nos provee un árbol no nos es suficiente. Esto ocurre debido a que en numerosas ocasiones existirán algunas condiciones que deben ser satisfechas para que el árbol en sí mismo se encuentre bien formado. De esta manera, el árbol nos provee de la estructura misma que tiene nuestra derivación, mientras que el conjunto de restricciones nos indicará las condiciones que aún nos restan por comprobar para poder completar la derivación.

Debido a que necesitaremos muy a menudo referirnos a elementos de una derivación, extenderemos el significado de algunos de los operadores comúnmente empleados en la teoría de conjuntos para referirnos a relaciones aplicables a nuestra definición de derivación.

Definición 4.39 (pertenencia de juicios a derivaciones).
Dado un árbol de derivación dt y un paso de derivación (j,r), diremos que el paso de derivación (j,r) pertenece al árbol de derivación dt, en símbolos $(j,r) \in dt$ si y sólo si (j,r) es un nodo de dt.
De manera similar, dado un árbol de derivación dt, diremos que un juicio $j \in J_A$ pertenece a dicho árbol, en símbolos $j \in dt$ si y sólo si $\exists r \in Rules : (j,r) \in dt$.
Extendiendo la definición, diremos que un juicio j pertenece a una derivación (dt, cs) cuando se satisfaga que $j \in dt$. †

Una de las nociones primordiales que manejaremos en nuestra herramienta es la de derivaciones completas. Diremos que una derivación es completa cuando ya no posee incógnitas y todas sus restricciones han sido verificadas.

Definición 4.40 (derivación completa).
Dada una derivación (dt, cs), diremos que la misma está completa si y sólo si se satisfacen las siguientes condiciones:

- $\forall j \in dt$, se satisface que j se encuentra completamente definido.
- $\forall (j,r) \in dt$, tal que j no tiene premisas (i.e. (j,r) es una hoja en el árbol de derivaciones dt), se satisface que $r = Axiom$.
- $cs = \varnothing$ †

Sobre las derivaciones en sí, no hay demasiado que precisar, ya que en realidad se comportan como meros estados de una prueba. Lo que sí nos resulta interesante son las funciones con que contamos para manipular dichas derivaciones. Más precisamente aquéllas que nos permitan construir y eliminar trozos de derivaciones. A las primeras las llamamos funciones generadoras de premisas y serán estudiadas en la sección 4.4.1 en tanto que el proceso de eliminación de juicios pertenecientes a una derivación se analizará en la sección 4.4.2.

4.4 Manipulación de derivaciones

En las secciones anteriores nos abocamos a la tarea de definir lo que consideramos una derivación y cada uno de los componentes que forman parte de la misma. Así, primero introdujimos el concepto de expresión y vimos cómo combinando expresiones lográbamos construir juicios. Luego planteamos la idea de emplear construcciones para denotar restricciones que deberían satisfacer nuestras derivaciones y junto a ella mecanismos que permiten determinar la validez o no de las mismas. Finalmente esbozamos una definición de lo que consideraremos una derivación en nuestra aplicación. En esta sección lo que haremos será entonces dar algunos mecanismos que nos permitan construir y manipular tales derivaciones. Vale la pena aclarar que las modificaciones introducidas a las reglas de tipado para *PTS* empleadas para la generación de premisas en este capítulo se corresponden a las introducidas por Barthe [Bar99].

4.4.1 Generación de premisas y restricciones

Las definiciones que daremos a continuación nos permitirán desarrollar derivaciones a partir de la generación de premisas para los juicios que forman parte del árbol de derivación en el cual estamos interesados.

Lo primero que haremos será determinar el comportamiento que deberían tener nuestras funciones generadoras de premisas. Éstas deberán recibir un juicio como argumento y retornar un conjunto de posibles premisas que conduzcan a la deducción de dicho juicio. Más aún, cada elemento del conjunto de posibles soluciones no sólo debe contener las premisas sino que además indicar cuál es la regla que permite efectuar tal deducción junto al conjunto de condiciones que se deben verificar en caso de que realmente optásemos por emplear tales premisas en nuestra derivación.

Definición 4.41 (generación de premisas para la regla de *Axiom*).
Definimos la función generadora de premisas para la regla de *Axiom* como:

$$g_{Ax} : J_A \to \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr)\big)\Big)$$

la cual esta dada por:

$$g_{Ax}(\Gamma \vdash M : N) \quad = \quad \begin{cases} \Big\{\big(Axiom,\ [],\ \{\in_S^R (M), \in_S^R (N), \in_A^R (M,N)\}\big)\Big\} & \text{si } \Gamma = [] \\ \varnothing & \text{c.c.} \end{cases}$$

†

En esta primera definición, estamos dando la manera en que se pueden construir premisas para un juicio a través de la regla de *Axiom*. En esencia, lo que estamos diciendo es que, si recibimos un juicio que contiene un contexto vacío, entonces podemos decir que el mismo se obtiene a partir de la regla de *Axiom* si es que somos capaces de verificar que las expresiones M y N son axiomas dentro de la especificación de *PTS* que estemos utilizando. Además, como dicha regla no contiene premisas, la lista de premisas resulta vacía.

Uno podría cuestionarse en este momento si para retornar las premisas de un juicio, realmente necesitamos de una lista, o si en su lugar podríamos decir que las mismas forman un conjunto. Si bien el orden de las premisas no resulta importante para el resultado de una derivación, en nuestro caso particular, necesitaremos que las premisas generadas se encuentren siempre en una misma ubicación, lo que nos permitirá hacer algunas deducciones más adelante. De allí que requerimos el uso de una lista y no un conjunto.

Definición 4.42 (generación de premisas para la regla de *Start*).
Definimos la función generadora de premisas para la regla de *Start* como:

$$g_{St} \,:\, J_A \,\to\, \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr)\big)\Big)$$

la cual está dada por:

$$g_{St}(\Gamma \vdash M \,:\, N) \;=\; \begin{cases} \Big\{\big(\, Start, \\ \quad [\Delta \vdash N \,:\, ?_k[]], & \text{si } \Gamma = \Delta, (x:A) \,\wedge\, x \notin dom(\Delta) \\ \quad \{\in^R_S (?_k[]), =^R (x,M), =^R (A,N)\}\big)\Big\} & \\ \varnothing & \text{c.c.} \end{cases}$$

donde $?_k[]$ es una incógnita nueva. †

Nótese que cuando pedimos que un identificador de incógnita sea nuevo, no sólo estamos pidiendo que no esté siendo empleado en ninguno de los juicios que forma parte del árbol de derivación, sino que además, no se use en ninguna de las restricciones que forma parte de dicha derivación. Necesitamos pedir esta condición para asegurarnos de que la incógnita que estamos introduciendo no entre en conflicto con alguna otra incógnita que se encuentre presente en la derivación que estemos efectuando.

Refiriéndonos estrictamente al comportamiento que presenta la función g_{St}, vemos que la misma refleja el mecanismo de generación de premisas a partir de la regla de *Start*. Es decir, dado un juicio $\Gamma \vdash M \,:\, N$, indica que el mismo puede ser obtenido aplicando la regla de *Start* si es que podemos verificar que la expresión N se corresponde con el tipo del último elemento que forma parte del contexto Γ y además si podemos demostrar que N se encuentra tipado por un *sort*.

Definición 4.43 (generación de premisas para la regla de *Weakening*).
Definimos la función generadora de premisas para la regla de *Weakening* como:

$$g_{Wk} \,:\, J_A \,\to\, \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr)\big)\Big)$$

la cual esta dada por:

$$g_{Wk}(\Gamma \vdash M : N) = \begin{cases} \Big\{ \big(\mathit{Weakening}, \\ \quad [\Delta \vdash M : N, \Delta \vdash C : ?_k[]], \\ \quad \{\in^R_{V \cup S}(M), \in^R_S (?_k[])\}\big)\Big\} & \text{si } \Gamma = \Delta, (x:C) \wedge x \notin dom(\Delta) \\ \varnothing & \text{c.c.} \end{cases}$$

donde $?_k[]$ es una incógnita nueva. †

Al igual que para el caso de la regla *Start*, necesitamos que k sea un identificador de incógnita que no haya sido empleado en ninguna parte de la derivación. Dejando de lado esta aclaración, nuestra función g_{Wk} refleja el mecanismo que conduce a la generación de premisas de un juicio a través de la aplicación de la regla *Weakening*. Para ello, dado un juicio $\Gamma \vdash M : N$ establece que se puede derivar tal juicio si logramos demostrar que M tiene tipo N en un contexto reducido (Γ sin su último elemento) y que el tipo de éste ultimo elemento de Γ es un *sort*. Claro que además, deberemos verificar que M es o bien una variable o bien un *sort*.

Definición 4.44 (generación de premisas para la regla de *Application*).
Definimos la función generadora de premisas para la regla de *Application* como:

$$g_{Ap} : J_A \to \mathscr{P}\Big(\big(\mathit{Rules}, [J_A], \mathscr{P}(\mathit{Constr})\big)\Big)$$

la cual esta dada por:

$$\begin{aligned} g_{Ap}(\Gamma \vdash M : N) = \Big\{ & \big(\mathit{Application}, \\ & [\Gamma \vdash ?_{k_1}[] : (\Pi x :?_{k_2}[] . ?_{k_3}[]), \Gamma \vdash ?_{k_4}[] : ?_{k_2}[]], \\ & \{=^R (M, ?_{k_1}[]\ ?_{k_4}[]), =^R (N, ?_{k_3}[]/[?_{k_4}[]/x])\}\big)\Big\} \end{aligned}$$

donde $?_{k_1}[]$, $?_{k_2}[]$, $?_{k_3}[]$, $?_{k_4}[]$ son incógnitas nuevas. †

Más allá de que luce un poco engorrosa la definición de g_{Ap}, si uno presta atención notará que no hace más que reflejar el paso de derivación que nos permite derivar el juicio $\Gamma \vdash M : N$ a través de la regla de *Application*. En nuestra definición establecemos el hecho de que para poder aplicar dicha regla deberíamos ser capaces de poder demostrar que contamos con una función desconocida $?_{k_1}[]$ de tipo $\Pi x :?_{k_2}[] . ?_{k_3}[]$ y además una expresión $?_{k_4}[]$ cuyo tipo se corresponda con el del parámetro que recibe $?_{k_1}[]$.

Si M refleja el resultado de aplicar $?_{k_1}[]$ a $?_{k_4}[]$, claramente debemos verificar que dicha expresión representa la aplicación de tales expresiones ($M = ?_{k_1}[]\ ?_{k_4}[]$, en nuestro caso). Además deberíamos ser capaces de chequear que la expresión N (el tipo de la expresión M) se corresponde con el tipo de los valores que retorna la función que estamos aplicando, donde las ocurrencias del parámetro que recibe dicha función son

reemplazadas por el valor al cual ha sido aplicada la misma ($N = ?_{k_3}[]/[?_{k_4}[]/x]$, para nuestro caso particular).

Definición 4.45 (generación de premisas para la regla de *Product*).
Definimos la función generadora de premisas para la regla de *Product* como:

$$g_{Pr} : J_A \to \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr)\big)\Big)$$

la cual esta dada por:

$$g_{Pr}(\Gamma \vdash M : N) = \begin{cases} \Big\{\big(\, Product, \\ \quad [\Gamma \vdash A : ?_{k_1}[], \Gamma, x:A \vdash B : ?_{k_2}[]], \\ \quad \{\in^R_S (N), \in^R_S (?_{k_1}[]), \in^R_S (?_{k_2}[]), \\ \qquad \in^R_R (?_{k_1}[], ?_{k_2}[], N)\}\big)\Big\} & \text{si } M = \Pi x:A\,.\,B \\ \Big\{\big(\, Product, \\ \quad [\Gamma \vdash ?_{k_1}[] : ?_{k_3}[], \Gamma, y:A \vdash ?_{k_2}[] : ?_{k_4}[]], \\ \quad \{=^R (M, \Pi y:A\,.\,?_{k_2}[]), =^R (?_{k_1}[], A), \\ \qquad \in^R_S (N), \in^R_S (?_{k_3}[]), \in^R_S (?_{k_4}[]) \\ \qquad \in^R_R (?_{k_3}[], ?_{k_4}[], N)\}\big)\Big\} & \text{c.c.} \end{cases}$$

donde pedimos que $?_{k_1}[]$, $?_{k_2}[]$, $?_{k_3}[]$, $?_{k_4}[]$ sean incógnitas nuevas y además que y sea una variable nueva. †

La función g_{Pr} es la encargada de generar el conjunto de premisas a partir de las cuales resulta posible derivar el juicio que recibe como argumento. Es decir, dado un juicio $\Gamma \vdash M : N$ distinguimos dos casos bastantes que comparte la misma idea, pero a través de resultados un poco diferente. Debido a que la manipulación de expresiones desconocidas se puede tornar un poco difícil de manipular cuando tiene asociadas una lista de sustituciones, es que consideramos dos escenarios distintos.

- Cuando la expresión M ya denota un producto dependiente, entonces nos podemos ahorrar la tarea de definir incógnitas que formen un nuevo producto dependiente y pedir posteriormente que esta nueva expresión sea igual a M a través de una restricción en nuestra derivación. Si hiciéramos esto, debido a que las construcciones que estaríamos comparando denotan un producto, al verificar su igualdad de acuerdo a lo expuesto en la definición 4.30, necesitaríamos efectuar un reemplazo de identificadores en alguna de las dos expresiones, lo cual podría eventualmente introducir sustituciones a una *incógnita*. Expresiones desconocidas con listas de sustituciones anidadas podrían llegar a tornarse difíciles de manejar al momento de verificar algunas propiedades de éstas, como por ejemplo la igualdad de éste frente a otras expresiones. Diferenciando el caso en que M ya es un producto dependiente, nos estamos librando (al menos en este paso) del inconveniente que representa trabajar con ese tipo de expresiones desconocidas.

- Cuando M no denota un producto dependiente, entonces no existe ningún motivo que nos impida crear nuevas incógnitas que conformen un producto dependiente y añadir una nueva restricción a nuestra derivación que solicite la igualdad entre dicho producto y nuestra expresión M. Esto se debe a que, si M no es una expresión desconocida, muy probablemente la comparación falle en el mismo momento en que la efectuemos. Por otro lado, si M es una incógnita sin sustituciones asociadas, al momento de efectuar la comparación terminaremos por asignar el valor del nuevo producto dependiente a la incógnita que denota M. Finalmente si M es una incógnita que contiene sustituciones, entonces no tenemos manera alguna de evitarnos el problema de trabajar con este tipo de expresiones, ya que las mismas provenían de un análisis previo.

Como se puede apreciar, la idea es tratar de introducir la menor cantidad de *incógnitas* que podamos. Pero si en algún punto surgieron expresiones de esta forma, entonces no nos quedará mas remedio que lidiar con las mismas.

Dejando de lado esta distinción de casos, el resto de la función se comporta de manera idéntica en ambas situaciones. Es decir, genera las premisas que piden demostrar que el argumento que recibe el producto dependiente que estamos derivando sea un *sort* y que además asumiendo la existencia de dicho valor resulte posible probar que el valor de retorno del producto también se encuentra correctamente tipado, siendo su tipo igualmente un *sort*. Finalmente la última condición que nos resta por pedir es que el tipo del producto dependiente también se corresponda a un *sort* y que la relación entre éste y el valor recibido y devuelto se encuentre avalada por alguna regla de la especificación del *PTS* que estemos utilizando.

Definición 4.46 (generación de premisas para la regla de *Abstraction*).
Definimos la función generadora de premisas para la regla de *Abstraction* como:

$$g_{Ab} : J_A \to \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr)\big)\Big)$$

la cual esta dada por:

$$
g_{Ab}(\Gamma \vdash M : N) \quad = \quad \begin{cases}
\Big\{\big(\mathit{Abstraction},\ [\Gamma, x:A \vdash B : B', \Gamma \vdash N : ?_k\,[]],\ \{\in^R_S (?_k\,[])\}\big)\Big\} & \text{si} \quad M = \lambda x:A\,.\,B \wedge N = \Pi y:A'\,.\,B' \\
\Big\{\big(\mathit{Abstraction},\ [\Gamma, x:A \vdash B : ?_{k_2}\,[],\ \Gamma \vdash (\Pi x:?_{k_1}\,[]\,.\,?_{k_2}\,[]) : ?_{k_3}\,[]],\ \{=^R (N, \Pi x:?_{k_1}\,[]\,.\,?_{k_2}\,[]),\ \in^R_S (?_{k_3}\,[])\}\big)\Big\} & \text{si} \quad M = \lambda x:A\,.\,B \wedge N \neq \Pi y:A'\,.\,B' \\
\Big\{\big(\mathit{Abstraction},\ [\Gamma, y:A' \vdash ?_{k_1}\,[] : B',\ \Gamma \vdash N : ?_{k_2}\,[]],\ \{=^R (M, \lambda y:A'\,.\,?_{k_1}\,[]),\ \in^R_S (?_{k_2}\,[])\}\big)\Big\} & \text{si} \quad M \neq \lambda x:A\,.\,B \wedge N = \Pi y:A'\,.\,B' \\
\Big\{\big(\mathit{Abstraction},\ [\Gamma, z:?_{k_1}\,[] \vdash ?_{k_3}\,[] : ?_{k_2}\,[],\ \Gamma \vdash (\Pi z:?_{k_1}\,[]\,.\,?_{k_2}\,[]) : ?_{k_4}\,[]],\ \{=^R (M, \lambda z:?_{k_1}\,[]\,.\,?_{k_3}\,[]),\ =^R (N, \Pi z:?_{k_1}\,[]\,.\,?_{k_2}\,[]),\ \in^R_S (?_{k_4}\,[])\}\big)\Big\} & \text{c.c.}
\end{cases}
$$

donde pedimos que $?_{k_1}\,[]$, $?_{k_2}\,[]$, $?_{k_3}\,[]$, $?_{k_4}\,[]$ sean incógnitas nuevas y además que z sea una variable nueva. †

Como podemos apreciar, la función g_{Ab} genera las premisas y restricciones que se deberían satisfacer para que un juicio sea derivado a partir de la aplicación de la regla *Abstraction*. Al igual que lo ocurrido para el caso de la función g_{Pr}, optamos por dividir el conjunto de soluciones posibles por casos, según la forma que adopte el juicio sobre el cual estamos interesados en generar la información.

Una vez más, ya que el manejo de incógnitas que contienen sustituciones se encuentra más limitado que el de *incógnitas* sin sustituciones asociadas, tratamos de evitar la primer clase de expresión tanto cuanto podamos. En este caso, para lograrlo, analizamos la estructura que presenta el juicio recibido como argumento por la función g_{Ab}. Como sabemos, una de las situaciones que conduce a la obtención de incógnitas con sustituciones se da cuando queremos comparar dos abstracciones o productos dependientes entre sí. Debido a que la variable capturada en ambos casos puede diferir de una expresión a otra, por lo general para verificar la igualdad entre tales expresiones debemos efectuar una sustitución de variables sobre una de las expresiones analizadas. Si ésta contiene incógnitas, podemos potencialmente finalizar con la clase de expre-

siones que estamos tratando de evitar. Para no caer en esta situación, dividimos el comportamiento de nuestra función de acuerdo a los siguientes casos:

- Si el término y el tipo que forman parte del juicio ya tienen la forma de una abstracción y un producto dependiente, entonces empleamos las mismas expresiones que forman parte de éstos para construir las premisas y restricciones que necesitamos.
- Si sólo una de las expresiones se encuentra parcialmente definida (presenta la forma de una abstracción o un producto), entonces se emplean las expresiones de ésta y se generan incógnitas sólo para la expresión que no posee la estructura en que estamos interesados.
- Si ni el término ni el tipo denotan una abstracción y un producto respectivamente, entonces se generan incógnitas para definir ambas expresiones.

Desde luego que, mientras más definidas se encuentren las expresiones involucradas, menos incógnitas deberemos generar y menos restricciones deberán ser verificadas.

Definición 4.47 (generación de premisas para la regla de *Conversion*).
Definimos la función generadora de premisas para la regla de *Conversion* como:

$$g_{Co} : J_A \to \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr)\big)\Big)$$

la cual esta dada por:

$$\begin{aligned} g_{Co}(\Gamma \vdash M : N) \quad = \quad & \Big\{\big(Conversion, \\ & [\Gamma \vdash M : ?_{k_1}[], \Gamma \vdash N : ?_{k_2}[]], \\ & \{\in^R_S (?_{k_2}[]), =^R_\beta (N, ?_{k_1}[])\}\big)\Big\} \end{aligned}$$

donde pedimos que $?_{k_1}[]$, $?_{k_2}[]$ sean incógnitas nuevas. †

La función g_{Co} es la encargada de generar el conjunto de premisas y restricciones que deben ser verificadas para que un juicio pueda ser derivado a través de la regla de *Conversion*. Para expresar ésto, dado el juicio $\Gamma \vdash M : N$ se generan dos nuevas incógnitas, la $?_1[]$ y la $?_2[]$. Establecemos además que se debería demostrar que M tiene tipo $?_1[]$ y que además esta incógnita es β-equivalente a la expresión N. Finalmente declaramos que para que la regla sea aplicable deberíamos ser capaces de verificar que N tiene tipo *sort*.

Definición 4.48 (generación de premisas y restricciones).
Definimos la función generadora de premisas y restricciones para juicios *gen* como:

$$gen : J_A \to \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr)\big)\Big)$$

la cual se encuentra definida por:

$$\begin{array}{rcll} gen\ (\Gamma \vdash M : N) & = & g_{Ax}(\Gamma \vdash M : N) & \cup \\ & & g_{St}(\Gamma \vdash M : N) & \cup \\ & & g_{Wk}(\Gamma \vdash M : N) & \cup \\ & & g_{Ap}(\Gamma \vdash M : N) & \cup \\ & & g_{Pr}(\Gamma \vdash M : N) & \cup \\ & & g_{Ab}(\Gamma \vdash M : N) & \cup \\ & & g_{Co}(\Gamma \vdash M : N) & \dagger \end{array}$$

Esta nueva función no hace más que combinar los resultados obtenidos por el resto de las funciones generadoras que definimos anteriormente. Así, si para un juicio dado existen múltiples reglas que nos permitan derivarlo, entonces todos los posibles caminos de derivación se verán reflejados como elementos del conjunto retornado por la función *gen*. Desde luego que sólo nos interesarán aquellas premisas que se encuentren asociadas a restricciones que no resulten falsas. Por ese motivo, necesitaremos de un procedimiento que nos permita filtrar las soluciones que podrían ser verificadas de aquéllas que no podrán serlo.

Definición 4.49 (filtrado de soluciones).
Definimos la función de filtrado de soluciones obtenidas a través de la generación de premisas como:

$$\begin{array}{l} fSols\ :\ \mathscr{P}\Big(Constr\Big) \times \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr)\big)\Big) \rightarrow \\ \qquad \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr), [Subst^?], \mathscr{P}(Constr)\big)\Big) \end{array}$$

dada por:

$$fSols\ (gc, \{e_1, \ldots, e_k, (r, js, cs)\}) = \begin{cases} \big\{(r, js/subs, cs_{new}, subs, gc_{new})\big\} \cup fSols\ (gc, \{e_1, \ldots, e_k\}) & \text{si } ([], gc \cup cs) \overset{\text{check}}{\Longrightarrow} (subs,\ gc_{new}) \\ fSols\ (gc, \{e_1, \ldots, e_k\}) & \text{c.c.} \end{cases}$$

$$fSols\ (gc, \varnothing) = \varnothing$$

donde:

$$([], cs) \overset{\text{check}}{\Longrightarrow} (\hat{\delta_1}, \ldots, \hat{\delta_m}, cs_{new}) \qquad \dagger$$

Básicamente, esta nueva función lo que nos provee es de un mecanismo de filtrado de soluciones obtenidas a través de la función *gen*. La función $fSols$ recibe dos argumentos. El primero denota el conjunto de restricciones globales que debe satisfacer la derivación en la cual se ubica el juicio para el cual estamos intentando calcular las premisas. El segundo elemento es un conjunto de posibles soluciones obtenidas a través de *gen*. Para efectuar el filtrado de soluciones, lo que hacemos es, para cada una de las potenciales soluciones, tomamos las restricciones que se corresponden a dicho caso, combinamos éstas con las restricciones globales y si este nuevo conjunto resulta verdadero o al menos reducible, entonces tomamos dicha posible solución como válida.

La función retorna como resultado un conjunto de soluciones cuyas restricciones han demostrado no ser falsas. Cada elemento de dicho conjunto está compuesto por:

- La regla por la cual se puede deducir el juicio original a partir de la lista de premisas que se está retornando.
- La lista de premisas a partir de la cual se puede derivar el juicio para el cual estamos generando premisas y restricciones.
- El conjunto de restricciones que se añadirán al conjunto de restricciones vigentes en la derivación al cual pertenecía el juicio original.
- La lista de sustituciones de incógnitas que se pudieron deducir a través del proceso de verificación de restricciones.
- El nuevo conjunto de restricciones que se deben verificar en la derivación sobre la cual se está trabajando.

Algo que vale la pena hacer notar, es que la lista de sustituciones de incógnitas que fueron deducidas ya son aplicadas a la lista de premisas, de modo que una vez que decidamos qué solución del conjunto retornado por $fSols$ usaremos, sólo debemos aplicar dichas sustituciones al resto de los juicios que conforman la derivación.

Empleando esta última función, estamos en condiciones de dar la función generadora de premisas y restricciones verificables para un juicio dado.

Definición 4.50 (generación de premisas y restricciones verificables).
Dada la derivación $(dt, \{c_1, \ldots, c_n\})$, y un juicio j, tal que $j \in dt$, definimos la función generadora de premisas y restricciones verificables para dicho juicio como:

$$\begin{aligned} gFindSols \quad : \quad & \mathscr{P}(Constr) \times J_A \rightarrow \\ & \mathscr{P}\Big(\big(Rules, [J_A], \mathscr{P}(Constr), [Subst^?], \mathscr{P}(Constr)\big)\Big) \end{aligned}$$

dada por:

$$gFindSols\ (cs,\ j) \quad = \quad fSols\ (cs, gen\ (j)) \qquad \dagger$$

Esta última función es muy sencilla y nos sirve para obtener los elementos en los que estábamos interesados desde un comienzo. Esto es, dado un conjunto de restricciones iniciales y un juicio, nos retorna un conjunto de tuplas que representan cada uno de los posibles caminos de derivación que nos pueden conducir a dicho juicio (en un solo paso de derivación) junto a las restricciones que deberemos verificar y los cambios que introduce cada uno de ellos.

4.4.2 Eliminación de premisas

En la sección anterior vimos los mecanismos que empleábamos para generar premisas y restricciones a partir de un juicio dado. En esta nueva sección nos concentraremos en el proceso inverso, es decir, los mecanismos con que contamos para eliminar las premisas de un juicio dado. Pero como veremos, no sólo estaremos interesados en borrar dichas premisas de nuestra derivación, sino que además requeriremos que toda la información deducida a partir de las mismas pueda ser igualmente suprimida.

Una primera aproximación a la solución de este tipo de problemas, consistiría en simplemente deshacernos de las premisas en cuestión. De todas formas, si hiceramos

esto no estaríamos descartando la información que fue adquirida a partir de la generación de las mismas. Para poner un ejemplo, supongamos que bajo la especificación de *PTS* para el sistema λC, tenemos la siguiente derivación:

$$\frac{}{[] \vdash ?_0[] \ : \star}$$

Evidentemente, podríamos comenzar a construir nuestra derivación aplicando, entre otras posibilidades, la regla de *product* o de *application*. Supongamos que optamos por la primer opción, de modo que la derivación resultante sería de la forma:

$$\frac{[] \vdash ?_1[] \ : ?_3[] \qquad [x_1 :?_1[]] \vdash ?_2[] \ : ?_4[]}{[] \vdash \Pi x_1 :?_1[] \,.\, ?_2[] \ : \star}\ \text{pr}$$

Donde además aún nos restaría por satisfacer el siguiente conjunto de restricciones:

$$\left\{\in_R^R (?_3[], ?_4[], \star), \in_S^R (?_3[]), \in_S^R (?_4[])\right\}$$

Pero más importante, resulta notar que la incógnita $?_0[]$ ha sido sustituida por $\Pi x_1 : ?_1[] \,.\, ?_2[]$, de modo que si ahora quisiéramos eliminar las premisas generadas, para aplicar digamos, la regla de *application*, y nuestro proceso de eliminación se limitara sólo a eliminar las premisas, obtendríamos como resultado la derivación:

$$\frac{}{[] \vdash \Pi x_1 :?_1[] \,.\, ?_2[] \ : \star}$$

A partir de la cual, evidentemente no podemos aplicar la regla de *application* tal y como sucedió en un comienzo. Más aún, el conjunto de restricciones se mantienen, aún cuando éstas ya han dejado de ser requeridas. Todo esto se deba a que, simplemente eliminando las premisas, no nos hemos deshecho de la información aprendida a través del uso de la regla de *product*.

Una segunda aproximación consistiría en recalcular las restricciones y premisas para cada uno de los juicios que se encuentran entre la raíz de la derivación y el juicio para el cual estamos eliminando sus premisas. Pero esta nueva táctica tampoco nos brinda los resultamos esperados, ya que podría suceder que en algún punto estemos utilizando las reglas *application*, *abstraction* o *product*, lo cual nos diseminaría el problema en varias de las ramas que forman parte del árbol de derivación.

Pero incluso algo más problemático que ésto consiste en determinar cual es el nuevo conjunto de restricciones que debería ser satisfecho. Esto se debe a que nada nos garantiza que dicho conjunto sea en realidad un subconjunto del conjunto actual, ya que algunas de las restricciones actuales podrían haber sido obtenidas a partir del análisis de otras más complejas que ya han dejado de existir como una unidad de restricción.

De modo que el único camino que nos queda por considerar consiste en efectuar la reconstrucción total de la derivación, para lo cual requeriremos de las funciones generadoras definidas en la sección anterior. Para llevar a cabo la eliminación de premisas, procederemos de la siguiente manera.

Primero, tomaremos el árbol de derivación del cual partimos, y quitamos los nodos correspondientes a las premisas del juicio sobre el cual estamos aplicando la eliminación de premisas. Luego, empleando la información que nos brindan las reglas que empleamos en cada uno de los pasos de derivación que aún se conservan en nuestro árbol, estaremos en condiciones de reconstruir parcialmente desde cero el nuevo árbol de derivaciones, lo cual en el proceso añadirá algunas restricciones de las que ya teníamos antes de efectuar el borrado de premisas.

Debido a que las premisas que eliminamos solo podrían haber introducido información a la derivación modificando el valor de las incógnitas presentes en el juicio al cual derivaban, tomaremos el conjunto de *incógnitas* que ocurren en éste como nuestro conjunto de incógnitas "interesantes". Más adelante veremos el tratamiento que le damos a éstas. Finalmente, empleando la información de las expresiones y contextos aun contenidos en el viejo árbol de derivaciones, podremos completar algunos de los detalles que nos queden en nuestro nuevo árbol, obteniendo así un árbol similar al original, salvo los juicios y restricciones introducidos por las premisas que eliminamos.

Para que todo este proceso se pueda llevar a cabo, requeriremos de una función auxiliar que nos provea de las premisas y restricciones que se corresponden a un juicio dado a través de una regla particular. Algo similar ya habíamos considerado al momento de definir las funciones generadoras de premisas en la sección anterior, de modo que emplearemos las funciones que definimos en ese entonces. Lo que necesitamos ahora es de una función que nos permita introducir la regla que estamos interesados utilizar como un argumento de la misma.

Notación 4.10 (proyecciones):
Dada un n-upla de la forma $(e_1,\ldots,e_n)$, definimos la función de proyección del i-ésimo elemento de la misma a través de la función π_i, de modo que:

$$\pi_i\ (e_1,\ldots,e_n)\ =\ e_i \qquad \diamondsuit$$

Definición 4.51 (*appRule*, aplicación de reglas a juicios).
Dado un juicio $j \in J_A$ y una regla $rg \in Rules$, definimos la función:

$$appRule\ :\ J_A \times Rules \to \mathscr{P}\Big(\big([J_A],\ \mathscr{P}(Constr)\big)\Big)$$

la cual se encuentra dada por:

$$appRule\ (j,r) \quad = \quad (\lambda sol\,.\,(\pi_2\ sol, \pi_3\ sol))\ solSet$$

donde:

$$solSet = \begin{cases} g_{Ax}\ (j) & \text{si } r = Axiom \\ g_{St}\ (j) & \text{si } r = Start \\ g_{Wk}\ (j) & \text{si } r = Weakening \\ g_{Ap}\ (j) & \text{si } r = Application \\ g_{Pr}\ (j) & \text{si } r = Product \\ g_{Ab}\ (j) & \text{si } r = Abstraction \\ g_{Co}\ (j) & \text{si } r = Conversion \end{cases}$$

y además dada una tupla $(e_1,\ldots,e_n)$, denotaremos por π_i la proyección del i-ésimo elemento de dicha tupla. †

Aquí no debemos confundir la notación λ que empleamos para construir una función con el λ que empleamos como constructor de expresiones en E_A. De aquí en adelante se tornara algo frecuente el uso de la notación λ en la definición de funciones, por lo que deberemos prestar especial atención para poder diferenciar cuando se esta empleando en el contexto de E_A y cuando no.

Una de las etapas fundamentales que forman parte de la reconstrucción de una derivación consiste en armar un árbol de derivación empleando solamente la información provistas por las reglas contenidas en el antiguo árbol de derivación. Esto es, dado un árbol de reglas y un juicio, deberíamos ser capaces de emplear dicha información para intentar reconstruir, al menos en forma parcial, una derivación que contenga como raíz el juicio del cual partimos.

Definición 4.52 (construcción de derivaciones a partir de árboles de reglas).
Dado un árbol formado por reglas del conjunto *Rules* y un juicio $j \in J_A$, definimos la función:

$$mkDerRule \ : \ Tree\ Rules \times J_A \rightarrow Derivation_{\bowtie}$$

dada por:

$$mkDerRule\ ((r,[r_1,\ldots,r_k]),\ j) \quad = \quad \begin{cases} \Big(((j',r),\ st'),cs_{new}\Big) & \text{si } appRule\ (j,r) = \{([p_1,\ldots,p_k],cs_{judge})\} \\ & \quad \wedge\ ([],cs_{judge}) \overset{\text{check}}{\Longrightarrow} (\vec{\delta},cs_0) \\ & \quad \wedge\ ([],cs_1 \cup \ldots \cup cs_k \cup cs_0) \overset{\text{check}}{\Longrightarrow} (\vec{\delta}',cs_{new}) \\ \Big(((j,r),[]),\varnothing\Big) & \text{si } appRule\ (j,r) = \varnothing \\ \bowtie & \text{c.c.} \end{cases}$$

donde:

$$\begin{array}{rcl} [(dt_1,cs_1),\ldots,(dt_k,cs_k)] & = & [mkDerRule\ (r_1,{p_1}'),\ldots,mkDerRule\ (r_k,{p_k}')] \\ j' & = & (j\ /\ \vec{\delta})\ /\ \vec{\delta}' \\ {p_i}' & = & p_i\ /\ \vec{\delta} \qquad \forall i \in \{1,\ldots,k\} \\ st' & = & [dt_1/\vec{\delta}',\ldots,dt_k/\vec{\delta}'] \end{array}$$

†

Como dijimos, esta función nos permite obtener derivaciones a partir de un juicio si es que conocemos la secuencia de reglas que aplicaremos en cada paso de dicha derivación. Para lograrlo, hacemos uso de la función auxiliar *appRule* que nos permitía obtener la lista de premisas y el conjunto de restricciones generadas al aplicar una regla sobre un juicio. Aquí se hace evidente porqué al definir las funciones generadoras de premisas pedíamos que las mismas se retornaran en una lista y no un conjunto. Como tales funciones retornan las premisas siempre en el mismo orden, resulta posible emplear la información de las reglas empleadas en una derivación, para construir una derivación similar.

En nuestra última función, al recibir un árbol de reglas junto a un juicio a partir del cual estamos interesados en construir la nueva derivación, analizamos los resultados posibles de aplicar la regla raíz al juicio recibido. Si no existe ninguna premisa posible para tal juicio, entonces retornamos la derivación que consta de un solo paso de derivación conformado por el juicio y la regla en cuestión, sin ninguna premisa posible.

Por otro lado, si al aplicar la regla raíz (r) al juicio que recibimos (j) obtenemos una posible solución (recordemos que de acuerdo a nuestra definición 4.51, dicho conjunto consta de un solo elemento), emplearemos entonces la información que ésta

nos brinda para generar un nuevo paso en la derivación. Al aplicar *appRule*, obtendremos entonces la lista de premisas para el juicio j, junto a las restricciones que se deben verificar en el paso que estamos generando. Si este conjunto de restricciones es al menos reducible, entonces obtendremos un conjunto de condiciones más sencillas junto a una secuencia de sustituciones ya deducidas. Si aplicamos dichas sustituciones a las premisas generadas a través de *appRule*, y empleamos la información del árbol de reglas, entonces recursivamente podemos obtener los arboles de derivación y las restricciones que deben satisfacer cada una de las derivaciones que conducen a las premisas de j empleando la regla r.

Si las restricciones que habíamos obtenido en un primer momento, junto a las nuevas que obtenemos por recursión son verificables, entonces no tenemos más que aplicar las sustituciones que hayamos deducido sobre los árboles de derivación y el juicio j. Con las sustituciones ya aplicadas según corresponda, podemos emplear los árboles obtenidos para construir la nueva derivación y complementar a ésta con el resulta de reducir el conjunto formado por todas las restricciones que habíamos obtenido.

Empleando esta nueva función, estamos en condiciones de definir un mecanismo que nos permita duplicar una derivación empleando solo la información que nos proveen las reglas empleadas en cada uno de los pasos que constituye la misma. Pero antes dejaremos en claro a qué nos estamos refiriendo cuando hablamos de efectuar un mapeo sobre un árbol finitario.

Definición 4.53 (mapeo de árboles finitarios).
Definimos el mapeo sobre árboles finitarios a través de la función:

$$tMap : (a \to b) \times Tree\ a \to Tree\ b$$

dada por:

$$tMap\ (f, (e, [st_1, \ldots, st_n])) = (f\ e, [tMap\ (f, st_1), \ldots, tMap\ (f, st_n)]) \qquad \dagger$$

Definición 4.54 (reconstrucción de derivaciones a través de sus reglas).
Dada una derivación (dt, cs), definimos la función reconstructora de derivaciones:

$$rebuildDer : Derivation \to Derivation_{\bowtie}$$

definida como:

$$rebuildDer\ ((\Gamma \vdash M : N, r), st) = mkDerRule\ (tMap\ (\pi_2, dt), \Gamma \vdash ?_0[] : N)$$

donde:

$$dt \quad = \quad (\Gamma \vdash M : N, st) \qquad \dagger$$

Básicamente, el parámetro $tMap\ (\pi_2, dt)$ nos permite construir un árbol conformado por las reglas que se emplean en la derivación original. Una vez que tenemos dicho árbol, ya estamos en condiciones de utilizarlo para reconstruir (al menos parcialmente) una nueva derivación empleando tal información. El problema aquí se traduce entonces en determinar cuál es el juicio del cual debemos partir en la construcción de la nueva derivación.

Resulta claro que, debido a que el término que forma parte del juicio es el componente principal que debemos determinar y cuya estructura resulta desconocida en un primer momento, podemos descartar la información que nos brinda éste y reemplazarlo por una incógnita, en nuestro caso $?_0\,[\,]$. Con el contexto y el tipo que conforman el juicio no ocurre lo mismo. Es decir, estos dos componentes forman parte de la información inicial con que contamos para desarrollar nuestra derivación, de modo que éstos no pueden ser descartados.

Si bien la ocurrencia de incógnitas puede darse en cualquier trozo de la derivación, resulta cierto el hecho de que desde un comienzo conocemos completamente el contexto y el tipo (teorema) que usaremos en nuestra derivación. Más aún, debido a que el contexto y tipo son conocidos completamente desde que comienza el proceso de derivación, podemos estar seguros que al menos éstos no contienen incógnitas y por lo tanto no han adquirido nueva información por medio de la deducción de reglas que hayamos efectuado. De este modo, podemos emplearlos sin ninguna preocupación como el contexto y la expresión que denota el tipo desde los cuales iniciaremos la construcción de la nueva derivación.

Un punto a destacar es que, como ya debe resultar evidente, los nombres de variables e incógnitas presentes en la nueva derivación, no necesariamente deberían ser los mismos que los que ocurren en la derivación de la cual partimos. Más aún, en la derivación original es posible que algunas incógnitas hayan sido asignadas a valores mediante la intervención directa del usuario, y este tipo de información no puede ser generada a partir del uso exclusivo de las reglas. Para resolver este tipo de inconvenientes necesitaremos de la ayuda de algunas funciones auxiliares cuya tarea será la de completar el nuevo árbol de derivación empleando información del original.

Definición 4.55 (chequeo de sustituciones).
Definimos la función de verificación de sustituciones:

$$textSubs \;:\; Subst^{?} \;\to\; Subst^{?}_{\bowtie}$$

dada por:

$$testSubs\;([E_1/i_1,\ldots,E_n/i_n]) \;=\; \begin{cases} testSubs\;([E_1/i_1,\ldots,E_{k-1}/i_{k-1},E_{k+1}/i_{k+1},\ldots,E_n/i_n]) \\ \qquad \text{si } \exists j,k \in \{1,\ldots,n\} \;:\; j<k \wedge i_j = i_k \wedge E_j =_\alpha E_k \\ \bowtie \quad \text{si } \exists j,k \in \{1,\ldots,n\} \;:\; j<k \wedge i_j = i_k \wedge E_j \neq_\alpha E_k \\ [E_1/i_1,\ldots,E_n/i_n] \quad \text{c.c.} \end{cases}$$

†

Como ya mencionamos anteriormente, al reconstruir una derivación existe un conjunto de incógnitas que consideramos interesantes. Al analizar cada uno de los elementos que conforma la derivación podremos deducir algunos valores a los cuales pueden ser asignados estas incógnitas. Una vez que hayamos estudiado todos ellos, necesitaremos combinar los resultados hayados en una única solución. La función *testSubs* nos provee este mecanismo. Dicho de otro modo, combinando todas las asignaciones que hayamos podido deducir, la nueva función verifica que no existan asignaciones duplicadas. En caso de que se encuentre dos incógnitas con asignaciones, se chequea que tales asignaciones sean α-equivalentes.

Ahora que ya hemos determinado en qué manera unificaremos los resultados de asignaciones interesantes que vayamos encontrando, estamos en condiciones de definir

las funciones que nos permitirán completar nuestro nuevo árbol de derivaciones empleando la información que nos brinda el árbol original. Para ello, comenzaremos por exponer la función encargada de combinar dos expresiones de E_A.

Definición 4.56 (combinación de expresiones).
Definimos la función de combinación de expresiones:

$$mergeExpr \;:\; \mathscr{P}(\mathbb{N}) \times Subst \times E_A \times E_A \to (E_A, Subst^?)_{\bowtie}$$

dada por:

$$mergeExpr\ (ks,\delta,s_1,s_2) = \begin{cases} (s_1,[]) & \text{si } s_1 = s_2 \\ \bowtie & \text{c.c.} \end{cases}$$

$$mergeExpr\ (ks,\delta,v_1,v_2) = \begin{cases} (v_1,[]) & \text{si } v_1 = v_2/\delta \\ \bowtie & \text{c.c.} \end{cases}$$

$$\begin{aligned} mergeExpr\ (ks,\delta,(M_1\ N_1),(M_2\ N_2)) = {} & (\lambda(M',\delta^?_M).(\lambda(N',\delta^?_N).(\lambda\delta^?.(M'\ N',\delta^?))_{\bowtie} \\ & \quad testSubs\ (\delta^?_M \# \delta^?_N))_{\bowtie} \\ & \quad\quad mergeExpr\ (ks,\delta,N_1,N_2))_{\bowtie} \\ & \quad\quad\quad mergeExpr\ (ks,\delta,M_1,M_2) \end{aligned}$$

$$\begin{aligned} mergeExpr\ (ks,\delta,(\lambda x_1 : M_1 . N_1),\lambda x_2 : M_2 . N_2) = {} & (\lambda(M',\delta^?_M).(\lambda(N',\delta^?_N).(\lambda\delta^?.(\lambda x_1 : M' . N',\delta^?))_{\bowtie} \\ & \quad testSubs\ (\delta^?_M \# \delta^?_N))_{\bowtie} \\ & \quad\quad mergeExpr\ (ks,\delta\#[x_1/x_2],N_1,N_2))_{\bowtie} \\ & \quad\quad\quad mergeExpr\ (ks,\delta,M_1,M_2) \end{aligned}$$

$$\begin{aligned} mergeExpr\ (ks,\delta,(\Pi x_1 : M_1 . N_1),\lambda x_2 : M_2 . N_2) = {} & (\lambda(M',\delta^?_M).(\lambda(N',\delta^?_N).(\lambda\delta^?.(\Pi x_1 : M' . N',\delta^?))_{\bowtie} \\ & \quad testSubs\ (\delta^?_M \# \delta^?_N))_{\bowtie} \\ & \quad\quad mergeExpr\ (ks,\delta\#[x_1/x_2],N_1,N_2))_{\bowtie} \\ & \quad\quad\quad mergeExpr\ (ks,\delta,M_1,M_2) \end{aligned}$$

$$mergeExpr\ (ks,\delta,M,?_k[]) = \begin{cases} (?_k[],[M/k]) & \text{si } k \in ks \\ (M,[]) & \text{c.c.} \end{cases}$$

$$mergeExpr\ (ks,\delta,M,?_k[\delta_1,\ldots,\delta_n]) = \begin{cases} (?_k[[\delta_1/\delta,\ldots,\delta_n/\delta]],[]) & \text{si } k \in ks \\ (M,[]) & \text{c.c.} \end{cases}$$

$$mergeExpr\ (ks,\delta,M,N) = \bowtie \quad \text{para todo otro caso}$$

donde si para cada $i \in \{1,\ldots,n\}$, tenemos que:

$$\delta_i = [E^i_1/x^i_1,\ldots,E^i_l/x^i_l]$$

δ_i/δ significará:

$$[(E_1^i/\delta)/x_1^i, \ldots, (E_l^i/\delta)/x_l^i] \qquad \dagger$$

Tratemos ahora de comprender un poco que hace nuestra función *mergeExpr*. Como ya dijimos, ésta sera la encargada de complementar la información faltante de las expresiones que forman parte de la nueva derivación con la información brindada por las expresiones que componen la derivación original. Para entender su comportamiento, primero veamos los parámetros que ésta recibe. En primer lugar toma un conjunto de naturales, el cual declara el conjunto de identificadores de incógnitas que se consideran interesantes en la derivación que estamos replicando. El segundo parámetro denota las sustituciones de identificadores (variables) que hemos podido determinar hasta el momento. Los últimos dos argumentos representan las expresiones a ser combinadas. El primero se corresponde con la expresión existente en la derivación original y el segundo con la generada. Como resultado, retorna un par compuesto por la expresión resultante de la combinación y la sustitución de identificadores interesantes que se haya logrado deducir del proceso de combinación.

Como ya dijimos, el proceso de construcción de una derivación a partir de un árbol de reglas, nos construye una derivación idéntica a la original excepto renombre de variables, identificadores de incógnitas y valores asignados a ellas. Con esta nueva función, planteamos una solución a este problema, proveyendo de un mecanismo de combinación de expresiones.

Si las expresiones analizadas se corresponden con dos *sorts*, entonces ambas expresiones resultan combinables y la expresión retornada es uno de estos *sorts*. Lo mismo ocurre en el caso de las variables, con la salvedad de que debemos verificar que la variable perteneciente a la expresión original sea idéntica a la variable generada aplicando la sustitución de identificadores que ya hemos logrado deducir.

El caso de la aplicación, abstracción y producto dependiente son muy similares entre si. Básicamente lo que se hace es combinar las subexpresiones intervinientes y unificar las asignaciones de incógnitas interesantes que hayamos encontrado. Una vez hecho esto, podemos construir una expresión como resultado de la combinación de los dos parámetros recibidos. Si en alguna etapa de la combinación se llega a una no solución ($\bowtie$), entonces éste símbolo es retornado.

En el caso en que la expresión generada se corresponda con una incógnita sin sustituciones, debemos verificar si el identificador de la misma se corresponde con una incógnita interesante o no. Si lo es, entonces retornamos la misma incógnita, pero declaramos una sustitución que indica que dicha incógnita puede ser sustituida por la expresión contra la cual fue comparada. Si la incógnita no resulta interesante, entonces resulta evidente que podemos sustituir la incógnita por la expresión original.

Por otro lado, si la incógnita se presenta junto a una lista de sustituciones y el identificador de la misma se encuentra en el conjunto de incógnitas interesantes, entonces no podremos determinar que efectivamente dicha incógnita pueda ser sustituida por la expresión con la cual la estamos comparando. Sin embargo, al menos podemos aplicar la sustitución de variables que hayamos podido deducir a las sustituciones que forman parte de la incógnita. En cualquier otro caso, la combinación no resulta factible.

Siguiendo el mismo principio, podemos ir un poco mas lejos, y declarar una función similar que trabaje sobre los contextos.

Definición 4.57 (combinación de contextos).
Definimos la función de combinación de contextos:

$$mergeContext \ : \ \mathscr{P}(\mathbb{N}) \times C_A \times C_A \rightarrow (C_A, Subst^?, Subst)_{\bowtie}$$

dada por:

$$\begin{array}{lcl} mergeContext\ (ks,\ (\Gamma, x:M),\ (\Delta, y:N)) & = & (\lambda(\Theta, \delta_C^?, \delta_C).\ (\lambda(E, \delta_E^?).\ (\lambda\delta^?.\ ((\Theta, x:E), \delta^?, \delta_C \# [x/y]))_{\bowtie} \\ & & \quad testSubs\ (\delta_C^? \# \delta_E^?))_{\bowtie} \\ & & \quad\quad mergeExpr\ (ks, \delta_C, M, N))_{\bowtie} \\ & & \quad\quad\quad mergeContext\ (ks, \Gamma, \Delta) \\ mergeContext\ (ks,\ [],\ []) & = & ([],\ [],\ []) \end{array}$$

†

Esta nueva función se comporta de manera similar a la que definimos anteriormente para la combinación de expresiones. De todas formas existen algunas diferencias que vale la pena rescatar. En primer lugar, los parámetros y el valor retornado presentan algunos cambios. A diferencia de la función para expresiones, la referida a contextos no recibe como argumento una sustitución de identificadores. Esto se debe a que, de acuerdo a lo establecido por el lema 2.1, las variables que ocurren en un contexto se encuentran autocontenidas en el mismo. Por otro lado, se añade un nuevo elemento a la tupla retornada. Éste se corresponde con una sustitución de identificadores, las cuales pueden ser deducidas al comparar ambos contextos.

Algo que puede que aun no resulte del todo evidente, es que varios de los supuestos que utilizamos (como el orden de las premisas y su correspondencia con las reglas ya generadas o el orden de los elementos que forman parte de un contexto) se encuentran fundamentados sobre el hecho de que las funciones generadoras retornan para casos idénticos, resultados iguales respetando incluso el orden de éstos.

Volviendo al comportamiento particular de la función *mergeContext*, vemos que primero intenta combinar los tramos iniciales de los contextos comparados. Si ésta puede ser llevada a cabo sin ningún inconveniente, se procede a realizar la combinación de las expresiones que forman parte del último par que compone los contextos recibidos como argumentos. Si esta nueva combinación resulta igualmente exitosa, se verifica que las asignaciones de incógnitas que se hayan podido deducir no se contradigan entre si, retornándose el contexto combinado, junto a la sustitución de incógnitas resultante. Además se devuelve una sustitución que contiene las asignaciones de identificadores que se deben efectuar para convertir los elemento del dominio del contexto generado en elementos del dominio del contexto original.

Yendo un nivel más arriba, podemos definir la función encargada de combinar el juicio original junto al generado.

Definición 4.58 (combinación de juicios).
Definimos la función de combinación de juicios:

$$mergeJudgement \ : \ \mathscr{P}(\mathbb{N}) \times J_A \times J_A \rightarrow (J_A, Subst^?)_{\bowtie}$$

dada por:

$$
\begin{array}{l}
mergeJudgement\ (ks,\ \Gamma_1 \vdash M_1 : N_1,\ \Gamma_2 \vdash M_2 : N_2) \quad = \\
\quad (\lambda(\Theta, \delta^?_C, \delta_C).\ (\lambda(M', \delta^?_M).\ \lambda(N', \delta^?_N).\ (\lambda\delta^?.\ (\Theta \vdash M' : N', \delta^?))_{\bowtie} \\
\quad\quad testSubs\ (\delta^?_C \# \delta^?_M \# \delta^?_N))_{\bowtie} \\
\quad\quad\quad mergeExpr\ (ks, \delta_C, N_1, N_2))_{\bowtie} \\
\quad\quad\quad\quad mergeExpr\ (ks, \delta_C, M_1, M_2))_{\bowtie} \\
\quad\quad\quad\quad\quad mergeContext\ (ks, \Gamma_1, \Gamma_2)
\end{array}
$$

†

A esta altura ya debería resultar evidente el comportamiento que presenta esta nueva función. La única diferencia que presenta respecto a las que vimos anteriormente es que, como ésta neva función se encarga de los juicios, primero efectúa una combinación de los contextos involucrados, luego de las expresiones y si no existe ninguna contradicción, entonces construye un nuevo juicio que conserva la forma del juicio generado con la información provista por el juicio original.

Siguiendo el mismo razonamiento, podemos extender la noción de combinación a los árboles de derivación.

Definición 4.59 (combinación de árboles de derivación).
Sea DT el conjunto de los árboles de derivación que dimos en la definición 4.37. Definimos entonces la función de combinación de árboles de derivación:

$$mergeDerTree : \mathscr{P}(\mathbb{N}) \times DT \times DT \rightarrow (DT, Subst^?)_{\bowtie}$$

dada por:

$$
\begin{array}{l}
mergeDerTree\ (ks,\ ((j,r),\ [st_1,\ldots,st_n]),\ ((j',r'),\ [st_1{}',\ldots,st_n{}'])) = \\
\qquad (\lambda(j_{new},\ \delta^?_{judge}). \\
\qquad\quad (\lambda(dt_1,\ \delta^?_1). \\
\qquad\qquad \vdots \\
\qquad\qquad (\lambda(dt_n,\ \delta^?_n). \\
\qquad\qquad\quad (\lambda\delta^?. \\
\qquad\qquad\qquad (((j_{new}, r),\ [dt_1,\ldots,dt_n]),\ \delta^?))_{\bowtie} \\
\qquad\qquad\qquad\quad testSubs\ (\delta^?_1 \# \ldots \# \delta^?_n \# \delta_{judge}))_{\bowtie} \\
\qquad\qquad\qquad\qquad mergeDerTree\ (ks,\ st_n,\ st_n{}'))_{\bowtie} \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\quad mergeDerTree\ (ks,\ st_1,\ st_1{}'))_{\bowtie} \\
\qquad\qquad\qquad\qquad\qquad mergeJudgement\ (ks,\ j,\ j')
\end{array}
$$

†

Como ya dijimos, los juicios pueden ser procesados como unidades independientes una de las otras, por tal motivo, nuestra función combinadora de árboles de derivación

no tiene que hacer más que combinar los juicios raíz de ambos árboles y luego llamarse recursivamente sobre cada uno de los árboles que descienden del juicio raíz. Si tras analizar todos los subárboles, ninguna sustitución se contradice, entonces podemos retornar el árbol resultante de la combinación, junto a la posible asignación de incógnitas a expresiones.

Con las funciones que hemos dado hasta este momento nos basta para reconstruir un árbol de derivación empleando otro árbol como modelo. De todas formas, para poder reconstruir una derivación, también deberíamos ser capaces de determinar el conjunto de restricciones que deben acompañar a la nueva derivación. Una primera aproximación podría consistir en emplear las funciones generadoras dadas en la sección 4.4.1. Pero para este caso, las funciones definidas anteriormente no nos serán de utilidad. Esto ocurre como consecuencia de que éstas recibían como parámetro un juicio y partir del mismo generaban una lista de premisas junto a un conjunto de restricciones. Ahora, sin embargo, la lista de premisas para cada juicio ya resulta conocida, de modo que sólo requerimos de la generación de premisas dado un juicio junto a sus premisas. Para poder lograrlo, debemos definir una nueva función constructora de restricciones.

Definición 4.60 (construcción de restricciones).
Definimos la función constructora de restricciones:

$$constrG \;:\; [J_A] \times Rules \to \mathscr{P}(Constr)_{\bowtie}$$

dada por:

$$\begin{array}{lcl}
constrG\,([[] \vdash M \;:\; N], Axiom) & = & \{\in^R_S (M), \in^R_S (N), \in^R_A (M,N)\} \\
constrG\,([(\Gamma, x:A \vdash M \;:\; N), & = & \{\in^R_S (N'), =^R (x,M), =^R (A,N), \\
\quad (\Gamma \vdash M' \;:\; N')], Start) & & \quad =^R (M',N)\} \\
constrG\,([(\Gamma, x:C \vdash M \;:\; N), & = & \{\in^R_{V\cup S} (M), \in^R_S (N''), =^R (C,N'') \\
\quad (\Gamma \vdash M' \;:\; N'), & & \quad =^R (M,M'), =^R (N,N')\} \\
\quad (\Gamma \vdash M'' \;:\; N''), Weakening) & & \\
constrG\,([(\Gamma \vdash M \;:\; N), & = & \{=^R (M, \Pi x:A\,.\,M''), =^R (M',A), \\
\quad (\Gamma \vdash M' \;:\; N'), & & \quad \in^R_S (N'), \in^R_S (N''), \in^R_R (N',N'',N)\} \\
\quad (\Gamma, x:A \vdash M'' \;:\; N''), Product) & & \\
constrG\,([(\Gamma \vdash M \;:\; N), & = & \{=^R (M, \lambda x:C\,.\,M'), =^R (N, \Pi x:C\,.\,N'), \\
\quad (\Gamma, x:C \vdash M' \;:\; N'), & & \quad =^R (N,M''), \in^R_S (N'')\} \\
\quad (\Gamma \vdash M'' \;:\; N''), Abstraction) & & \\
constrG\,([(\Gamma \vdash M \;:\; N), & = & \{=^R (M, M'\,M''), =^R (A,N''), \\
\quad (\Gamma \vdash M' \;:\; \Pi x:A\,.\,B), & & \quad =^R (N, B/[M''/x])\} \\
\quad (\Gamma \vdash M'' \;:\; N''), Application) & & \\
constrG\,([(\Gamma \vdash M \;:\; N), & = & \{=^R (M,M'), =^R (N,M''), \in^R_S (N''), \\
\quad (\Gamma \vdash M' \;:\; N'), & & \quad =^R_\beta (N,N')\} \\
\quad (\Gamma \vdash M'' \;:\; N''), Conversion) & & \\
constrG\,([j_1, \ldots, j_n], NoRule) & = & \varnothing
\end{array}$$

y para los casos no contemplados anteriormente decimos que:

$$constrG\,([j_1, \ldots, j_n], r) \;= \bowtie \qquad \dagger$$

El lector atento notará que la lista de juicios que recibe como parámetro esta función se corresponden el juicio y las premisas de éste que podrían ser obtenidos a través de la aplicación de una regla de derivación en particular. El primer elemento de la lista se corresponde con el juicio derivado y el resto de la lista con las premisas de éste. Nótese que aquí, una vez más, resulta importante el orden en que vienen dadas las premisas, ya que se esta haciendo una suposición sobre dicho orden.

Dada una lista de juicios y una regla, esta nueva función nos genera la lista de restricciones generales que se deberían satisfacer en cada caso. Algunas de las restricciones generadas pueden resultar redundantes, pero dado que no sabemos cuán completo se encuentran los juicios analizados, debemos considerar todas las posibilidades. Esto no quita el hecho de que en la mayoría de los casos prácticos, muchas de las restricciones generadas puedan ser descartadas ya que pueden ser verificadas trivialmente.

Haciendo uso de esta nueva función auxiliar, podemos declarar ahora sí una función que nos permita reconstruir la información referida a restricciones para una derivación dada, que si recordamos era el componente que nos estaba faltando a la hora de generar una nueva derivación a partir de un trozo de otra derivación.

Definición 4.61 (recostrucción de restricciones para derivaciones).
Definimos la función de reconstrucción de derivaciones:

$$rebuildConstr \;:\; DT \;\rightarrow\; ([Subst^{?}],\, \mathscr{P}(Costr))$$

dada por:

$$\begin{aligned}
&rebuildConstr\;((j,r),[((p_1,r_1),st_1),\ldots,((p_n,r_n),st_n)]) = \\
&\qquad(\lambda cs.\\
&\qquad\quad(\lambda(\delta_1^{?},cs_1).\\
&\qquad\qquad\vdots\\
&\qquad\quad(\lambda(\delta_n^{?},cs_n).\\
&\qquad\qquad(\lambda\delta^{?}.\\
&\qquad\qquad\quad(\delta^{?},cs_{new}))_{\bowtie}\\
&\qquad\qquad\qquad testSubs\;(\delta_1^{?}\#\ldots\#\delta_n^{?}\#\delta_{new}^{?}))_{\bowtie}\\
&\qquad\qquad\qquad\quad rebuildConstr\;((p_n,r_n),st_n))_{\bowtie}\\
&\qquad\qquad\qquad\qquad\vdots\\
&\qquad\qquad\qquad\quad rebuildConstr\;((p_1,r_1),st_1))_{\bowtie}\\
&\qquad\qquad\qquad\qquad constrG\;([j,p_1,\ldots,p_n],r)
\end{aligned}$$

todo esto si:

$$([],\, cs \cup cs_1 \cup cs_1 \cup \ldots \cup cs_n) \overset{\text{check}}{\Longrightarrow} (\delta_{new}^{?},\, cs_{new})$$

en caso contrario, decimos que:

$$rebuildConstr\;((j,r),[dt_1,\ldots,dt_n]) = \bowtie \qquad \dagger$$

En líneas generales, la función de recostrucción de restricciones para derivaciones recibe como argumento un árbol de derivaciones y a partir de éste construye un conjunto de restricciones que deben ser varificadas para que el árbol pueda ser completado. Para construir tal conjunto, primero se generan las restricciones junto a la lista de sustituciones deducidas tanto para el juicio principal como para los subárboles que conforman las premisas de dicho juicio. Si la operación hasta aquí ha tenido exito, procedemos a unificar las sustituciones que encontramos hasta ese momento junto a las deducidas en el proceso de verificación de las restricciones que hayamos podido generar. El resultado de esta unificación junto al conjunto de restricciones resultantes conforman el valor retornado por la función.

Finalmente, nos encontramos en condiciones de construir una función que nos permita eliminar las premisas de un juicio que forma parte de una derivación, junto a la información que se haya podido deducir a través del uso de tales premisas. En nuestra siguente definición, estamos asumiendo que el argumento *dt* que recibe nuestra función se corresponde con el árbol de derivación, al cual ya se le han quitado las premisas que se deseaba eliminar. Esto no consiste en otra cosa más que remover del árbol de derivación los nodos que contenian la información de tales premisas.

Definición 4.62 (eliminación de premisas en derivaciones).
Definimos la función de eliminación de premisas para derivaciones como:

$$delPart \ : \ Derivation \rightarrow (Derivation, Subst^{?})_{\bowtie}$$

dada por:

$$\begin{aligned} delPart\ (dt, cs) \quad = \quad & (\lambda (dt', cs'). \\ & (\lambda (dt_{new}, \delta^{?}_{dt}). \\ & (\lambda (\vec{\delta}^{?}, cs_{new}). \\ & ((dt_{new} \ / \ \vec{\delta}^{?}, cs_{new}), \delta^{?}_{dt} \ / \ \vec{\delta}^{?}))_{\bowtie} \\ & rebuildConstr\ (dt_{new}))_{\bowtie} \\ & mergeDerTree\ (ks, dt, dt'))_{\bowtie} \\ & rebuildDer\ (dt, cs) \end{aligned}$$

donde *ks* contiene el conjunto de identificadores de incognitas que ocurren en el juicio cuyas premisas estamos eliminando. †

Básicamente, en primer lugar, reconstruimos un nuevo árbol de derivación (dt') haciendo uso de la información que nos proveen las reglas presenten en dt. Luego, extraemos toda la información referida a los contextos y expresiones que forman parte del árbol original y que nos puedan resultar de utilidad. Una vez hecho esto, procedemos a recostruir el conjunto de restricciones que debe satisfacer nuestro nuevo árbol de derivación, retornando dicho conjunto junto al nuevo árbol que hemos generado.

Con lo formalizado hasta el momento, nos alcanza definir nuestro prototipo de asistente y darle a éste algunos mecanismos básicos que le permitan trabajar de manera rudimentaria con expresiones, contextos, juicios, restricciones y derivaciones. En nuestro próximo capítulo daremos una breve descripción de la manera en que las ideas presentadas en este capítulo fueron implementadas en nuestro prototipo.

Capítulo 5

Implementación

En este capítulo explicaremos la implementación que fue llevada a cabo, sobre el prototipo de asistente que fue formalizado en el capítulo anterior. Si bien el diseño planteado resulta propio, la idea general sobre las funcionalidades que debería proveer una herramienta de éstas características, se encuentran inspiradas en el trabajo de Lena Magnusson [Mag94]. Algunas de las funcionalidades y ventajas que ofrece nuestra implementación son:

- Posibilidad de trabajar con múltiples derivaciones bajo diferentes especificaciones de *PTSs*.
- Mecanismos de generación y eliminación de premisas.
- Dos modos de manipulación de derivaciones. Uno manual que permite analizar lo que va ocurriendo con la derivación paso a paso y uno semi-automático que agiliza la tarea de construcción.
- Posibilidad de almacenar el estado de las derivaciones sobre las cuales se esta trabajando, para continuar con las mismas en futuras sesiones.
- Mecanismos de configuración de especificaciones de *PTS*, junto a procedimientos que nos permiten emplear tales especificaciones en cualquier derivación que requiramos.
- Generación de código *LaTex* para la representación de *PTS*s o derivaciones en tres formatos diferentes:
  - Como un sólo árbol de derivación.
  - Como una secuencia de árboles de derivación.
  - Como una lista tabulada de juicios.
- Posibilidad de efectuar acciones de *deshacer* y *rehacer* sobre cualquiera de las derivaciones sobre las que estemos trabajando, de manera independiente entre sí.
- Manipulación simbólica de derivaciones.

Pero para poder comprender un poco mas nuestra implementación, primeramente daremos una idea general de los módulos que componen nuestro sistema, y también una noción de la función que cumple cada uno de estos módulos. Luego haremos algunas acotaciones referidas a algunos detalles partículares que hacen a la implementación, para concluir con una breve descripción de la interfaz gráfica que presenta nuestro prototipo. El lector no debería esperar en este capítulo una descripción detallada de cada una de las funciones que forman parte de nuestra herramienta. Para ello sugerimos consultar la documentación del código fuente que acompaña a nuestro sistema.

5.1 Módulos

A continuación daremos una idea más precisa de cada uno de los módulos que forman parte de nuestro sistema y la funcionalidad que aporta cada uno de ellos, además de la relación existente entre cada uno de ellos. Pero antes de ello, y para comprender un poco mejor como se encuentran organizados los componentes de nuestro sistema, procederemos a comentar como se encuentran ubicados dentro del árbol de directorios que forma parte de nuestra implementación. Mirando la carpeta principal, notaremos que existen tres directorios:

Assistant: Es, tal vez, el más importante de todos. En éste se ubican los archivos que implementan las estructuras y funcionalidades que formalizamos en el capítulo anterior. Es decir, aquí se centra toda la lógica de manipulación de derivaciones que emplea nuestro prototipo de herramienta. Veremos en más detalle estos módulos en la sección 5.1.1.

Interface: En esta carpeta se encuentran definidos los archivos que describen los componentes visuales que forman parte de la interfaz gráfica, pero que no se relacionan con el comportamiento propio de la interfaz.

PTS: Cuando avancemos un poco más en la descripción de nuestra implementación, veremos que una de las funcionalidades que ésta ofrece, es la de configurar el *PTS* con el cual se va a trabajar. En esta carpeta se encuentran definidos algunos archivos de configuración de sistemas de tipos, los cuales son provistos por defecto junto a la herramienta.

Además de estos directorios, existen varios archivos que no se encuentran ubicados en el directorio raíz. Dependiendo del prefijo que presenta cada uno de ellos, los podemos agrupar en dos conjuntos:

GUI: Éstos se encargan de definir el comportamiento y las acciones que debe llevar a cabo cada uno de los componentes que forman parte de la interfaz gráfica frente a peticiones del usuario o respuestas del sistema.

IO: En estos archivos se encuentran definidas las funciones que le permite al asistente comunicarse con el sistema de archivos. Como veremos mas adelante, ésto nos permitirá grabar y recuperar información referida a las derivaciones o especificaciones de *PTS*s.

Además de estos dos grandes conjuntos, existen algunos archivos más:

- Un par de archivos *.lang* que no hacen más que definir el resaltado de sintaxis para contextos y expresiones.

- Un archivo *EventAdministrator* que define un administrador de eventos basado en invocación implícita que usamos en la implementación de la interfaz gráfica de nuestra herramienta. Daremos un poco más de detalle junto a una explicación de su finalidad más adelante, en la sección 5.1.3.

Como ya dijimos, los módulos fundamentales para el correcto funcionamiento del sistema son, por un lado, los que se encuentran dentro de la carpeta *Assistant* y por otro los que definen el comportamiento de la interfaz (*GUI...* y *IO...*) junto al administrador de eventos. A menudo nos referiremos al primer conjunto de archivos (los del directorio *Assistant*) como el *núcleo del asistente*, mientras que cuando hablemos de *interfaz del asistente* nos estaremos refiriendo al segundo grupo (incluyendo el administrador de eventos).

5.1.1 Núcleo del asistente

Como ya dijimos, llamamos *núcleo del asistente* al conjunto de módulos que definen las funcionalidades básicas inherentes al comportamiento de nuestro asistente de demostración. Cada uno de los archivos que conforman el núcleo, se encargan de modelar alguna de las estructuras o funcionalidades que dimos en el capítulo 4.

En la figura 5.1 se pueden observar los módulos que forman parte del núcleo del asistente y la relación de dependencia que existe entre ellos. Como se puede apreciar, la estructura que adoptan los mismos es casi jerárquica, en la cual, los módulos inferiores definen estructuras simples y exportan algunas funcionalidades que permiten trabajar sobre las mismas. Haciendo uso de tales servicios, los módulos superiores pueden definir funciones y mecanismos más complejos, valiéndose de las abstracciones provistas por módulos inferiores.

Como dijimos anteriormente, cada uno de los módulos declarados se encarga de declarar alguno de los componentes que formalizamos en el capítulo anterior. Veamos entonces cual es la función específica de cada uno de ellos y el concepto que abstraen en cada caso.

Módulo: Misc

En él definimos funciones auxiliares que emplearemos en el resto del sistema. La mayoría de ellas se refieren a la manipulación de árboles finitarios, como la búsqueda de un elemento, la modificación de algún valor, mecanismos de eliminación de nodos (necesario para que la eliminación de premisas funcione correctamente), el mapeo de árboles introducido en la definición 4.53, y demás funciones similares.

Así mismo, usamos este módulo para definir otras funciones auxiliares de ordenación de elementos, eliminación de duplicados, y compraración de cadena de caracteres entre otras.

Módulo: PTSSpecification

En este módulo damos nuestra definición de *sort*, *axioma* y *reglas*, además de la estructura que emplearemos para almacenar una especificación de *PTS* introducida en la sección 4.1. También lo utilizamos para definir las funciones que trabajan sobre los *PTS*, tales como $\in_S$, $\in_A$ y $\in_R$, dadas en la definición 4.4 o la inclusión de especificaciones formalizada en la definición 4.5. Así mismo, empleamos este módulo para definir símbolos especiales referidos a la especificación de *PTSs* como el de $\bowtie$ introducido en la definición 4.1

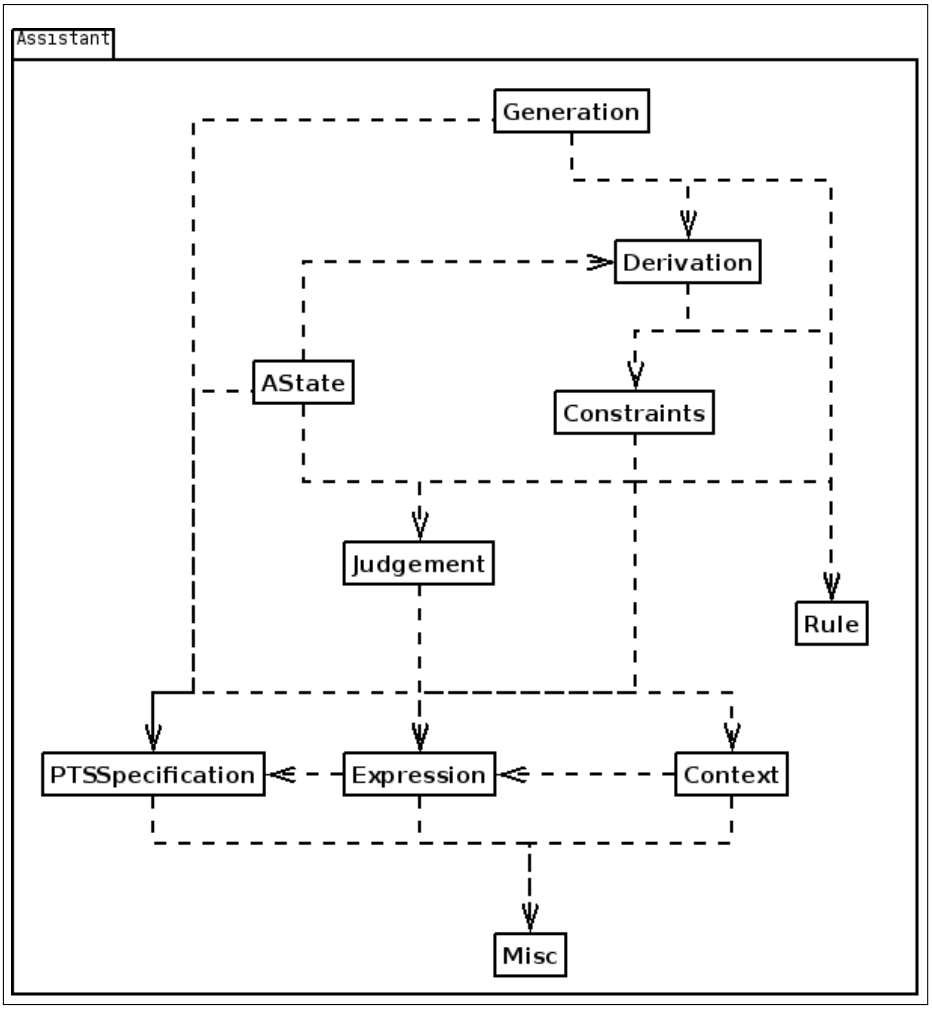

Figura 5.1: Dependencia de módulos en el núcleo del asistente

Módulo: Expression

Este es uno de los módulos mas importantes que forman parte de nuestro sistema. En él declaramos la estructura que emplearemos para denotar las expresiones que manejará nuestra herramienta, las cuales fueron dadas a lo largo de la sección 4.2. Además, definimos las funciones que trabajan directamente sobre expresiones, como lo son la de búsqueda de variables libres (4.8), la función que determina el conjunto de incógnitas que ocurren en una expresión (4.9), o la que establece la α o β equivalencia entre expresiones, entre otras que hemos dado.

Empleamos este módulo también para declarar una clase de *Haskell* [has90], la que nos permite extender la noción de sustitución, tanto de identificadores como de incógnitas, no sólo sobre el conjunto de expresiones, sino que además, instanciando tal clase adecuadamente, resulta bastante sencillo e intuitivo extender la sustitución a construcciones más complejas, como lo pueden ser los contextos, juicios, restricciones o árboles de derivaciones, entre otros.

Módulo: Context

Tal y como su nombre lo indica, en este módulo declaramos la estructura que representará a un contexto en nuestro sistema. Así mismo, definimos mecanismos que permiten añadir nuevos elementos, consultar qué pares de identificadores y expresiones componen un contexto dado o verificar cuándo un contexto se encuentra completa o correctamente definido.

Módulo: Judgement

En este módulo definimos la estructura que representará a un juicio, según lo establecido en la sección 4.3.1. En realidad este módulo es bastante sencillo, ya que no existen funciones complejas que trabajen a nivel de juicios, salvo tal vez, la encargada de determinar cuándo un juicio se encuentra bien formado o completamente definido.

Módulo: Rule

Si bien cuando formalizamos los componentes que forman nuestra herramienta, introdujimos la noción de *regla* tan sólo como un elemento secundario empleado en la declaración de los nodos que conforman un árbol finitario de juicios, en realidad el conjunto de funcionalidades que se puede asociar a una regla es mayor al que nosotros declaramos originalmente. Es por ello que optamos por emplear un módulo aparte para la declaración de los nombres de reglas.

Módulo: Constraints

Uno de los conceptos fundamentales que declaramos al momento de formalizar nuestra herramienta, fue el de las restricciones, al cual le dedicamos toda la sección 4.3.2. En este módulo no solo definiremos las estructuras que representarán a cada una de las restricciones que es capaz de manipular el sistema, sino que además las funciones y relaciones encargadas de verificar y reducir un conjunto de restricciones. En particular, damos una implementación de la relación $\overset{\text{check}}{\Longrightarrow}$ introducida en la definición 4.35 junto a un conjunto de funciones auxiliares que permiten imitar el comportamiento de la relación introducida entonces.

Es además en este mismo módulo que definimos las funciones encargadas de generar el conjunto de restricciones para ciertos juicios y premisas (ver definición 4.60).

Módulo: Derivation

Es sabido que una de las estructuras mas importantes con la que trabaja nuestra herramienta, es la derivación, definida en este módulo. Por ello, comenzamos por definir los conceptos de *paso de derivación* y *árbol de derivación*, finalizando con una implementación de lo que consideramos una derivación. Si recordamos nuestra definición de derivación (4.38), veremos que ésta no constituye más que un contenedor para un *árbol de derivación* y un conjunto de *restricciones*. Pero tan importantes como el concepto de derivación son las funciones que nos permitirán manipular los elementos que forman parte de una derivación, definidas también en este módulo.

Además, es aquí que definimos un mecanismo que resulta fundamental para garantizar el correcto funcionamiento de nuestra aplicación. Si recordamos las funciones generadoras de premisas que vimos en la sección 4.4.1, veremos que para algunas de ellas resultaba necesario declarar algunos nombres de variables o incógnitas *"nuevas"*,

que no fuesen utilizadas en la derivación sobre la cual estábamos trabajando. Un problema que surgía de ésto era que, si alguna expresión de nuestra derivación contenía un incógnita, entonces no podíamos, a priori, determinar cuales incógnitas o variables podrían pasar a ser utilizadas por la derivación al definirse (tal vez parcialmente) alguna de las incógnitas que ocurren en la derivación.

Para solucionar este inconveniente, en este mismo módulo declaramos un mecanismo de generación de variables e incógnitas nuevas que nos soluciona dicho problema. El mismo consiste en anexar a cada derivación, un conjunto (potencialmente infinito) de variables e incógnitas del cual se pueden ir retirando elementos que no hayan sido empleados previamente en la derivación. Podemos encontrar este mecanismo bajo el nombre de *DS*, en alusión a que en cierta forma, conserva parte del *Derivation State* que estamos manipulando.

Módulo: Generation

Recordando el capítulo anterior, veremos que las últimas secciones del mismo están dedicadas a los procesos de generación y eliminación de premisas para juicios que forman parte de un *árbol de derivación*. Claramente, los procedimientos destinados a tal tarea, operan sobre las derivaciones, de modo que la declaración de tales mecanismos podría efectuarse en el mismo módulo que define las derivaciones.

No obstante, en procura de conservar la simplicidad de los módulos definidos, consideramos que las funciones extras, si bien se encuentran vinculadas a las derivaciones, proveen un conjunto completamente nuevo de funcionalidades. Por este motivo, es que los mecanismos encargados de generar y eliminar premisas se encuentran declaradas en un módulo aparte.

Módulo: AState

Una particularidad propia de la implementación de nuestra herramienta radica en que, a diferencia de lo que vimos en el capítulo anterior, necesitaremos de una estructura que nos permita almacenar el *"estado"* de una derivación (de allí el nombre de este módulo, por *"Assistant State"*). Si bien a nivel del *núcleo del asistente* no existen declaraciones monádicas que nos permitan conservar el estado de nuestra aplicación, nada nos impide declarar cuales serán los valores y estructuras que deberíamos almacenar con la finalidad de conservar el estado de la derivación que estamos usando. De este modo, si en algún momento se decide construir una interfaz completamente nueva, para conservar el estado del asistente, solo deberíamos hacer persistente una de las estructuras *AState* provista por este módulo. Entonces, los componentes que forman parte de dicha estructura son:

- La especificación del *PTS* que estemos empleando.
- La derivación sobre la cual nos encontramos trabajando.
- El juicio que estamos tratando de manipular. Si bien dicho juicio también se encuentra declarado en la derivación que ya estamos almacenando, al poder acceder al mismo de manera directa, nos estamos asegurando una respuesta más pronta a las invocaciones que efectuemos sobre éste, como lo son las consultas o las llamadas a funciones generadoras de premisas, entre otras.
- La posición que ocupa el juicio que estamos manipulando dentro de todo el árbol que conforma la derivación. Puede que ahora no nos resulta evidente el por qué

requerimos de este campo, pero cuando veamos las implementación de la interfaz gráfica, notaremos que a menudo querremos efectuar ciertas operaciones sobre la representación gráfica del juicio sobre el cual estamos trabajando y para poder llevar a cabo ésto, necesitamos de un mecanismo que nos garantice que el juicio y su representación gráfica se corresponden. Además, si quisiéramos eliminar las premisas de dicho juicio, no nos alcanza con conocer la forma que éste tiene (ya que dos juicios idénticos podrían ocurrir en un mismo árbol de derivación), de modo que necesitamos conocer la ubicación precisa que éste ocupa dentro de dicho árbol.

Ahora que ya hemos dado una idea general de los módulos que forman parte del *núcleo del asistente*, estamos en condiciones de continuar con aquellos que tienen por objeto declarar el comportamiento que presentará cada uno de los componentes que forman parte de la interfaz gráfica de nuestra herramienta.

5.1.2 Interfaz del asistente

En la sección anterior nos dedicamos a dar una breve descripción de los módulos que conforman el bloque principal de la herramienta, encargada de hacer los cálculos, deducciones y manejo de derivaciones. Ahora veremos de manera similar las estructuras y funcionalidades provistas por los módulos encargados de declarar el comportamiento de los componentes que forman parte de la interfaz gráfica. Es cierto que hasta el momento no hemos dicho nada respecto a dicha interfaz, e incluso, no es uno de los temas sobre los cuales nos explayaremos en demasía. Pero también resulta cierto que conforma una de las partes fundamentales que hace al funcionamiento de nuestra herramienta, y por lo tanto, se merece al menos que efectuemos un breve reseña al respecto.

A modo de introducción, podemos observar los módulos que forman parte de la interfaz en la figura 5.2, junto a la relación de dependencia que existe entre los mismos. Como se puede apreciar, varios de los módulo que allí figuran, ya fueron dados en el momento en que introdujimos el *núcleo del asistente*. Su presencia en este nuevo diagrama no tiene más objeto que el de representar cómo los nuevos módulos de la interfaz se encuentran relacionados con los que conformaban el *núcleo del asistente*. Por si aún no ha resultado del todo evidente, la interfaz no hace mas que proveer de un medio de comunicación amigable entre el usuario y las herramientas que provee el núcleo mismo. Por este motivo, es que no debería extrañarnos la presencia de un módulo de abstracción gráfica por cada uno de los módulos que nos definía una nueva estructura en el *núcleo del asistente*. Veamos a continuación, una breve reseña de las funcionalidades que provee cada uno de estos nuevos módulos.

Módulo: IOPTS

Como dijimos anteriormente, los módulos cuyo nombre comienzan con el prefijo *IO* se refieren a la comunicación del asistente con el sistema de archivos. En particular, este módulo se encarga de grabar y obtener la información referida a la especificación de *PTS*s hacia y desde archivos, para lo cual, exporta un conjunto de funciones encargadas de escribir información referida a las especificaciones y un conjunto de funciones encargadas de interpretar tales datos. La sintaxis de los archivos generados y aceptados por éste módulo puede ser consultada en el apéndice B que acompaña a este mismo informe.

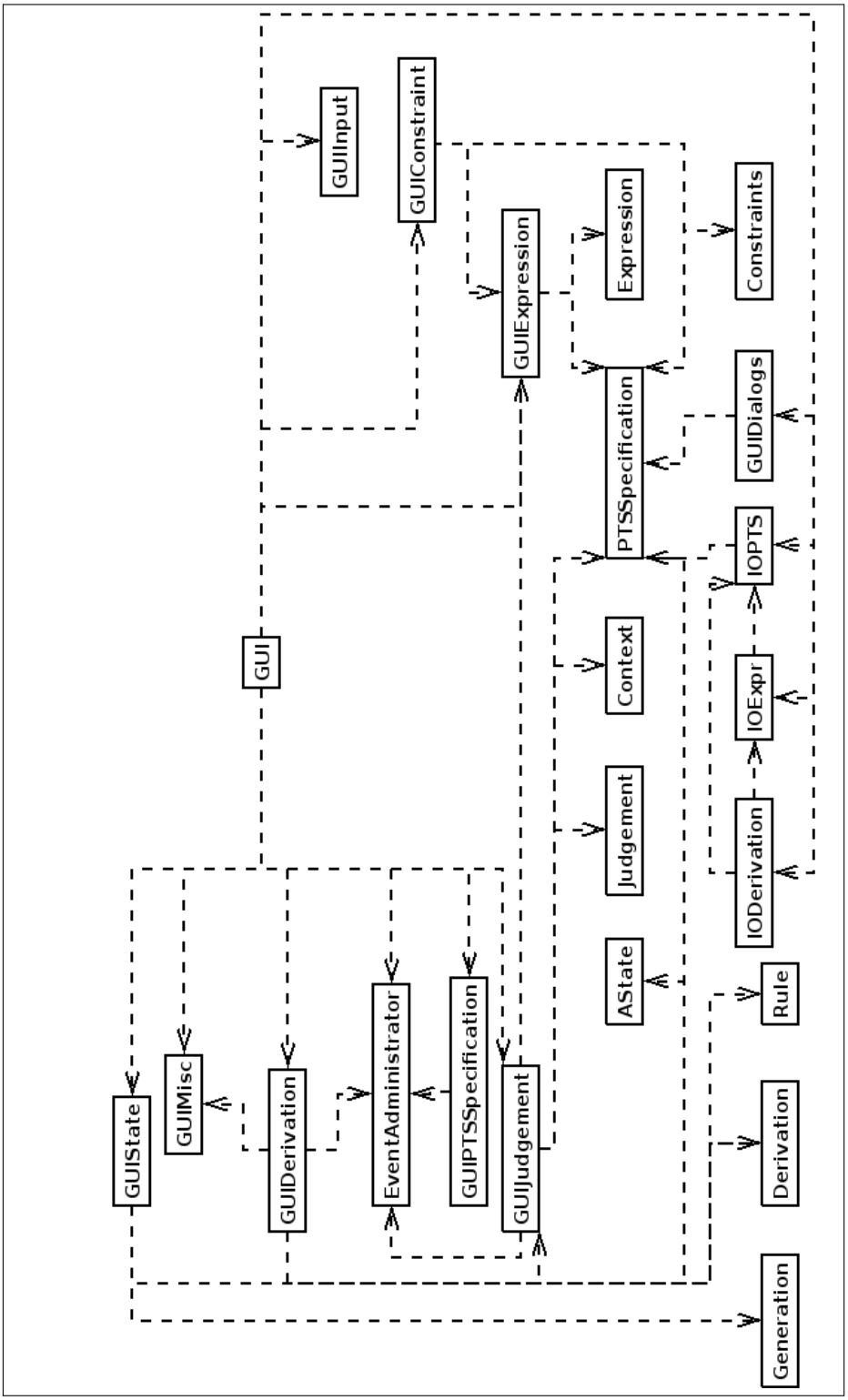

Figura 5.2: Dependencia de módulos con la interfaz del asistente

Módulo: IOExpr

Otro componente que a menudo estaremos interesados en almacenar serán las expresiones. En realidad este módulo no nos resulta de gran utilidad por sí sólo, ya que nunca estaremos interesados en almacenar el valor de una sola expresión. Sin embargo, las funciones que ofrece pueden ser empleadas por otros módulos, como el *IODerivation* que veremos a continuación y el *GUI*, que lo necesitará para interpretar las expresiones ingresadas por el usuario al momento de definir una nueva derivación.

Módulo: IODerivation

Uno de los mecanismos en los que sí estamos interesados, es en uno que nos permita grabar y obtener las derivaciones (tal vez incompletas) sobre las cuales estemos trabajando. Para lograrlo, es que contamos con este módulo, el cual valiéndose de *IOPTS* e *IOExpr*, nos permite llevar a cabo dicha tarea. Una descripción más detallada de la sintaxis aceptada por éste módulo en la declaración de derivaciones, puede ser consultada en el apéndice C.

Como ya dijimos anteriormente, una de las ventajas que ofrece nuestra herramienta es la de poder generar el código *LaTeX* que represente a una derivación. Las funciones que permiten tal conversión, se encuentran declarada también en los módulos de *IO*. El *IODerivation* exporta funciones que permiten la conversión de derivaciones, el *IOExpr* funciones para expresiones y el *IOPTS* para especificaciones de *PTS* respectivamente.

Módulo: GUIPTSSPecification

Este módulo se encarga de exportar los componentes que representarán gráficamente a una especificación de *PTS* junto a los *sorts* que forman parte de la misma. Además, declara el comportamiento de uno de los componentes gráficos que le da al sistema la habilidad de personalizar la representación gráfica de los elementos que forman parte de un *PTS*.

Módulo: GUIExpression

Uno de los componentes gráficos más utilizados por nuestra interfaz es el que declara como deben visualizarse las expresiones que manipula nuestro asistente. Además de proveer de un constructor de tales representaciones, el módulo *GUIExpression* exporta las funciones encargadas de la representación de sustituciones, tanto de identificadores como de incógnitas.

Módulo: GUIJudgement

Como ya debe resultar evidente a esta altura, la mayoría de los elementos que fueron declarados a nivel del asistente (*sorts*, expresiones, juicios, etc.) poseen su correspondiente componente de interfaz que los representa. En éste caso, el módulo *GUIJudgement* se encarga de la representación gráfica de juicios. Además de los constructores de representaciones de juicios, exporta funciones que manipulan tales representaciones, como el resaltado de juicios o la modificación de los mismos.

Módulo: GUIConstraint

Desde el momento en que nuestro sistema es capaz de manipular restricciones sobre expresiones, resulta evidente que necesitaremos de un módulo que nos provea de la representación gráfica de las mismas. A diferencia de lo ocurrido con los componentes gráficos introducidos hasta ahora, las restricciones cumplen sólo una función informativa a nivel de la interfaz, de modo que no se requieren funciones que sean capaces de manipularlas una vez que han sido creadas.

Módulo: GUIDerivation

Este módulo exporta la representación gráfica de una derivación, que a la larga, resulta ser el componente gráfico más utilizado por nuestra herramienta. Sin embargo, debido a que todo el trabajo de manipulación de derivaciones es llevado a cabo a nivel del asistente, sólo necesitamos de funciones que nos permitan construir tal representación y, a lo sumo, eliminar una parte de tal representación.

Módulo: GUIInput

En este caso, no nos encontramos frente a un módulo que modele algunos de los componentes declarados anteriormente, sino que se trata de uno relacionado estrictamente con la interfaz gráfica de nuestra herramienta. Una de las posibilidades que brinda nuestro prototipo, es poder declarar cuáles serán el contexto y expresión original a partir de la cual intentaremos construir una derivación. El módulo *GUIInput* es el encargado de proporcionar la interfaz que le permite al usuario ingresar dicha información.

Módulo: GUIMisc

Tal y como ocurría a nivel del *núcleo del asistente*, el conjunto de funciones auxiliares referidas al manejo de componentes gráficos se encuentran declaradas aquí. Entre ellas algunas referidas al manejo de contenedores y consulta mejorada de propiedades de elementos visuales.

Módulo: GUIDialogs

En un intento por facilitar la modificación de los mensajes generados por el sistema, todos éstos se encuentran agrupados en este módulo. También optamos por declarar aquí funciones generadoras de mensajes de información y de error, con la finalidad de unificar criterios, evitando al mismo tiempo duplicar código innecesariamente a lo largo de la implementación de nuestra herramienta.

Módulo: GUIState

Cuando dimos los módulos que forman parte del *núcleo del asistente*, introdujimos uno encargado de declarar una estructura capaz de almacenar el estado del asistente, en caso de ello fuera necesario. El módulo *GUIState*, casualmente, es el encargado de almacenar de manera monádica el estado de la interfaz. Pero debido a que nuestra herramienta permite trabajar de manera simultánea con varias derivaciones a la vez, no nos alcanza con almacenar una sola instancia de un *AState*. Esto nos motiva a que definamos nuestro estado de la interfaz como una tupla, conteniendo la siguiente información:

- El nombre de la derivación sobre la cual estamos trabajando, el cual nos permite distinguirlo del resto de las derivaciones que podamos tener cargadas en el sistema en ese mismo momento.
- Un mapeo de nombre de derivaciones a información de derivaciones, la cual incluye:
  - Un mapeo de *sorts* a los símbolos gráficos que los representan.
  - La representación gráfica de la derivación, lo que nos elimina el trabajo de tener que redibujarla cada vez que decidamos intercambiar la derivación sobre la cual estamos trabajando.
  - El estado del asistente para dicha derivación. Es decir, un elemento *AState*
- La información de la derivación sobre la cual se esta trabajando en ese momento. La misma incluye:
  - El árbol de derivación.
  - Un mapeo de nombres de reglas a posibles soluciones del proceso de generación de premisas para el juicio que se encuentre seleccionado en ese momento.
  - El nombre de la regla que se encuentra momentáneamente seleccionada para ser aplicada en el proceso de generación de premisas.
- Las dimensiones de la ventana principal del asistente, lo que le permite recalcular las posiciones de los juicios y derivaciones dentro de la ventana de trabajo, si esta es redimensionada durante una sesión.
- La información referida a la historia del proceso de derivación que permita deshacer o rehacer los cambios efectuados en cada una de las derivaciones sobre las cuales estemos trabajando. La información consiste en un mapeo de nombres de derivaciones a estructuras que almacenan datos sobre el estado de la derivación en momentos específicos, incluyendo el mapeo de *sorts* a símbolos, la representación gráfica de la derivación y el estado del asistente.

Evidentemente, además de la estructura que almacena el estado de la interfaz, declara varias funciones auxiliares que permiten consultar y modificar los valores almacenados en la misma, incluyendo acciones especificas como almacenar el estado actual, o deshacer/rehacer un cambio entre otros.

Módulo: GUI

Éste es el módulo principal de la interfaz gráfica. Sólo exporta una función, la que se corresponde con la función principal de manejo del asistente. Este módulo se encarga de cargar toda la interfaz y efectuar las llamadas que sean necesarias a los demás módulos que conforman el sistema para llevar a cabo las acciones que se requieran para que el asistente funcione correctamente. En líneas generales, las tareas que efectúa son entre otras:

- Cargar los componentes gráficos definidos para la interfaz declarados en la carpeta *Interface*.

- Asignar el comportamiento de cada uno de los componentes que forman parte de la interfaz, tales como botones, combos, listas, menues, etc.
- Cargar y descargar las ventanas que forman parte de la herramienta.
- Hacer las invocaciones necesarias al resto de los módulos.
- Chequear la consistencia de los datos manejados por los componentes de la aplicación.
- Cargar varios de los eventos que manejará el administrador, permitiendo así a otros componentes acceder a servicios básicos en la manipulación de la representación gráfica de las derivaciones.

No daremos mayores detalles de este módulo debido a que la complejidad del mismo es excesiva y escapa al objetivo principal de este trabajo, consistente en definir un prototipo de asistente de demostración.

5.1.3 El administrador de eventos

Cuando dimos una descripción inicial de los módulos que forman parte de nuestra implementación, dijimos que uno de ellos era el administrador de eventos. Si bien el mismo no forma parte de la interfaz del asistente, sí resulta necesario para el correcto funcionamiento de la misma. La necesidad de este módulo se hace presente por dos motivos:

1. Cada uno de los componentes que representan gráficamente a los elementos de una derivación son, al mismo tiempo, susceptibles a la interacción del usuario. Por ejemplo, los juicios pueden ser seleccionados.
2. Muy a menudo necesitaremos invocar a procedimientos de actualización de la interfaz y el administrador puede ser empleado también para reducir el número de parámetros que deben ser dados a las funciones actualizadoras.

En líneas generales, el administrador lo que nos permite es asignar código que retorne un valor de tipo *IO()* (la mónada de entrada/salida utilizada en *Haskell*) a un nombre de evento dado. De esa manera, cualquier componente que tenga acceso al administrador puede ejecutar dicho código, sin la necesidad de recibir argumentos adicionales. Los tipos exportados son:

- **EventName:** que denota el nombre de un evento.
- **Event:** que denota un evento.
- **EventID:** que denota una clave que identifica a un evento cargado en el administrador.
- **EventDataBase:** que denota el contenedor mismo de eventos, donde se almacenan los eventos que pueden ser invocados.

Siendo las funciones más importantes que exporta este módulo, las siguientes:

- **eventNew : String** $\times$ **IO()** $\rightarrow$ **Event**
  La cual crea un nuevo evento a partir de un nombre y una función de *IO()*. Nótese que esto no implica que estemos añadiendo el evento al administrador.

- **eventInsert : EventDataBase × Event → IO(EventId)**
  El cual nos permite añadir un nuevo evento al contenedor de eventos, para que pueda ser luego invocado por quien lo requiera. El valor de *eventId* retornado, resulta de utilidad a la hora de remover el evento del contenedor.

- **eventExec : EventDataBase × EventName → IO()**
  Esta es la función que nos permite ejecutar las acciones asociadas a algún evento en particular. El valor de *IO()* retornado se corresponde con el resultado de la ejecución del código asociado a dicho evento.

- **eventRemove : EventDataBase × EventId → IO()**
  Con esta última función podemos remover algún evento del contenedor. Para ello, necesitaremos conocer del identificador generado al momento de cargar el evento en el sistema.

Como ya dijimos, la necesidad de un administrador de eventos surge por dos motivos fundamentales. El primero de ellos era que los componentes que representan elementos de la derivación, pueden recibir también eventos del entorno. De ese modo, por ejemplo, al hacer click sobre un juicio, nos gustaría registrar en el sistema que dicho juicio ha sido seleccionado por el usuario o denotar de cierta manera que el mismo ha sido seleccionado. Ahora bien, para poder resaltar el juicio seleccionado, necesitamos redibujar la representación de la derivación sobre la que estemos trabajando. Pero para poder llevar a cabo esta acción, necesitaríamos que *GUIJudgement* invoque alguna función del módulo *GUIDerivation* o *GUI*, lo que nos conduciría a una inclusión cíclica de módulos.

Haciendo uso del administrador de eventos, el módulo *GUI* puede almacenar un evento con las instrucciones como redibujar la derivación y pasar el contenedor de eventos al módulo *GUIJudgement*. De éste modo, cuando este último necesite redibujar la derivación, sólo debe ejecutar dicho evento, sin la necesidad de conocer la identidad de los otros módulos.

Otra situación en la que dijimos que el administrador de eventos nos simplificaba nuestros diseño radicaba en el número de parámetros que debe recibir algunas funciones. A menudo, algunas funciones de módulos secundarios necesitarían recibir varios elementos como argumento, simplemente para modificar algunos valores de manera estándar. Como dichas modificaciones resultarían idénticas, pasando sólo una referencia al contenedor de eventos, logramos:

- Reducir el número de parámetros.

- Evitar tener código duplicado en varios módulos, facilitando la modificación de las funcionalidades asociadas a dicho código, en caso de ser necesario.

5.2 Descripción de la interfaz

Concluiremos nuestra descripción de la implementación realizada, mostrando algunas de las ventanas que forman parte de la interfaz de nuestra aplicación. Para cada caso, enunciaremos cuales son los elementos principales que las componen y daremos una breve descripción de la función que cumple cada uno de ellos.

La ventana principal de nuestra herramienta es la dada en la figura 5.3. A través de la misma el usuario puede crear, guardar o abrir derivaciones, acceder a las ventanas de

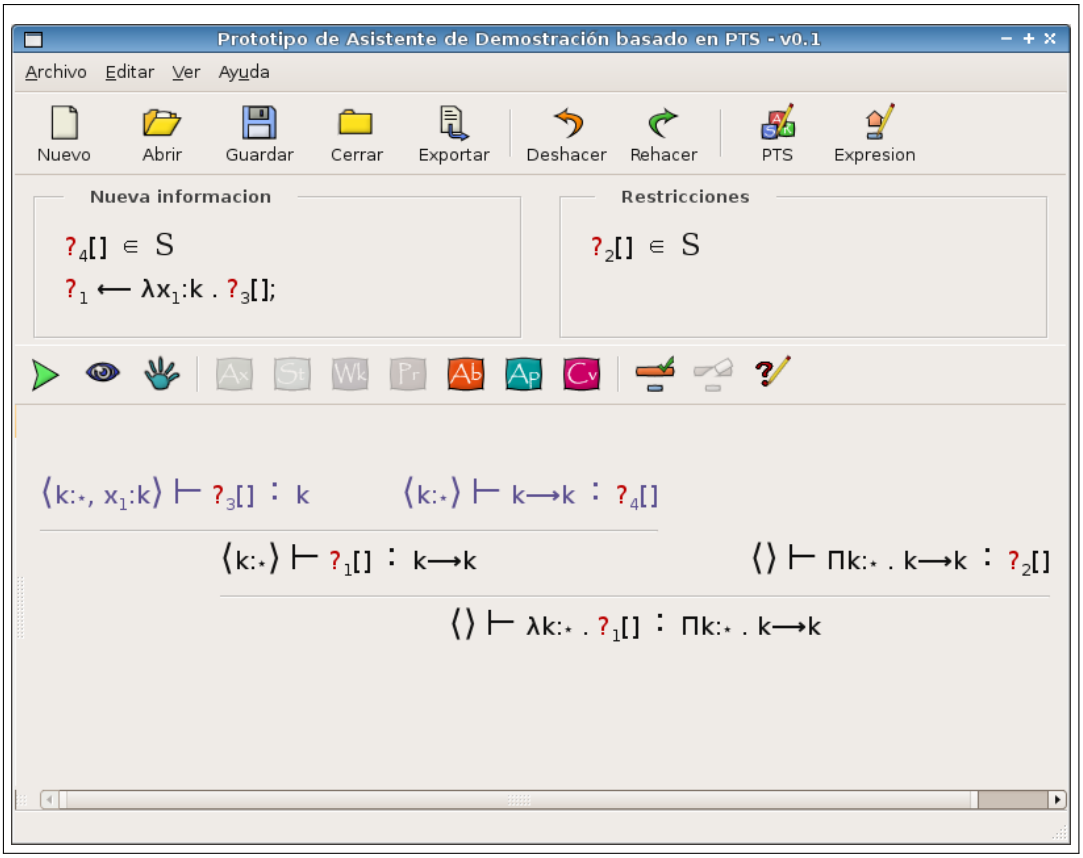

Figura 5.3: Vista de la pantalla principal de la aplicación

configuración de *PTS*s, exportar los datos generados, y acceder a varios mecanismos que le permiten manipular la derivación sobre la cual se esta trabajando.

Como se puede apreciar, la ventana principal cuenta con dos barras de herramientas. A través de la barra superior, el usuario puede, de izquierda a derecha, acceder a los siguientes comandos:

- Crear una nueva derivación.
- Cargar una derivación a partir de un archivo.
- Guardar la derivación actual en un archivo.
- Cerrar la derivación sobre la cual se esta trabajando.
- Exportar la derivación actual en formato *LaTex*.
- Deshacer los cambios efectuados en la derivación actual.
- Rehacer los últimos cambios en la derivación actual.
- Acceder a la ventana de configuración de *PTS*. La misma se muestra en la figura 5.4

- Acceder a la ventana de carga de contexto y expresión original a partir de la cual se efectuará la derivación. La misma puede ser vista en la figura 5.5

En tanto que la segunda barra de herramientas, permite al usuario:

- Mostrar u ocultar la lista de derivaciones que se encuentran cargadas en el sistema.
- Mostrar u ocultar las reglas empleadas en cada uno de los pasos que forman parte de la derivación actual.
- Activar o desactivar el modo semi-automático de generación de premisas.
- Emplear algunas de las reglas disponibles en el proceso de generación de premisas. Éstas son: *axiom*, *start*, *weakening*, *product*, *abstraction*, *application* y *conversion*.
- Aplicar la regla seleccionada en el proceso de generación de premisas.
- Eliminar las premisas del juicio seleccionado.
- Asignar una expresión a una incógnita.

Además de las barras de herramientas, esta ventana cuenta con tres áreas de trabajo, una de ellas activa. En el recuadro superior derecho se muestran las restricciones que aun restan por ser satisfechas en el proceso de derivación. A su izquierda se ubica un recuadro similar, en donde se muestras las restricciones que van a ser añadidas en caso de optar por aplicar la regla que se haya seleccionado, como así también las asignaciones a incógnitas que se hayan podido deducir en el mismo proceso de generación. Finalmente, el área de trabajo más importante es la inferior. En ella se muestra el estado actual de la derivación y es en ésta en que el usuario puede seleccionar con que juicio desea trabajar.

La configuración de la especificación del *PTS* que se empleará en la derivación puede llevarse a cabo a través de la ventana mostrada en la figura 5.4. En la parte superior, la misma cuenta con una barra de herramientas que permite:

- Cargar la especificación de un *PTS* a partir de un archivo que cumpla la sintaxis dada en el apéndice B.
- Guardar la configuración del *PTS* actual en un archivo, para que pueda ser empleado posteriormente en alguna otra derivación.
- Eliminar alguno de los *sorts*, *axiomas* o *reglas* que forman parte de la especificación actual.
- Definir un nuevo *sort* en la especificación actual.
- Definir un nuevo *axioma* en la especificación actual.
- Definir una nueva *regla* en la especificación actual.
- Aplicar los cambios efectuados en la especificación actual y cerrar la ventana.

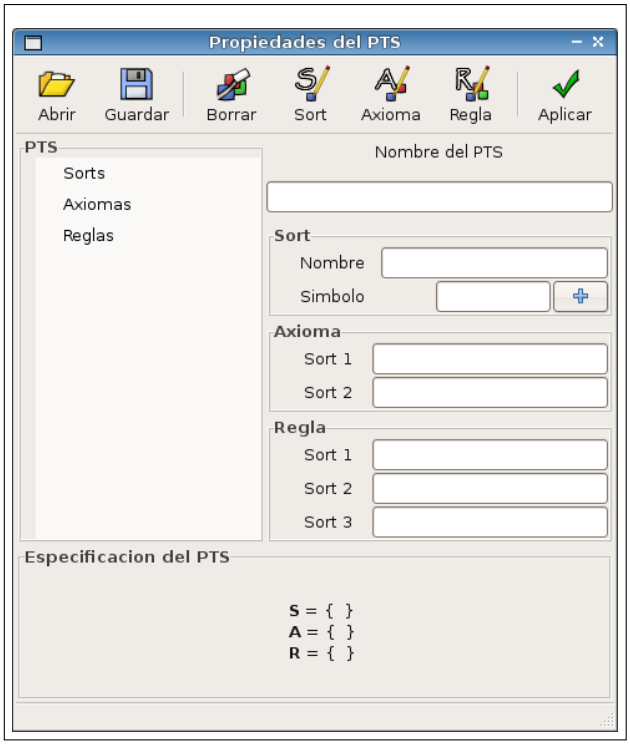

Figura 5.4: Vista de la ventana de configuración de *PTS*

Debajo de la barra de herramientas, sobre la izquierda, nos encontramos con la lista de *sorts*, *axiomas* y *reglas* que forman parte de la especificación actual. Sobre la derecha, nos encontramos con los recuadros que permiten cargar el nombre de la especificación y los elementos que la componen. Un caso particular es el recuadro destinado a la declaración de *sorts*, la cual, además de un nombre, nos brinda la posibilidad de definirle un símbolo particular que lo identifique. Algunos símbolos estándar pueden ser cargador empleando el botón contiguo a la casilla destinada a tal fin.

Finalmente, en la parte inferior de la ventana, se encuentra un recuadro en el cual se detalla de manera resumida, la información referida a la especificación actual.

Otra de las ventanas de uso frecuente, es la destinada a la carga del contexto y la expresión a partir de la cual se iniciará el proceso de generación de la derivación. La misma es bastante sencilla y puede ser apreciada en la figura 5.5. En la barra superior, consta con sólo dos botones. Uno para aplicar los cambios efectuados y otro para salir descartando los cambios.

Un poco más abajo aparecen dos recuadro, uno para describir el contexto y otro la expresión que denota el tipo del cual se partirá. La sintaxis aceptada por ambos recuadros se corresponde con la sintaxis dada para contextos y expresiones respectivamente en el apéndice C. Además, ambos recuadros brindan la posibilidad de deshacer o rehacer los cambios efectuados de manera independiente el uno del otro.

Finalmente, la última ventana que resulta relevante es la de asignación de valores a incógnitas, presentada en la figura 5.6. La misma, además de las opciones habituales de edición, presenta dos recuadros. El superior se emplea para denotar el identificador numérico que identifica a la incógnita que será sustituida, en tanto que en el recuadro

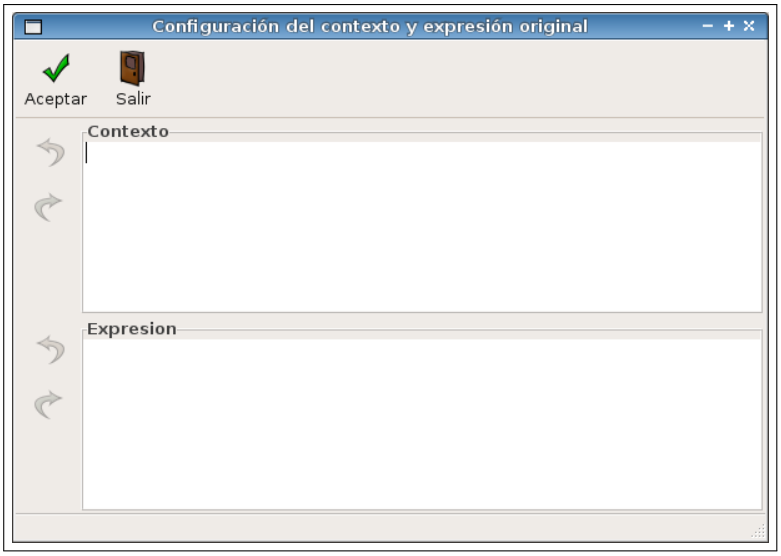

Figura 5.5: Vista de la ventana de carga inicial de derivaciones

inferior, se debe especificar la expresión por la cual se desea reemplazar dicha incógnita.

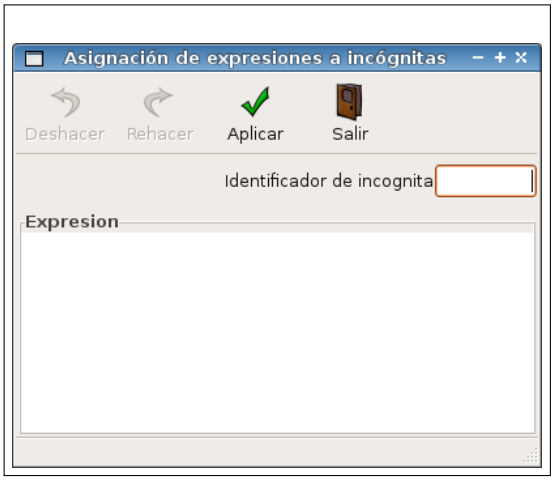

Figura 5.6: Vista de la ventana de asignaciones de incógnitas

De esta manera, con lo expuesto sobre los módulos y la interfaz, ya podemos dar por concluida nuestra descripción de la implementación realizada. Como se puede apreciar, no consideramos necesario dar mayores detalles sobre la misma debido a que con la información brindada por la documentación que acompaña al código junto a los apéndices que acompañan a éste informe, la información extra que podríamos brindar se tornaría redundante.

Capítulo 6

Conclusiones

Iniciamos el presente trabajo proponiendonos como objetivo implementar un prototipo de asistente de demostración basado en algún sistema de tipos simple, como lo puede ser el $\lambda\!\rightarrow$ o el $\lambda 2$. Sin embargo, al observar que el campo de aplicación de dicha herramienta se vería ampliado enormemente al emplear sistemas de tipos más complejos, se optó por desarrollar una herramienta capaz de ser parametrizada a través de *Sistemas de Tipos Puros*, una generalización de los sistemas de tipos convencionales presentes en el cálculo lambda.

Dado que primeramente debimos comprender los *PTS*, lo que se hizo fue tomar como referencia algunos de los sistemas de tipos más simples mencionados anteriormente y analizamos las relaciones existentes entre éstos y algunos sistemas de tipos más complejos. De ese modo, fuimos capaces de ir incrementando gradualmente el número de expresiones que eramos capaces de describir empleando cada vez sistemas de tipos más complejos. Para reducir la brecha entre los diferentes sistemas, optamos por elegirlos de manera tal que cada nuevo sistema que introducíamos incluía las características de sus predecesores y añadía a su vez algunas nuevas propiedades. Siguiendo esa metodología, valiendonos del conocimiento adquirido al analizar sistemas tales como el $\lambda\omega$ o el cálculo de construcciones, fuimos capaces de introducir los *Sistemas de Tipos Puros* junto a algunas de las propiedades más importantes que los caracterizan.

Luego, introdujimos la noción propuesta por el *Isomorfismo de Curry-Howard*, junto a una idea de como éste se relaciona con el cálculo lambda, permitiendonos utilizar expresiones de dicho cálculo para modelar pruebas formales propias de la *Lógica Intuicionista*.

Habiendo superado la etapa de estudio de la teoría ya existente, nos abocamos a dar una descripción formal para un prototipo de asistente de demostración que, valiéndose de los principios establecidos por la correspondencia de *Curry-Howard*, permite construir pruebas constructivistas de programas. Además, viendo el potencial que representan los *Sistemas de Tipos Puros*, le brindamos a nuestra herramienta la posibilidad de trabajar sobre cualquier especificación de *PTS*.

Al momento de formalizar los elementos que formarían parte de nuestra aplicación, uno de los primeros pasos consistió en definir expresiones capaces de contener subexpresiones no definidas y aún así conservar algunas propiedades fundamentales. Fue así que introdujimos el concepto de *incógnitas*, lo que demandó la tarea de extender el dominio de varias definiciones, quedando incluso algunas no del todo completamente

definidas, tal y como ocurrió en el caso de las variables libres, por ejemplo. Una vez definido nuestro conjunto de expresiones, debimos extender los resultados obtenidos a estructuras más complejas, como lo son los contextos, juicios y derivaciones.

Al mismo tiempo y dada la naturaleza del problema con el que estábamos tratando, surgió la necesidad de introducir el concepto de restricciones. Al definirlos, debimos declarar algunos mecanismos que le permitieran a nuestra herramienta manipular y verificar la veracidad o no de un conjunto de restricciones. Y no sólo ello, sino que además nos vimos en la obligación de definir procedimientos dedicados a la generación de premisas y restricciones, lo que le otorgó a nuestra herramienta un alto grado de automatización en el proceso que conlleva la construcción de una derivación.

Además de mecanismos de construcción, declaramos un conjunto de funciones destinadas a lidiar con la tarea de eliminar premisas para un juicio dado y cuando lo hicimos, descubrimos que dicha tarea requería de un análisis más detallado de la situación, de modo que resultó necesario declarar mecanismos de re-generación de restricciones y derivaciones.

Finalmente, empleando los conocimientos adquiridos en las etapas anteriores, fuimos capaces de dar una implementación de la herramienta propuesta, proveyendo a la misma de una interfaz gráfica que facilita, simplifica y hace más amena su utilización.

Creemos haber logrado desarrollar un prototipo de asistente con cierto grado de robustez, capaz de brindar las funcionalidades básicas esperables en una herramienta de sus características. Sin duda, los servicios provistos por éste resultas bastante reducidos en comparación con otros asistentes de demostración basados en principios similares a los seguidos en el desarrollo de nuestra aplicación, pero de todas formas nos mostramos conformes con el grado de desarrollo adquirido.

En el afán de mejorar las prestaciones ofrecidas por nuestro prototipo, creemos que podrían efectuarse trabajos posteriores tendientes a mejorar aspectos tales como:

- El proceso de automatización que conlleva a la construcción de derivaciones. Resultaría óptimo definir un mecanismo de generación automática de premisas y restricciones que permita la construcción automática de múltiples pasos de derivación. Además, se debería considerar la posibilidad de definir un mecanismo que permita emplear trozos de pruebas en esa u otras pruebas que presenten características similares.

- El orden en que se debe configurar la derivación y la especificación del *PTS* deberían poder ser alterados o brindar al menos la posibilidad de agrupar pruebas similares bajo una misma especificación, si ésto fuese necesario.

- El conjunto de reglas a partir de la cual es posible generar premisas se encuentra fijo en esta primera versión. Una posibilidad sería de la declarar un mecanismo o incluso estructuras más complejas que permitieran la declaración de reglas como parte del proceso de demostración de un teorema. De esta manera obtendríamos una herramienta mucho más flexible, no acotada a un conjunto preestablecido de reglas y aplicable a un rango mayor de situaciones.

- Construir una biblioteca estándar de especificaciones de *PTS* y conjuntos de funciones comúnmente empleadas en demostraciones formales, de manera de simplificar el proceso de configuración y declaración de derivaciones.

- Extender la declaración de las expresiones permitidas por la aplicación, de manera que se puedan emplear construcciones más complejas (como *case* o análisis por casos), algunas estructuras de datos y constantes en el proceso de derivación.

- Permitir la edición manual de las derivaciones a través de alguna ventana de edición o mecanismo similar. Actualmente, las derivaciones pueden ser modificadas en forma directa alterando el archivo de derivaciones en la cual se encuentran almacenadas, pero un editor integrado facilitaría las tareas de control y verificación de los cambios efectuados.
- Debido a que el chequeo de β-igualdad puede tornarse indecidible en algunas situaciones, la verificación de dicha condición podría ser reemplazada por la búsqueda de β contracciones en una u otra dirección.
- Incrementar el poder de deducción con que cuenta la herramienta, para casos en los que algunas incógnitas puedan ser asignadas a valores de únicos, tal vez sin la necesidad de llegarse a una expresión completa que reemplace a dicha incógnita.
- Aumentar el conjunto de restricciones que el sistema sea capaz de representar, permitiéndose incluso operaciones lógicas entre las mismas, las cuales pueden incluir la disyunción o implicación, entre otras.

Apéndice A

Ejemplo

Durante el desarrollo de este trabajo, surgió la inquietud sobre cuán útil podría resultar dar un ejemplo concreto de la utilización de nuestra herramienta. Una posibilidad era la de transcribir alguna derivación *"interesante"* que fuera obtenida a través del uso de nuestra aplicación. Claro que en tal caso, estaríamos dejando de lado los detalles referidos a los pasos que condujeron a la obtención de dicha derivación. Incluso, ya varias de las derivaciones presentes en este informe, fueron generadas utilizando el mecanismo de exportación de código *LaTex* que provee nuestro prototipo, de modo que decidimos descartar esta opción.

Por ello es que, en lugar de simplemente escribir alguna derivación que presenta algún grado de complejidad, optamos por mostrar los pasos que conllevan a la obtención de una derivación simple, mostrando el estado de nuestro prototipo en cada uno de los pasos que implica la obtención de la derivación buscada.

Supongamos que queremos derivar la identidad polimórfica. En la sección 2.1.3 vimos que bajo la especificación de *PTS* para el sistema $\lambda 2$, el tipo de la misma se encuentra dado por:

$$\Pi\alpha : \star . \alpha \to \alpha$$

Veamos entonces como hallar un término (prueba) para dicho tipo (teorema). Lo primero que haremos será abrir el asistente, topándonos con la ventana principal de la aplicación, semejando a la mostrada en la figura A.1.

Para comenzar con nuestra derivación, primero deberemos asignar un nombre a la misma. Para ello, hacemos clic en el primer botón de la barra de herramientas superior, y accedemos así a la ventana de creación de derivaciones. En el recuadro emergente se nos pedirá que demos un nombre a la derivación que estamos a punto de crear. En nuestro caso particular, la llamaremos *"idPol_1"*. La situación planteada en este paso puede ser observado en la figura A.2.

Hacemos clic en *crear* obteniendo de esa manera, una nueva derivación creada, pero que aún no nos es mostrada en la ventana principal. Para poder acceder a la lista de derivaciones que se encuentran abiertas en ese momento, podemos hacer clic en el primer icono de la barra de herramientas secundaria, desplegando así una lista en la cual encontraremos la derivación que acabamos de crear. El siguiente paso consiste en asignar la especificación de *PTS* que emplearemos para llevar a cabo nuestra derivación. Para ello, seleccionando la derivación *"idPol_1"* de la lista de derivaciones, veremos

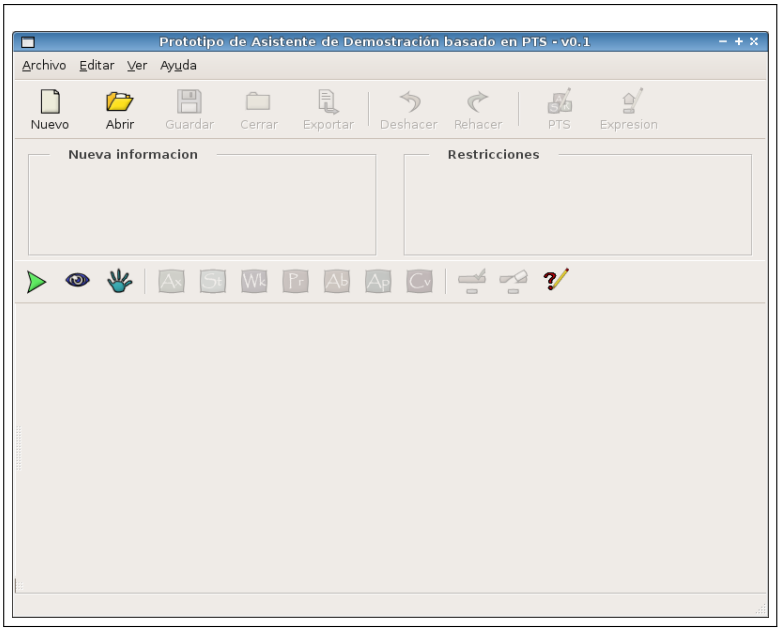

Figura A.1: Vista de la ventana principal de la aplicación

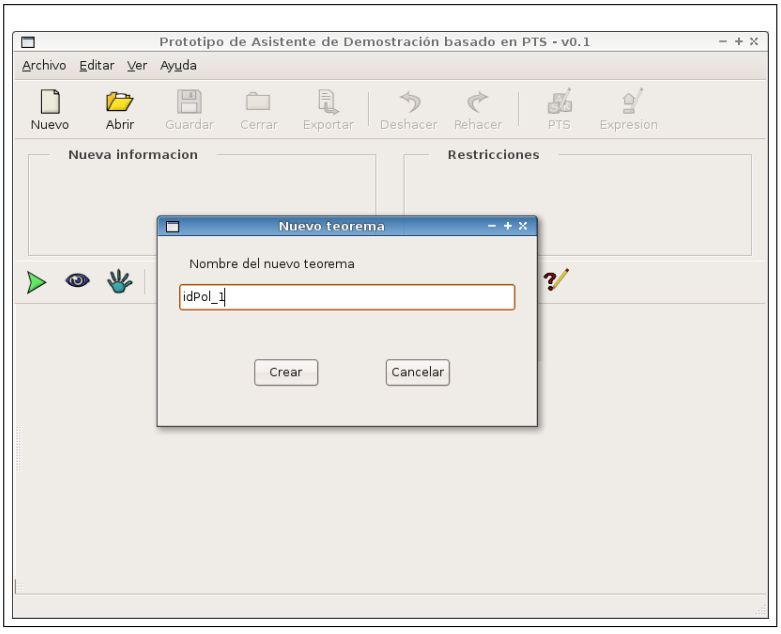

Figura A.2: Asignando el nombre a la nueva derivación

que el icono de acceso a la ventana de propiedades de *PTS* se nos habilita. Haciendo clic en el mismo, accedemos entonces a la ventana de configuración de *PTS*, la cual luce como se muestra en la figura A.3.

Supongamos por un instante que estamos queriendo hallar una prueba de la iden-

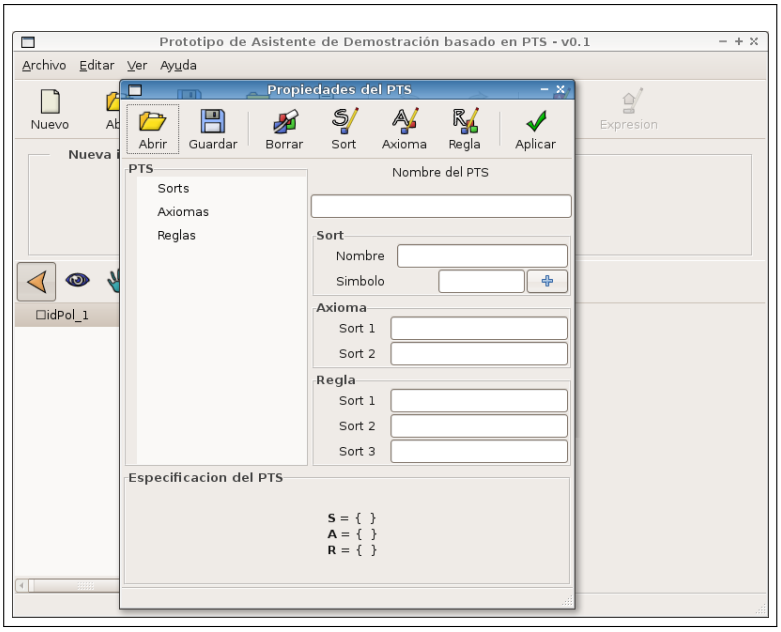

Figura A.3: Accediendo a la configuración del *PTS*

tidad polimórfica en un sistema en el cual no resulta posible lograr tal demostración. Digamos entonces que vamos a trabajar bajo el sistema $\lambda\!\rightarrow$. Para cargar la especificación del mismo, no tenemos más que hacer clic en *Abrir* y luego buscar en la carpeta *PTS* el archivo de configuración para dicho sistema. Al abrirlo, si la carga de datos resultó satisfactoria, la ventana de configuración debería haberse completado con la información referida al sistema $\lambda\!\rightarrow$, tal y como se muestra en la figura A.4.

Hacemos clic en *Aplicar* y retornamos a la ventana principal. Si prestamos atención, notaremos que ahora el último icono de la barra de herramientas principal se encuentra activado. Esto nos esta indicando que podemos cargar el contexto y la expresión a partir de la cual iniciaremos el proceso de derivación.

Para nuestro caso particular, dejaremos el recuadro de *Contexto* en blanco, y completaremos el recuadro de *expresion* con la expresión que representa el tipo $\Pi\alpha : \star . \alpha \rightarrow \alpha$ de acuerdo a la sintaxis dada en el apéndice C. La expresión ingresada debería parecerse a la mostrada en la figura A.5. Nótese que a medida que editamos el recuadro de *expresion* se nos habilitan las opciones de deshacer y rehacer para dicho recuadro.

Una vez que hemos finalizado, hacemos clic en *Aceptar*, para así retornar a la ventana principal, la cual ya se ha actualizado, de manera que ahora en el área de trabajo se muestra el primer juicio que formará parte de la derivación que construiremos, el cual contiene la información que acabamos de ingresar.

El lector atento notará que las ocurrencias de la variable *alpha* han sido sustituidas por la letra griega α. En realidad esto se debe a un mecanismo especifico de nuestra herramienta que se encarga de mapear las variables con nombres de letras griegas a sus respectivas representaciones gráficas.

A continuación, si seleccionamos el único juicio que se encuentra disponible, veremos que se habilitan tres opciones para la aplicación de una regla de derivación. Éstas son *abstraction*, *application* y *conversion*. Supongamos que seleccionamos la regla de *abstraction*. En ese momento, en el área principal de trabajo, se grafican las dos

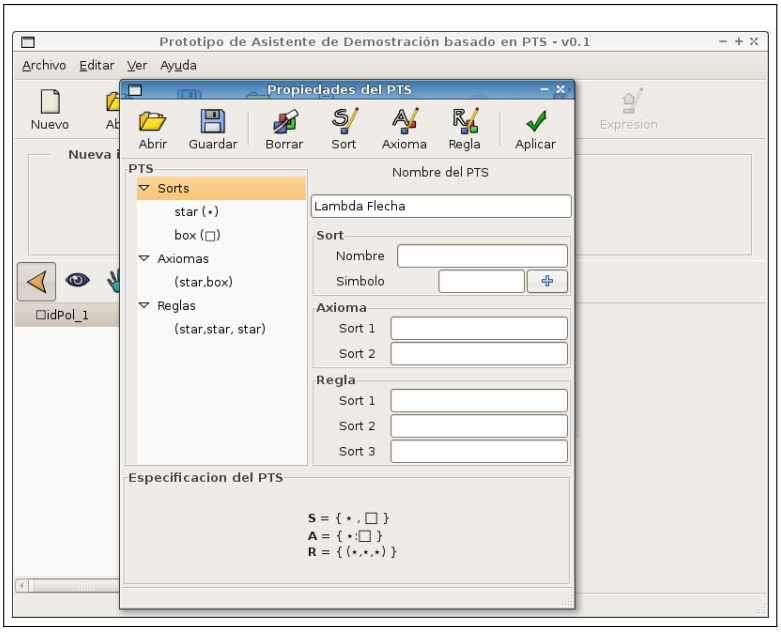

Figura A.4: Cargando la especificación del sistema $\lambda\rightarrow$

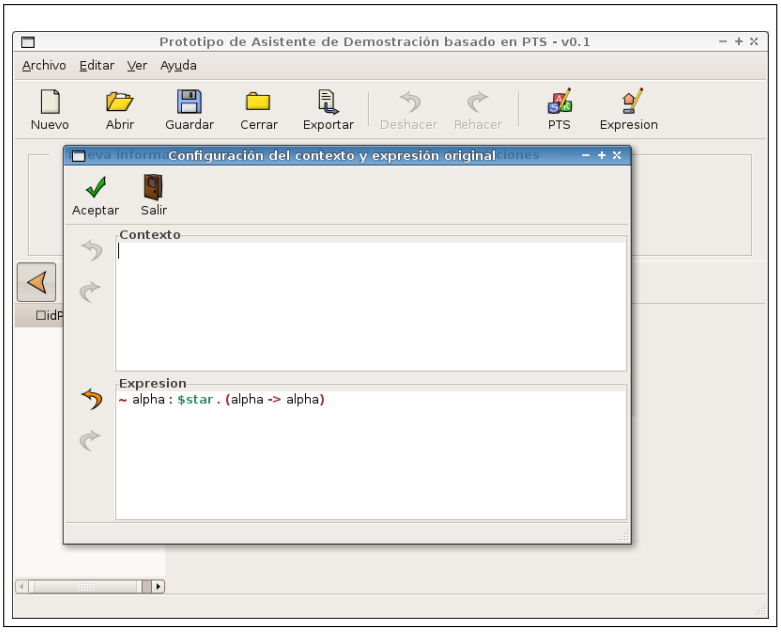

Figura A.5: Cargando la expresión original de la derivación

premisas que deberían ser satisfechas en caso de querer efectivamente emplear la regla de *abstraction*, tal y como se puede apreciar en la figura A.7.

Notemos que en la parte superior izquierda se detalla el conjunto de restricciones y asignaciones que se efectuarán en caso de que el usuario decida efectivamente aplicar

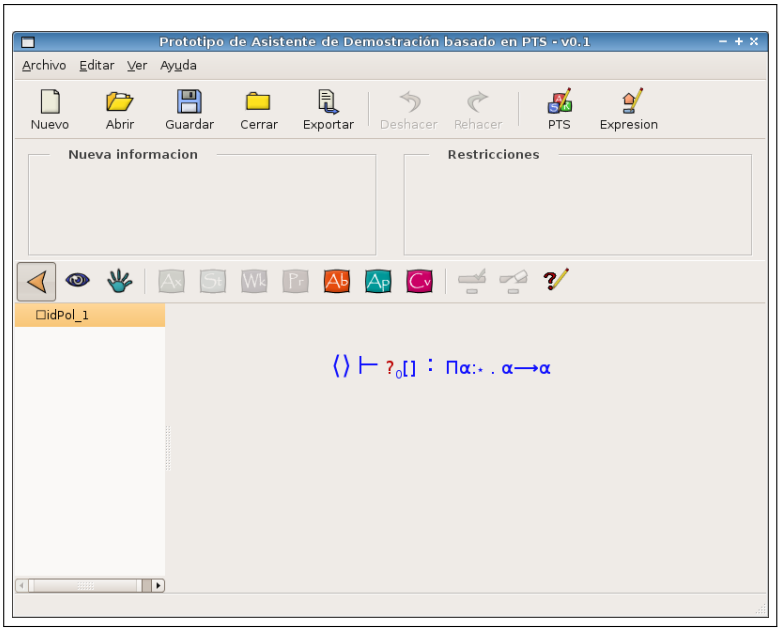

Figura A.6: Estado inicial de la derivación

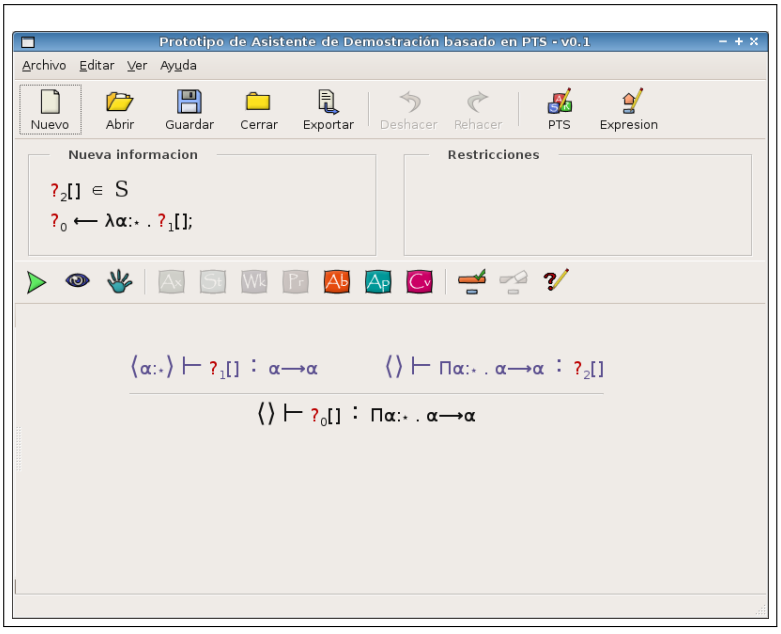

Figura A.7: Seleccionando la regla a ser aplicada

la regla de *abstraction*. En nuestro caso particular, se nos esta informando de que $?_2\,[]$ debería ser un *sort* y que además la incógnita $?_0\,[]$ será reemplazada por la expresión $\lambda\,\alpha : \star\,.\,?_1\,[]$. Si finalmente optamos por confirmar la aplicación de la regla haciendo clic en el antepenúltimo botón de la barra de herramientas secundaria, veremos cómo

el área de trabajo de nuestra aplicación se actualiza, luciendo ahora de manera similar a la imagen expuesta en la figura A.8.

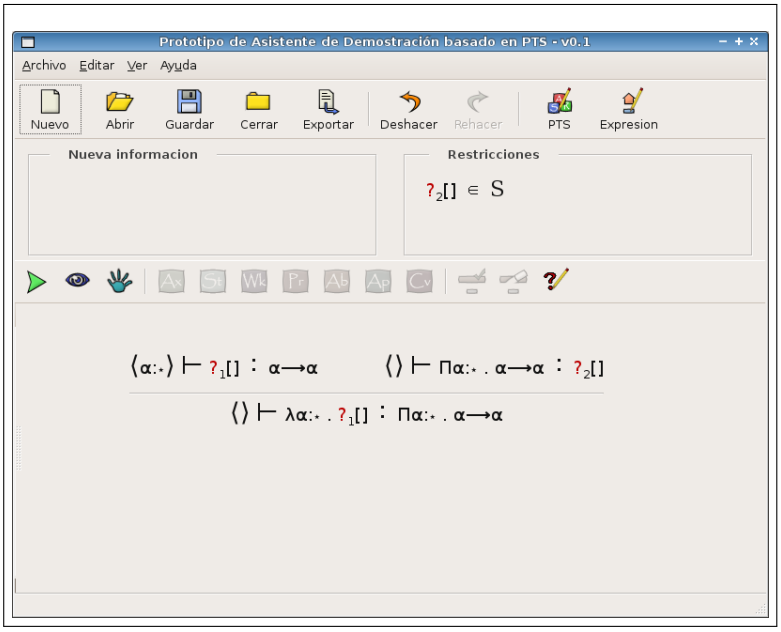

Figura A.8: Aplicando la regla seleccionada

En esta nueva captura, podemos observar que la restricción impuesta sobre la incógnita $?_2\,[]$ ha pasado a formar parte del conjunto de restricciones globales de nuestra derivación. Siguiendo el mismo procedimiento, aplicamos la regla de *product* sobre la segunda premisa que hemos generado. Pero entonces, se nos presenta una situación en la que deberíamos ser capaces de demostrar que $(\star,\star)$ forma parte del conjunto de axiomas del *PTS* que estamos empleando. Pero como estamos haciendo uso del sistema $\lambda\!\rightarrow$, dicho axioma no se encuentra disponible y por lo tanto no podemos proseguir con nuestra derivación.

Todo esto surge del hecho de que, al aplicar la regla de *product*, nuestra herramienta fue capaz de determinar que la única regla de la especificación que se podía emplear era la de la forma $(\star,\star,\star)$ y de allí se pudo deducir que $\star$ debería entonces tener tipo $\star$. Para poder solucionar este inconveniente, deberemos proceder a cargar la especificación del sistema $\lambda 2$, paro lo cual deberemos acceder nuevamente a la ventana de propiedades de *PTS* y cargar allí el archivo que contiene la configuración de dicho sistema. Llevando a cabo esta tarea, deberíamos llegar a una situación similar a la planteada en la figura A.9.

Luego podemos aplicar los cambios y regresar a la ventana principal de la aplicación. Un hecho que vale la pena destacar es que, debido a que la especificación del sistema $\lambda\!\rightarrow$ se encuentra contenida (de acuerdo a la definición 4.16) en la del sistema $\lambda 2$, todo el trabajo que hemos efectuado aún resulta válido bajo la nueva especificación de *PTS*. No obstante, al igual que nos ocurría anteriormente, bajo este nuevo sistema tampoco somos capaces de demostrar que $[] \vdash \star : \star$, en principio porque el axioma $(\star,\star)$ no se encuentra declarado en nuestra nueva especificación.

Nuevamente, el problema surge a partir de la información añadida a nuestra derivación en el momento en que decidimos aplicar la regla de *product*, de modo que los

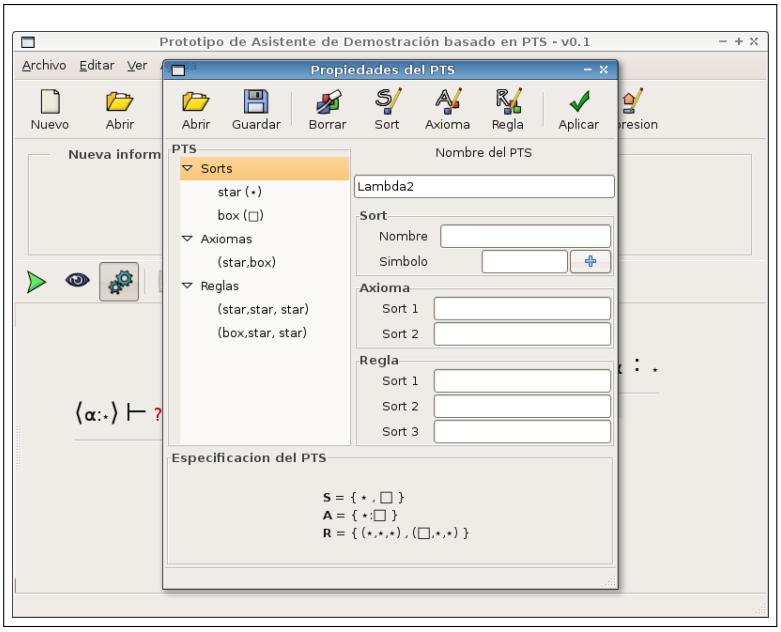

Figura A.9: Cargando la especificación del sistema $\lambda 2$

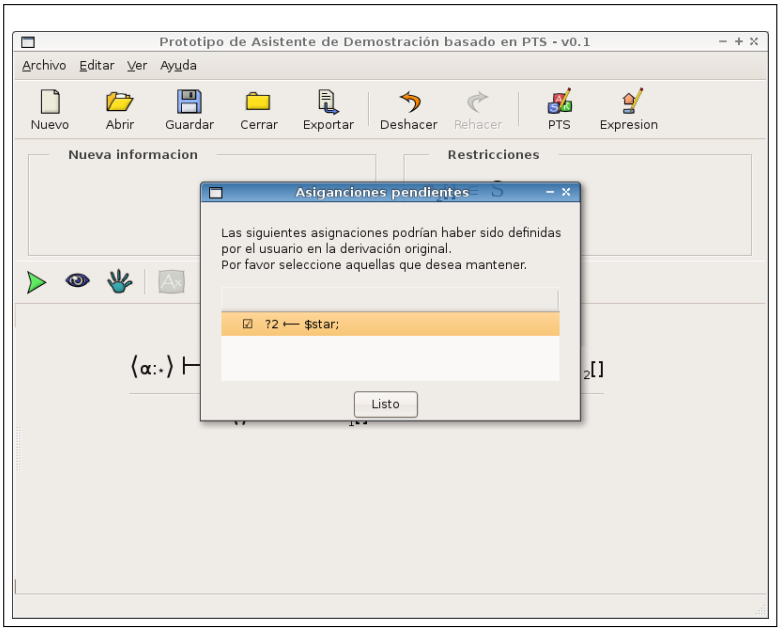

Figura A.10: Eliminación de premisas

cambios introducidos en aquel entonces deberían ser descartados. Una opción sería deshacer los últimos cambios efectuados, pero esto también nos retornaría a un estado anterior, en el cual aún nos encontrábamos trabajando bajo el sistema $\lambda\!\rightarrow$. De manera que optamos por emplear el sistema de eliminación de premisas. Para ello,

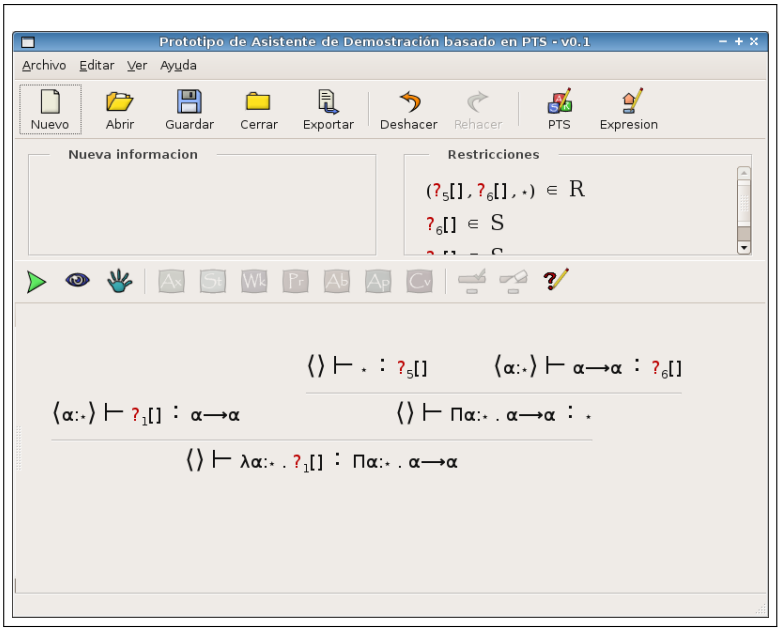

Figura A.11: Estado posterior a la eliminación de premisas

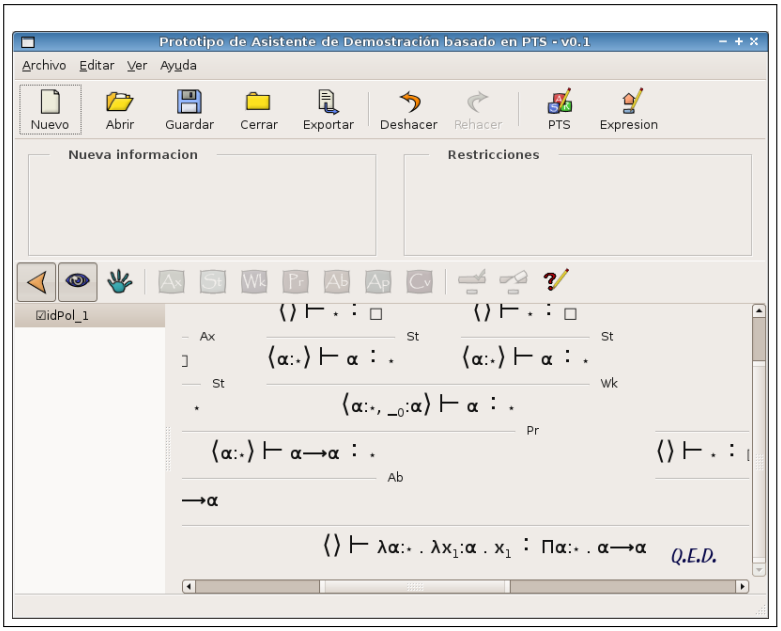

Figura A.12: La derivación ya concluida

seleccionamos el juicio $[] \vdash \Pi\alpha : \star . \alpha \to \alpha \ : \star$ y haciendo clic en el penúltimo botón de la barra de herramientas secundaria, procedemos a la eliminación de las premisas de dicho juicio. Durante el proceso, se nos presenta un cuadro de diálogo consultándonos sobre si deseamos conservar el valor que tenía previamente asignada la incógnita $?_2\,[]$ o

no, tal y como se muestra en la figura A.10. Como dicha expresión es el *sort* $\star$, el cual también nos servirá en nuestra nueva derivación, marcamos la casilla correspondiente a la asignación que deseamos conservar y procedemos a aceptar los cambios.

Una vez concluido el proceso de eliminación de premisas, estamos en condiciones de aplicar nuevamente la regla de *product* sobre el mismo juicio en que lo hicimos anteriormente. Sin embargo, a diferencia de lo ocurrido bajo el sistema $\lambda\!\rightarrow$, en el sistema $\lambda 2$ no existe una única regla definida en la especificación del *PTS*, de modo que en lugar de tener que demostrar que $\star : \star$, debemos verificar que $[] \vdash \star : ?_5[]$, quedando el tipo de $\star$ aún indefinido, tal y como se puede apreciar en la figura A.11.

Notemos además que el conjunto de restricciones que debe verificarse ya no se corresponde con el que teníamos anteriormente, ya que las reglas bajo las cuales estamos efectuando la nueva derivación difiere de la que teníamos entonces. De esta manera, podemos hacer uso de la regla de *axiom* sobre el nuevo juicio generado y proceder con el resto de la derivación de manera similar a la que hemos venido empleando hasta el momento.

Aplicando sucesivamente las reglas adecuadas, llegaremos finalmente a conseguir el término que se corresponde con el tipo cargado inicialmente. Una vez que la derivación ha concluido, es decir que todas sus hojas son axiomas, no existen incógnitas presentes ni tampoco restricciones que aún queden por ser verificadas, entonces decimos que la prueba ha concluido, tal y como se puede apreciar en la figura A.12.

Finalmente, podemos optar por guardar el resultado obtenido o exportar la derivación a código *LaTeX*. A continuación incluimos la derivación obtenida, empleando el formato de lista de juicios tabulados, exportada directamente desde nuestra herramienta:

Especificación del PTS: *"Lambda2"*

$$\begin{array}{lcl} \mathbb{A} & = & \{\star, \Box\} \\ \mathbb{S} & = & \{(\star, \Box)\} \\ \mathbb{R} & = & \{(\star, \star, \star), (\Box, \star, \star)\} \end{array}$$

Bajo el supuesto del PTS dado anteriormente, tenemos el siguiente teorema:

$$\Pi\alpha : \star \,.\, \alpha \;\rightarrow\; \alpha$$

Cuya prueba se obtiene mediante la derivación de la expresión:

$$
\begin{array}{ll}
[] \vdash \lambda\alpha : \star \,.\, \lambda x_1 : \alpha \,.\, x_1 \,:\, \Pi\alpha : \star \,.\, \alpha \to \alpha & \text{(Ab)} \\
\quad [\alpha : \star] \vdash \lambda x_1 : \alpha \,.\, x_1 \,:\, \alpha \to \alpha & \text{(Ab)} \\
\quad\quad [\alpha : \star, x_1 : \alpha] \vdash x_1 \,:\, \alpha & \text{(St)} \\
\quad\quad\quad [\alpha : \star] \vdash \alpha \,:\, \star & \text{(St)} \\
\quad\quad\quad\quad [] \vdash \star : \Box & \text{(Ax)} \\
\quad\quad [\alpha : \star] \vdash \alpha \to \alpha : \star & \text{(Pr)} \\
\quad\quad\quad [\alpha : \star] \vdash \alpha \,:\, \star & \text{(St)} \\
\quad\quad\quad\quad [] \vdash \star : \Box & \text{(Ax)} \\
\quad\quad\quad [\alpha : \star, \dagger_0 : \alpha] \vdash \alpha \,:\, \star & \text{(Wk)} \\
\quad\quad\quad\quad [\alpha : \star] \vdash \alpha \,:\, \star & \text{(St)} \\
\quad\quad\quad\quad\quad [] \vdash \star : \Box & \text{(Ax)} \\
\quad\quad\quad\quad [\alpha : \star] \vdash \alpha \,:\, \star & \text{(St)} \\
\quad\quad\quad\quad\quad [] \vdash \star : \Box & \text{(Ax)} \\
\quad [] \vdash \Pi\alpha : \star \,.\, \alpha \to \alpha : \star & \text{(Pr)} \\
\quad\quad [] \vdash \star : \Box & \text{(Ax)} \\
\quad\quad [\alpha : \star] \vdash \alpha \to \alpha : \star & \text{(Pr)} \\
\quad\quad\quad [\alpha : \star] \vdash \alpha \,:\, \star & \text{(St)} \\
\quad\quad\quad\quad [] \vdash \star : \Box & \text{(Ax)} \\
\quad\quad\quad [\alpha : \star, \dagger_0 : \alpha] \vdash \alpha \,:\, \star & \text{(Wk)} \\
\quad\quad\quad\quad [\alpha : \star] \vdash \alpha \,:\, \star & \text{(St)} \\
\quad\quad\quad\quad\quad [] \vdash \star : \Box & \text{(Ax)} \\
\quad\quad\quad\quad [\alpha : \star] \vdash \alpha \,:\, \star & \text{(St)} \\
\quad\quad\quad\quad\quad [] \vdash \star : \Box & \text{(Ax)}
\end{array}
$$

Apéndice B

Sintaxis de los archivos de PTS

Los archivos de configuración para especificaciones de *PTS* deben respetar la siguiente gramática:

```
<pts_file>      ::= <name> (<sort>)+ (<axiom>)+ (<rule>)+

<name>          ::= 'PTS:' '"' <string> '"'

<sort>          ::= '(' <sort_name> ')' ('{' <sort_symbol> '}')?
<sort_name>     ::= ( <letter> | <integer> )+
                  | '"' <string> '"'
<sort_symbol>   ::= <sort_sym_name> | <sort_sym_code>
<sort_sym_name> ::= <string>
<sort_sym_code> ::= '#' <integer>

<axiom>         ::= '(' <sort_name> ',' <sort_name> ')'

<rule>          ::= '(' <sort_name> ',' <sort_name> ',' <sort_name> ')'
```

Apéndice C

Sintaxis de los archivos de demostraciones

Los archivos que almacenan información sobre las derivaciones deben respetar la siguiente gramática:

```
<der_file>     ::= <pts_info> <der_info>

<pts_info>     ::= <pts_file>

<der_info>     ::= 'Theorem:' '"' <string> '"' (<judgement>)+

<judgement>    ::= '[' <context> ']' '|--' <expr> '::' <expr> <rule>

<context>      ::= (<ctx_elem> (';' <ctx_elem>)* )?
<ctx_elem>     ::= <id> '::' <expr>

<id>           ::= <named_id>
                 | <unnamed_id>
                 | <gen_id>
                 | '(' <id> ')'
<named_id>     ::= ( <letter> )+ ( <integer> )?
<unnamed_id>   ::= '!' <integer>
<gen_id>       ::= 'X' <integer>

<expr>         ::= <expr_app> ( '->' <expr_app> )*
<expr_app>     ::= <expr_parens> | ( <sub_expr> )+
<expr_parens>  ::= '(' <expr> ')'

<sub_expr>     ::= <expr_var>
                 | <expr_sort>
                 | <expr_unk>
                 | <expr_pi>
                 | <expr_abs>
```

```
<expr_var>    ::= <named_id>
                | <gen_id>
                | '(' <expr_var> ')'
<expr_sort>   ::= '$' '"' <string> '"' | '$' ( <letter> )+
<expr_unk>    ::= '?' <integer> ( '{' <subst> '}' )?
<expr_pi>     ::= '~' <id> ':' <expr> '.' <expr>
<expr_abs>    ::= '\' <expr_var> ':' <expr> '.' <expr>

<subst>       ::= '[' ( <subst_elem> ( ',' <subst_elem> )* )? ']'
<subst_elem>  ::= <expr> '/' <expr_var>

<rule>        ::= ( '[' <rule_name> ']' )?
<rule_name>   ::= 'AX'
                | 'ST'
                | 'WK'
                | 'AB'
                | 'AP'
                | 'PR'
                | 'CV'
```

Referencias

[Agd00] A system for incrementally developing proof and programs, 2000.

[Bar81] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland Publishing Co., 1981.

[Bar91] Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[Bar92] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992.

[Bar99] Gilles Barthe. Type-checking injective pure type systems. *J. Funct. Program*, 9(6):685–698, 1999.

[Ber88] Stefano Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Department of Computer Science, CMU, and Dipartimento Matematica, Universita di Torino, 1988.

[Ber90] Stefano Berardi. *Type dependence and constructive mathematics, PhD thesis*. PhD thesis, Dipartimento di Matematica, Università di Torino, 1990.

[Bro13] L. E. J. Brouwer. Intuitionism and formalism. *Bulletin of the American Mathematical Society*, 20(2):81–96, 1913.

[Cay99] A haskell-like language with a powerful type system based on dependent types., 1999.

[CF58] Haskell Brooks Curry and Robert Feys. *Combinatory Logic, Vol. I, Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1958.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Coq99] The coq proof assistant, 1999.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Proc. of Symposium on Automatic Demonstration, Versailles, France, Dec. 1968*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, Berlin, 1970.

[dLst] Gerard R. Renardel de Lavalette. Strictness analysis via abstract interpretation for recursively defined types. *Inf. Comput.*, 99(2):154–177, 1992, August.

[Epi04] A dependently typed programming language and an interactive programming environment., 2004.

[FI00] Daniel Fridlender and Mia Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, 2000.

[Fle06] Fabián Fleitas. Un ambiente de demostración basados en pts. trabajo final de licenciatura, fa.m.a.f. - universidad nacional de córdoba, 2006.

[Geu89] Herman Geuvers. Theory of constructions is not conservative over higher order logic. *Technical report, Computer Science Institute, Katholieke Universiteit Nijmegen*, 1989.

[Geu93] Herman Geuvers. *Logics and Type Systems*. PhD thesis, Computer Science Institute, Katholieke Universiteit Nijmegen, 1993.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'elimination des coupures dans l'analyse et la théorie des types. In Jens Erik Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92, Amsterdam, 1971. North-Holland.

[Gir72] Jean-Yves Girard. *nterprétation Fonctionelle et Élimination des Compures de l'Arithmétic d'Ordre Supérieur*. PhD thesis, These D'Etat, Universite Paris VII, 1972.

[GN91] Herman J. Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, apr 1991.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[has90] A general purpose, purely functional programming language featuring static typing, higher order functions, polymorphism, type classes, and monadic effects., 1990.

[Hey31] Arend Heyting. Die intuitionistische grundlegung der math. *Erkenntnis*, 2:106–115, 1931.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings, Symposium on Logic in Computer Science*, pages 194–204, Ithaca, New York, 22–25 June 1987. The Computer Society of the IEEE.

[How80] William Alvin Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, NY, 1980.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Math. Society Student Texts*. Cambridge University Press, Cambridge, 1986.

[LM88] Giuseppe Longo and Eugenio Moggi. Constructive natural deduction and its 'modest' interpretation. Technical Report CMU-CS-88-131, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1988.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. Phd thesis CST-65-90 (report ecs-lfcs-90-118), Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh, jul 1990.

[Mag94] Lena Magnusson. *The Implementation of ALF—a Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Phd thesis, Dept. of Computing Science, Chalmers Univ. of Technology and Univ. of Göteborg, 1994.

[Mar] Per Martin-Löf. A construction of the provable wellorderings of the theory of species.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Paris, France*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Ste72] Sören Stenlund. *Combinators, λ-terms and Proof Theory*. D. Reidel, Dordrecht, NL, 1972.

[Ter89] Jan Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Technical report, Department of Computer Science, University of Nijmegen, 1989.

[vBJ93] L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, jul 1993.

[Wil99] Roorda Jan Willem. Pure type system for functional programming, 1999.