



**PEDECIBA Informática**

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

---

# Diseño de un Lenguaje Tipado con Mecanismos de Reflexión de Código

Marcos Omar Viera Larrea

Trabajo de tesis para la obtención del grado de Magíster en Informática de la Universidad de la República en el programa de Maestría del área Informática del Pedeciba

Supervisor: Dr. Alfredo Viola
Instituto de Computación
Universidad de la República

Orientador: Dr. Alberto Pardo
Instituto de Computación
Universidad de la República

Presentación: 20 de agosto de 2007

*Diseño de un Lenguaje Tipado*
*con Mecanismos de Reflexión de Código*
Marcos Omar Viera Larrea

ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT07-18
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, agosto de 2007

# Resumen

En este trabajo se estudia el problema de diseñar un lenguaje de programación que pueda soportar ciertas formas de reflexión de manera de proveer de flexibilidad y a su vez garantice algún nivel de seguridad de tipado.

Se presenta la definición de un lenguaje con primitivas de reflexión de código, es decir, con la capacidad de manipular representaciones de código en tiempo de ejecución. Se propone un lenguaje funcional multi-etapas homogéneo, basado en un cálculo lambda simplemente tipado con un operador de punto fijo, anotaciones de etapas, sustitución explícita sobre el código y un mecanismo de *pattern matching*. Mediante este último se provee la capacidad de realizar análisis de código, utilizando patrones de código para la inspección de la estructura de expresiones citadas y su destrucción en sus partes componentes. El tipo para los valores de código captura el contexto de tipado para sus variables libres, pero no incluye el tipo del código. Por lo tanto, evaluar expresiones de código implica realizar un chequeo dinámico de tipos. Las expresiones citadas incluyen una anotación explícita de su contexto, la cual se utiliza para la inferencia dinámica de tipos durante el tipado de las expresiones de código.

Se sigue la propuesta de Sheard de utilizar el meta-lenguaje Ωmega como una herramienta para el diseño de lenguajes. En este sentido, se codificaron las semánticas estática y dinámica del lenguaje propuesto en términos de construcciones de Ωmega. Se argumenta la corrección del sistema de tipos del lenguaje con respecto a su semántica operacional en base al sistema de tipos del meta-lenguaje, concluyéndose que la evaluación de toda expresión bien tipada no falla.

**Palabras clave:** Reflexión, Sistemas de Tipos, Programación Multi-etapas, Análisis Intensional, Meta-lenguaje Ωmega.

# Agradecimientos

*Un sueño que se sueña solo*
*es solo un sueño.*
*Pero un sueño que se sueña juntos,*
*ya es una realidad.*

Raul Seixas

Con la presentación de este trabajo culmina para mí una etapa, la cual significó mucho esfuerzo y trabajo. Mirando hacia atrás, siento que tengo muchísmio que agradecer. Ni lo que pasó ni lo que pasará serían posibles sin la colaboración y afecto que he recibido a lo largo de todo este tiempo. Seguramente falten muchos nombres, desde ya pido disculpas por mi espantosa memoria.

Quiero agradecer a todos aquellos que participaron en mi formación, y han contribuído a generar en mí una cierta pasión por lo que hago. Debo agradecer mucho a quienes me guiaron en mis primeros pasos, como Dalton Martínez e Iván Bentancur. Quiero recordar la motivación en el liceo de Federico de Pallejas. A Daniel Perovich y Andres Vignaga, porque jugaron un papel muy importante en mi carrera. A Alfredo "Tuba" Viola, por esas largas charlas colmadas de consejos, por el afecto. A Alberto Pardo, por la paciencia, y porque es tanto o más culpable que yo del resultado de este trabajo. Muchas gracias a ellos y a todos los demás.

Estoy muy agradecido con Pablo "Fidel" Martínez López por la gran cantidad de comentarios, que aportaron mucho a la calidad de la tesis. También tengo que agradecer a Laura Bermúdez, por su ayuda, eficiencia y actitud siempre colaborativa.

Muchas gracias a mis compañeros del Instituto de Computación, un lugar en donde disfruto trabajar y donde he ganado muchos amigos. A mis compañeros de sala, Daniel Calegari, Jorge Corral, Mónica Martínez, Diego Rivero, Leonardo Rodríguez y Federico Sotto, por el aguante. Gracias Leíto por escuchar mis locuras como si fueran interesantes.

A los amigos que gané estudiando en Facultad, porque por suerte pasa el tiempo y cada vez hablamos menos de computación. Muchas gracias a mis amigos de San José, gracias por no saber nada de computación. ¡Qué sería de mí sin ustedes!

Quiero agradecer a mi familia. A mis padres, por el cariño, por la motivación, por el esfuerzo, ¡por todo! Sinceramente no encuentro palabras para expresar todo lo que tengo que agradecerles, sólo puedo decir que me siento orgulloso de ser su hijo.

Por último, quiero agradecer especialmente al amor de mi vida. ¡Gracias Caro! Por estar ahí, por bancarme en todo y aguantar tanto. Gracias por el amor que me brindás. Por tu locura, que me encanta, y por sobrevivir a la mía. Te quiero mucho mi amor, y seguro te voy a seguir agradeciendo toda la vida.

Con el fin de esta etapa comienza una nueva, con otro camino para recorrer. Los necesito a todos para seguir andando.

# Índice general

# Capítulo 1

# Introducción

*¿Quieres tú, tal vez, arrastrarnos a la angustia del miedo y del terror*
*hasta el extremo de llevar a nuestra alma al campo de la locura?*
*Dime, ¿qué te propones? ¿Qué es lo que intentas?*
*Dinos, ¿qué hemos de hacer?*

Hamlet

Con la evolución de los sistemas de computación y su complejidad creciente, resulta cada vez más importante tener en cuenta formas de incrementar su flexibilidad. Se torna muy complicado abarcar todas las características de un sistema muy complejo, o de naturaleza dinámica, sólo en las etapas de diseño y codificación, y resulta necesario poder analizarlo y/o manipularlo en tiempo de ejecución. Una forma posible de proveer a los sistemas con la habilidad de evolucionar durante su propia ejecución, es que los lenguajes de programación puedan soportar ciertas formas de **reflexión**, lo que se entiende como la habilidad de "*razonar sobre sí mismo*", es decir que un proceso pueda manipular su propio estado. Algunos ejemplos de uso de reflexión son: *debugging*, mantenimiento, optimización y adaptación de programas al medio en que se ejecutan. La implementación de reflexión que más se ha utilizado en la práctica es el API de reflexión del lenguaje Java, que permite examinar las clases, interfaces y objetos de un programa Java en ejecución. Un ejemplo del uso del API de reflexión son los JavaBeans[1], donde se utiliza para obtener las propiedades de los componentes de manera de que puedan ser manipulados mediante una herramienta visual de construcción. También se utiliza ampliamente para la implementación de *frameworks*. Por ejemplo, Castor es un *framework* que permite la persistencia de objetos Java como, entre otras cosas, documentos XML. Los datos en el modelo de objetos Java se transforman en archivos XML, y viceversa. Castor usa reflexión para determinar la estructura de los datos, y crear los elementos XML de acuerdo a esto.

Por otro lado, además de incrementar la flexibilidad, es deseable poder garantizar, en etapas tempranas, el comportamiento que tendrá un programa al momento de ejecutarse. Una forma de proveer de esto es mediante lenguajes de programación con **sistemas de tipos**, los cuales imponen ciertas restricciones a los programas que se pueden construir, clasificando los valores y las expresiones en tipos, y definiendo cómo se pueden manipular esos tipos y de qué manera pueden interactuar. Respetar esas restricciones implica que se puedan prever ciertos errores en los programas, evitando así que se comporten de una manera no deseada. La propiedad de los lenguajes de detectar esta clase de errores se

[1] Los JavaBeans son componentes de software reutilizables que se pueden manipular visualmente en una herramienta de construcción.

llama **seguridad de tipos**. Cuanto más temprana sea la etapa en la cual se detectan los errores de tipos, mayor es el nivel de seguridad que se provee. Por esto se suele asociar a este concepto con el de sistemas de tipos estáticos, que detectan los errores de tipos en tiempo de compilación. Además de proveer de seguridad, los tipos brindan información sobre los programas que puede ser muy útil, tanto para los humanos (como documentación del propósito y el comportamiento de los programas) como para el compilador (de manera de poderse realizar operaciones de optimización de código).

Es claro que la flexibilidad, vía reflexión, y la seguridad de tipos son conceptos que resultan contrapuestos, y que por lo tanto, proveer de uno usualmente va en detrimento del otro. El objetivo de esta tesis es estudiar esta problemática, diseñándose un lenguaje funcional que brinde algún mecanismo de reflexión, realizando un compromiso entre flexibilidad y seguridad.

## 1.1. Reflexión de Código

Un programa debería disponer de algún mecanismo que le permita acceder al mismo con el objetivo de poder manipular su estado. Se puede describir el comportamiento del programa utilizando un modelo de computación basado en un proceso interno, usualmente llamado intérprete, que lo lleva a cabo. Dado que el intérprete es una abstracción, que no tiene porque estar ligado a la implementación del lenguaje[2], existen varios posibles enfoques para especificar sus componentes. En este trabajo se tomará una perspectiva denotacional, en donde la evaluación se define en base a la expresión y un contexto determinado por ambientes y continuaciones. Por lo tanto, teniendo acceso a alguna de estas estructuras internas se puede manipular el proceso de evaluación del programa.

Friedman y Wand [FW84] introdujeron los conceptos de reificación, para referirse al proceso de convertir un componente del intérprete en un objeto que el programa pueda manipular, y de reflexión para el proceso inverso. Desde la perspectiva denotacional, un ejemplo de los componentes que pueden ser reificados es el código del programa. Llamaremos **reflexión de código** a la capacidad de manipular representaciones de código en tiempo de ejecución. De esta forma se puede por ejemplo implementar una aplicación junto a herramientas que razonen sobre ésta y/o la modifiquen, todo utilizando un único lenguaje y aprovechando así el conocimiento de los programadores, tanto del lenguaje elegido como del dominio del sistema.

La reificación de código puede ser llevada a cabo por un mecanismo de **citado**, como el que incluye Lisp. En Lisp, utilizando la función `quote`, que significa citar, se reifica la expresión que se pasa como argumento, evitando su evaluación y permitiendo que el código pueda ser manipulado. Por ejemplo, mientras que `(+ 2 2)` evalúa a `4`, `(quote (+ 2 2))` evalúa a la representación del código `(+ 2 2)`. La operación inversa a la reificación se realiza con la función `eval`, que lleva a cabo la evaluación de un código reificado. Por lo tanto, la evaluación de `eval (quote (+ 2 2))` tiene como resultado `4`.

Las construcciones de citado dividen la computación en etapas, en donde el resultado de una etapa es el código de la siguiente. Es por esto que a este tipo de meta-programación se la llama **programación multi-etapas** [Tah99].

Es deseable que un lenguaje reflexivo cuente con primitivas de síntesis y análisis de las mismas para poder manipular las estructuras reificadas. Esto es, que se puedan construir nuevas estructuras y analizar la existentes. En Lisp, por ejemplo, se pueden construir expresiones reificadas a partir de otras utilizando anti-citaciones, que se comportan como "huecos" en un *template*. Además, el código se representa como listas, por lo que se

[2] No es necesario que el lenguaje sea interpretado.

puede manipular mediante las operaciones usuales de listas, lo cual permite analizar su estructura.

Como Lisp es un lenguaje no tipado [CW85], no se puede brindar ninguna seguridad sobre el código generado. Existen varios lenguajes multi-etapas tipados que proveen de construcciones de citado análogas a las de Lisp. La mayoría de estos lenguajes, como MetaML [TS00], fueron concebidos como generadores de código seguro y optimizado. Esto lleva a que no se preste mayor atención a la capacidad de análisis de los mismos, por lo que no se ha desarrollado suficientemente un tratamiento formal de las características de los lenguajes que puedan analizar código.

## 1.2. Objetivos del Trabajo

En este trabajo se propondrá un lenguaje que combine las principales características de varios de los lenguajes existentes en la literatura, enfatizando en la capacidad de reflexión de código brindada, pero sin perder de vista la seguridad de tipos.

El lenguaje propuesto es un lenguaje multi-etapas que provee soporte para realizar **análisis intensional**, entendiéndose esto como la habilidad de observar la estructura de los programas objeto. Esto se llevará a cabo mediante un mecanismo de *pattern matching* que se utilizará para inspeccionar la estructura de las expresiones citadas y descomponerlas entre sus sub-expresiones componentes.

Una característica de los lenguajes como MetaML es que aseguran estáticamente que el código generado dinámicamente está bien tipado. Para esto se utiliza, para las expresiones citadas, el tipo $\langle\tau\rangle$ (o `cod` $\tau$), que significa código de $\tau$, donde $\tau$ es el tipo de la expresión que se está citando. El problema de este tipado es que se excluyen muchas funciones que descomponen o atraviesan la estructura de las expresiones. Por ejemplo, no se puede construir una función recursiva que recorra el código de un par `(9,true)`, que tiene tipo $\langle\texttt{int} \times \texttt{bool}\rangle$, ya que para recorrer cada sub-expresión se debe hacer una llamada recursiva con valores cuyos tipos ($\langle\texttt{int}\rangle$ y $\langle\texttt{bool}\rangle$) no coinciden con el de la llamada original. Existen otras propuestas [GMO03, SSP98] en las que se define un único tipo `cod` para el código, realizándose un chequeo de tipos en tiempo de ejecución del código generado. De esta manera se cede un poco de seguridad estática a favor de obtener flexibilidad. En esta tesis se seguirá esta última idea, teniendo un tipo con forma $\texttt{cod}^{\Gamma}$, donde $\Gamma$ es un contexto de tipos que refleja las variables libres de la expresión citada. Este enfoque es menos flexible que el del tipo único, pero permite manipular código abierto y asegurar estáticamente que sólo se evaluará código cerrado (sin variables libres).

Se seguirá la propuesta de Sheard [She05b] sobre el uso del lenguaje Ωmega como herramienta para el diseño de lenguajes. Las semánticas del lenguaje se codifican como un meta-programa, y el sistema de tipos de Ωmega garantiza que los programas en el meta-nivel respeten las propiedades impuestas por éstas. En base a esta codificación se demostrará que todo término bien tipado evalúa a un valor del mismo tipo, o reduce infinitamente.

Muchos de los resultados de esta tesis fueron publicados, en una versión reducida, en [VP06].

## 1.3. Estructura del Documento

Este trabajo tiene la estructura que se describe a continuación. En el Capítulo 2 se define el concepto de reflexión en los lenguajes de programación. Se describen las distintas funcionalidades que puede proveer un lenguaje para realizar computación reflexiva,

y se realiza una clasificación de las mismas. En base a una perspectiva denotacional se analizan las distintas estructuras que un programa puede manipular para operar sobre su computación. En el Capítulo 3 se profundiza en los mecanismos de reflexión de código en lenguajes funcionales tipados, analizando su estado del arte. Se describen las semánticas de los principales lenguajes de programación tipados que proveen de algún mecanismo de manipulación de código. En el Capítulo 4 se presenta un lenguaje de programación que provee de los mecanismos de reflexión de código deseados. Se describen estos mecanismos y se muestra su semántica. En el Capítulo 5 se codifica la semántica del lenguaje definido utilizando el lenguaje de meta-programación Ωmega. En base al sistema de tipos del meta-lenguaje, se argumenta sobre algunas propiedades de la semántica. Finalmente, en el Capítulo 6 se presentan algunas conclusiones y se definen líneas de trabajo futuro.

# Capítulo 2

# Reflexión

*Te advierto, quién quiera que fueres,*
*¡oh! tú que deseas sondear los arcanos de la Naturaleza,*
*que si no hallas dentro de tí mismo,*
*aquello que buscas, tampoco podrás hallarlo fuera.*
*Si tu ignoras las excelencias de tu propia casa,*
*¿cómo pretendes encontrar otras excelencias?.*
*En ti se halla oculto el tesoro de los tesoros.*
*¡Oh! hombre, conócete a ti mismo y conocerás al Universo y a los Dioses.*

Inscripción que figuraba en el Templo de Apolo, en Delfos.

En este Capítulo se definirá el concepto de reflexión computacional, y se presentará un marco general desde el cual se pueden estudiar los lenguajes que presentan algún tipo de reflexión.

## 2.1. Definición

El término *reflection*, al igual que su traducción al español **reflexión**, tiene raíz en el verbo latino *reflectĕre* que significa *"volver atrás"*. Desde el punto de vista físico es el fenómeno mediante el cual la propagación de una onda cambia de sentido sobre ciertas superficies, permitiéndose, por ejemplo, que una imagen se forme en un espejo. En un contexto psicológico significa que el sujeto de observación de una entidad es la entidad misma. El filósofo inglés John Locke definió reflexión como *"el conocimiento que el espíritu adquiere de sus propias operaciones y del modo de efectuarlas, por lo cual llega a tener en el entendimiento ideas de esas operaciones"*.

### 2.1.1. Reflexión Computacional

En base a las asociaciones autoreferenciales de las definiciones mencionadas, al comienzo de la década de los 80 Brian Smith [Smi82, Smi84] introdujo el concepto de reflexión en el marco de las ciencias de computación, refiriéndose a *"la habilidad de un agente de razonar no sólo introspectivamente, sobre sus propios e internos procesos de pensamiento, pero también externamente, sobre su comportamiento y situación en el mundo"*. Con el objetivo de formalizar una teoría de **reflexión computacional** elaboró una hipótesis de reflexión [Smi82].

> ***Hipótesis de Reflexión:*** *así como un proceso computacional puede ser construido para razonar sobre un mundo externo en virtud de la inclusión de un proceso ingrediente (intérprete) manipulando formalmente representaciones de ese mundo, entonces también un proceso computacional puede ser hecho para razonar sobre sí mismo en virtud de la inclusión de un proceso ingrediente (intérprete) manipulando formalmente representaciones de sus propias operaciones y estructuras.*

A lo largo de los últimos 20 años se han planteado varias definiciones de reflexión computacional, que van desde algunas muy generales, que la definen como "*la habilidad de un sistema de realizar una computación sobre sí mismo*"[LO93], hasta las más específicas, como la definición de Malenfat, Jacques y Demers [MJD96] que indica que reflexión es "*la habilidad de un programa de observar o cambiar su propio código así como todos los aspectos de su lenguaje de programación (sintaxis, semántica, o implementación), incluso en tiempo de ejecución*".

La definición de Maes [Mae87a], que queda en un punto medio entre las dos anteriores, es la que mejor se adecúa a los fines de esta tesis.

> ***Reflexión computacional*** *es el comportamiento exhibido por un sistema reflexivo, donde un sistema reflexivo es un sistema computacional que trata sobre sí mismo mediante una conexión causal.*

Por lo tanto un sistema reflexivo se incluye a sí mismo dentro de su propio dominio. El hecho de que esta inclusión se deba realizar mediante una **conexión causal** significa que un cambio realizado sobre la representación del sistema surte efecto en el sistema mismo.

Esta capacidad de un sistema de "conocerse a sí mismo" puede ser útil, por ejemplo, para la implementación de aplicaciones de *debugging*, dado que para depurar un programa se debe examinar el estado del proceso en medio de su ejecución y relacionarlo con su código fuente. Utilizando reflexión, el programa puede elegir cuándo y cómo expone su estado al desarrollador, y puede ser corregido durante su ejecución. Incluso, un sistema podría llegar a detectar errores de su propia ejecución y ofrecer alguna interface para corregirlos. La reflexión también es útil en aplicaciones altamente customizables o de dominio dinámico, que permiten realizar cambios de configuración o funcionalidad durante su ejecución. Una forma de realizar estas modificaciones es mediante la inserción de pequeños trozos de código, ya sea como scripts, plug-ins, add-ons, etcétera. El editor Emacs [Sta02], por ejemplo, puede ser customizado con scripts escritos en un dialecto de Lisp. Accediendo a los procedimientos y estado interno del editor, estos scripts pueden desde modificar variables de configuración hasta proveer de nuevas funcionalidades. Para esto es necesario tener algún tipo de reflexión, ya que el script debe poder acceder a la estructura interna del programa anfitrión, y este último debe poder permitir examinar y modificar su propia estructura durante su ejecución.

### 2.1.2. Lenguajes Reflexivos

Según Kiczales [KdRB91] un **lenguaje reflexivo** es aquel que permite manipular representaciones explícitas de aspectos implícitos del mismo lenguaje. Sin embargo el punto preciso en el que un lenguaje con facilidades de reflexión se vuelve un lenguaje reflexivo no está bien definido [Riv96]. En [MJD96], por ejemplo, se argumenta que una de las principales dificultades para encontrar una definición precisa de reflexión radica en la cantidad de lenguajes que exhiben algún tipo de comportamiento reflexivo, o proveen

de mecanismos de acceso a ciertos aspectos del estado del programa, y que un lenguaje es reflexivo *"cuando provee reflexión (completa)"*. Esta definición parece ser demasiado pretenciosa, ya que reflexión completa supone no imponer límites en cuanto a lo que el programa puede observar o modificar, para lo cual existen límites teóricos identificados, como ser el teorema de incompletitud de Gödel, situaciones paradójicas, entre otros.

En este trabajo consideraremos la visión de Maes [Mae87a] de que un lenguaje es reflexivo *"si reconoce reflexión como un concepto fundamental de programación, y provee de herramientas para manipular la computación reflexiva explícitamente"*. Estos lenguajes deben tener una **arquitectura reflexiva** con las siguientes caracterísitcas:

**Representación del Sistema** Se debe dar a cualquier sistema en ejecución acceso a datos que representan aspectos del sistema. Mediante la manipulación de esos datos los sistemas pueden realizar computación reflexiva.

**Conexión Causal** Se debe garantizar la conexión causal entre esos datos y los aspectos del sistema que representa. De esta forma la modificación que estos sistemas realizan a su propia representación se refleja en su propio estado y computación.

## 2.2. Mecanismos Reflexivos

Se le llama **mecanismos reflexivos** a las facilidades que provee un lenguaje de programación para permitir realizar alguna forma de computación reflexiva [Rod06]. A continuación se presenta el modelo de computación a partir del cual se describirán luego los posibles mecanismos de reflexión.

### 2.2.1. Modelo de Computación

Un **proceso** es un conjunto coherente de eventos que interactúan con datos del mundo externo durante un determinado tiempo. Usualmente se puede ver a un proceso como un programa corriendo y al mundo externo como su dominio de aplicación, cuyos datos se organizan en **estructuras**. En la Figura 2.1(a) se ilustra la interacción entre un proceso, representado como una circunferencia, y su dominio de aplicación (estructuras externas), que se representa como un paralelogramo. Como se muestra en la Figura 2.1(b) un proceso se compone a su vez de un **campo estructural** (estructuras internas) y un **procesador**. El campo estructural está compuesto por el programa y/o las estructuras de datos que mantienen el estado del mismo. Por su parte el procesador es el proceso interno que lleva a cabo el programa, usualmente llamado intérprete[1].

### 2.2.2. Acceso a las Estructuras Internas

Siguiendo el modelo de reducción de procesos de la Figura 2.1 el procesador se podría reducir a su vez en otro procesador y una serie de estructuras internas, como se muestra en la Figura 2.2(a). El programa que interpreta este nuevo procesador es un intérprete del lenguaje en el cual está escrito el programa del proceso. En el caso en el que este intérprete esté escrito en el mismo lenguaje que interpreta, tenemos un intérprete meta-circular, o un **procesador meta-circular** como decide llamarlos Smith por las mismas razones que se explicaron en la sección 2.2.1. Si continuamos estas reducciones de manera infinita llegaríamos a tener, como en la Figura 2.2(b), una pila infinita de procesadores

[1]Smith prefirió llamarlo procesador para evitar confusiones con la noción de programas interpretados.

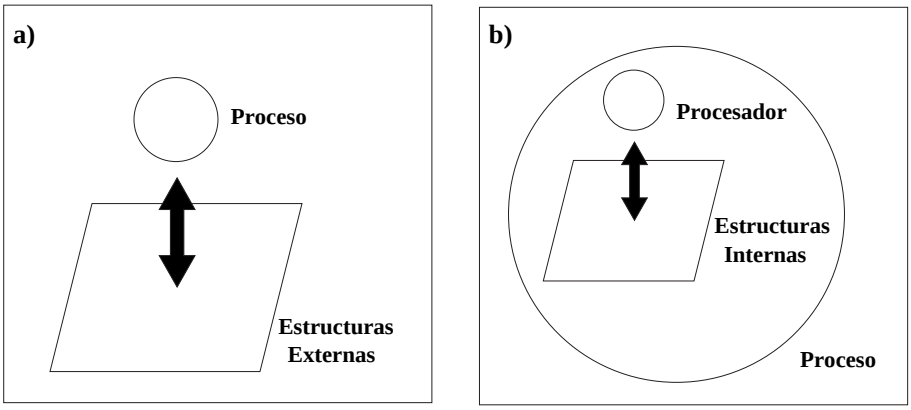


Figura 2.1: Reducción de un Proceso

meta-circulares (**procesadores reflexivos**), donde cada uno interpreta al que tiene abajo y el último (nivel 1) interpreta al programa del usuario.

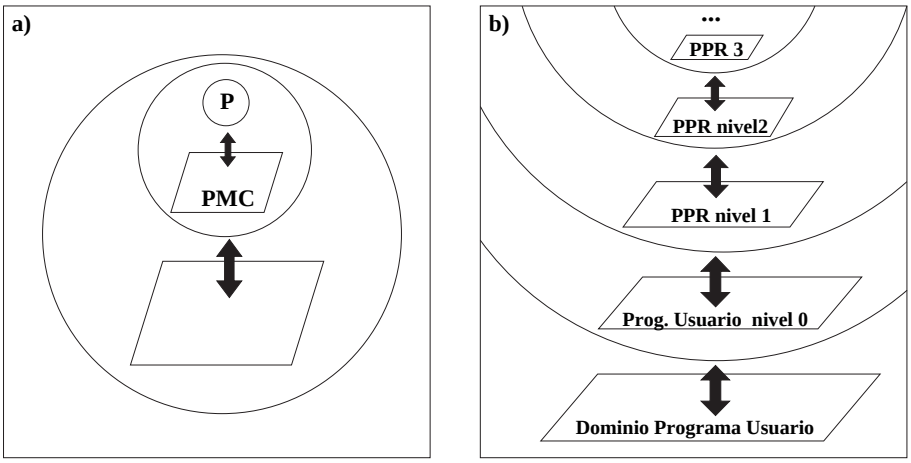


Figura 2.2: Torre Reflexiva

El lenguaje 3-Lisp [Smi82, Smi84, dRS84] es un dialecto reflexivo de Lisp, que basa su poder reflexivo en esta arquitectura, a la cual denomina **torre reflexiva**, y al uso de **procedimientos reflexivos**. Al invocarse un procedimiento reflexivo este se ejecuta en un nivel por encima que aquel desde el cual se realizó la invocación. De esta forma un programa puede acceder a su propia interpretación. La Figura 2.3(a) ilustra este traspaso de código desde un nivel a otro, que es posible porque tanto el proceso como su procesador están escritos en un mismo lenguaje. Dado que ese código forma parte ahora del código de su procesador, sus estructuras internas pasan a ser su dominio de aplicación, por lo que puede manipularlas asegurando la conexión causal.

La solución anterior es muy potente, pero de la misma manera también lo es compleja y difícil de implementar eficientemente [Ala04]. Por este motivo Friedman y Wand [FW84], intentando brindar un modelo más formal de reflexión, mostraron como descomponer el concepto de reflexión en dos operaciones llamadas **reificación** y **reflexión**. Mediante la reificación se puede modelar el acceso a las estructuras internas sin tener que contar con una torre de procesadores.

***Reificación:*** *la conversión de un componente del intérprete en un objeto que*

*el programa puede manipular.*

De esta forma, como se muestra en la Figura 2.3(b), se puede acceder a las estructuras internas colocándolas explícitamente como parte de su dominio de aplicación, a diferencia del mecanismo descrito anteriormente (Figura 2.3(a)), donde el acceso se da de forma implícita mediante la inserción de código en un ambiente en el cual estas estructuras forman parte de su dominio.

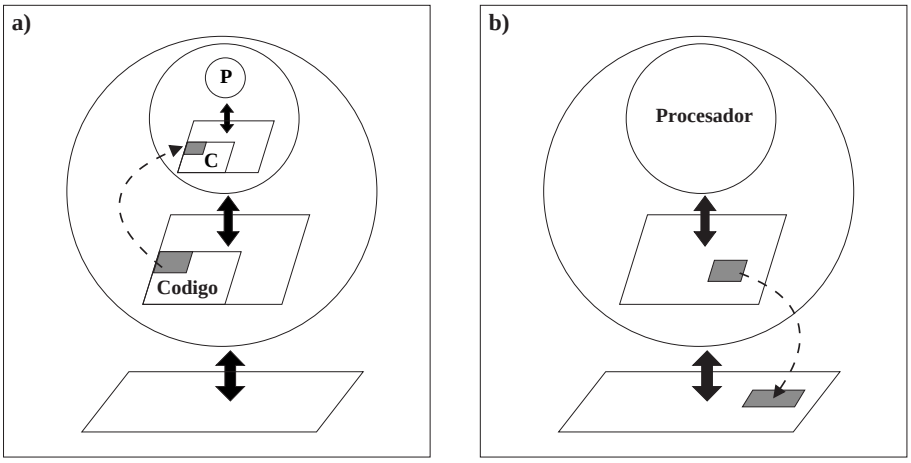


Figura 2.3: Acceso a las Estructuras Internas

La conexión causal entre los objetos reificados y las estructuras internas del procesador se obtiene por medio de la operación de **reflexión**.

> ***Reflexión:*** *la operación de tomar un valor manipulable por el programa e instalarlo como un componente del intérprete.*

Dado que utilizar el término reflexión tanto para llamar a la operación inversa de reificación como al concepto general puede llevar a confusiones [SF96], de ahora en más esta operación será llamada **absorción**[2].

En un trabajo posterior, Wand y Friedman [WF86] mostraron como agregar la torre de procesadores a su modelo utilizando meta-continuaciones. De esta forma los autores lograron una formalización de la torre reflexiva sin utilizar reflexión.

### 2.2.3. Computación Reflexiva

Un lenguaje no se puede llamar reflexivo si sólo provee operaciones de reificación y absorción de estructuras internas y no brinda mecanismos para manipularlas. Es más, dado que estas operaciones son inversas, su composición debería resultar en la identidad [MF96]. Cuando un sistema manipula a otro sistema, al primero se lo llama **meta-sistema** y al otro **sistema objeto**. Por lo tanto para tener reflexión se debe tener un meta-sistema conectado causalmente, cuyo sistema objeto sea él mismo.

El programa de un meta-sistema se llama **meta-programa** y especifica meta-computación sobre su sistema objeto, que en el caso de reflexión se llama **computación reflexiva**. Según Sheard [She01] los meta-programas se pueden categorizar entre **analizadores** y **generadores**. Un lenguaje que permite escribir analizadores debe contar con primitivas

[2]En varios trabajos posteriores a [FW84] se denomina absorción a esta operación, por ejemplo [Ste94, Meu98, Ala04].

de **análisis**, que brinden la capacidad de examinar las estructuras del sistema objeto. En cambio para programar generadores se debe poder construir nuevas estructuras, lo que llamaremos **síntesis**.

Los mecanismos reflexivos se resumen en la Figura 2.4. Mediante la reificación se pasa una estructura interna al dominio del sistema. Estas estructuras pueden ser analizadas y/o se pueden sintetizar nuevas estructuras (o modificaciones de las anteriores). Finalmente, a través de la absorción, las estructuras sobre las que se realizó meta-programación pueden pasar a convertirse en estructuras internas.

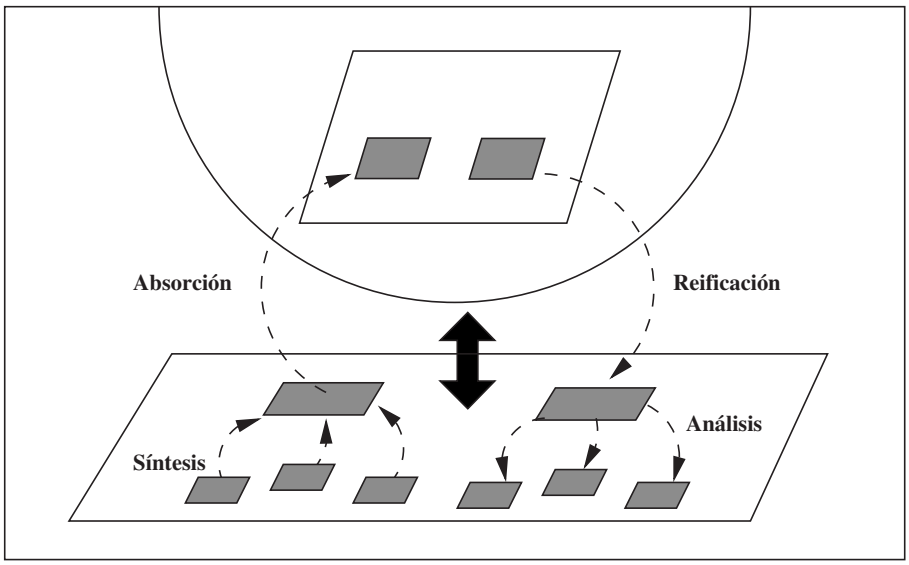


Figura 2.4: Mecanismos de Reflexión

Al contar con primitivas para realizar reificación y análisis un lenguaje está proveyendo la capacidad de programar sistemas que, al examinar sus estructuras internas, pueden razonar sobre sí mismos. A esto se lo llama **introspección** [BGW93].

> ***Introspección*** *es la habilidad de un programa de observar y por lo tanto razonar sobre su propio estado.*

Por su parte, las operaciones de síntesis y absorción permiten realizar lo que se conoce como **interseción**, que significa que un sistema puede modificar su comportamiento mediante la construcción de nuevas estructuras y su posterior internalización [BGW93].

> ***Intercesión*** *es la habilidad de un programa de modificar su propio estado de ejecución, o alterar su propia interpretación o significado.*

Este es un buen marco para poder analizar las capacidades reflexivas de los lenguajes [Ala04]. Para tener reflexión real es deseable que un lenguaje provea todas estas operaciones.

## 2.3. Clasificaciones

En esta sección haremos una reseña de las clasificaciones de reflexión que, a nuestro juicio, son las más importantes.

### 2.3.1. Por Paradigma

El concepto de reflexión se encuentra presente en muchos paradigmas de programación. Según Demers y Malenfant [DM95] ha sido estudiado especialmente en los paradigmas de programación funcional, lógica y orientada a objetos. Entre estos enfoques se pueden identificar ciertas similaridades así como diferencias fundamentales.

#### Programación Funcional

Los lenguajes 2-Lisp y 3-Lisp, creados por Smith para formalizar el concepto de reflexión, son dialectos del lenguaje funcional Lisp. Esto inspiró mucho trabajo en la comunidad de programación funcional en la década de los 80, sobre todo en la comunidad Lisp. En la última década la metaprogramación se ha transformado en un área formal de estudio, y ha llevado a la creación de varios lenguajes funcionales tipados [Tah99, Tah03, SP02, NP05, GMO03] que cuentan con ciertos mecanismos de reflexión. En el capítulo siguiente se analizarán algunos de estos lenguajes, ya que este es el paradigma en el que se trabajará en esta tesis.

#### Programación Orientada a Objetos

El concepto de reflexión en el contexto de la orientación a objetos fue introducido por Maes [Mae87a, Mae87b] en el lenguaje 3-KRS, alegando que "*ajusta más naturalmente en el espíritu de la programación orientada a objetos*". Esto se debie al enfoque en la abstracción que tiene el paradigma, el cual se logra mediante la encapsulación de las estructuras de datos y los procedimientos en entidades llamadas **objetos**. Según Maes "*es natural pensar que un objeto no solamente realice computación sobre su dominio, sino también sobre cómo puede realizar esa computación*".

Cazzola [Caz98] definió una taxonomía para los lenguajes reflexivos orientados a objetos, en la cual se distinguen dos grandes familias: **reificación de objetos** y **reificación de comunicación**. La primera consiste en reificar el tipo del objeto asociando cada entidad base a una o más meta-entitidades, mientras que en la segunda se reifican las interacciones de las entidades base en meta-entidades específicas.

En el enfoque de reificación de objetos se pueden distinguir dos modelos:

**Modelo de Meta-Clase** En la programación orientada a objetos una clase define las propiedades y comportamiento de un tipo de objetos concreto. En este modelo la clase de un objeto es su meta-objeto, existiendo como una entidad manipulable. Los meta-objetos pueden ser reificados por sus meta-meta-objetos, que en el caso de las clases serían sus meta-clases, modelando así la torre reflexiva. Dado que todas las instancias de una clase tienen el mismo meta-objeto, todas tienen el mismo meta-comportamiento. Este enfoque es común en lenguajes derivados de Smalltalk, donde la manipulación de clases como objetos existe desde antes que el concepto mismo de reflexión, como NeoClassTalk [Riv97].

**Modelo de Meta-Objeto** En lugar de indentificar al meta-objeto con la clase, en este modelo los meta-objetos son instancias de una clase específica (o derivada) para los mismos. Se establece una relación de "clientela" entre los meta-objetos y los objetos, donde los primeros interceptan los mensajes dirigidos a los últimos. Cada meta-objeto puede referenciar a varios objetos, y cada objeto puede estar conectado a lo largo del tiempo a más de un meta-objeto.

En el enfoque de reificación de comunicación se pueden distinguir dos modelos:

**Modelo de Reificación de Mensaje** En este modelo los mensajes son entidades de primera clase, que reifican las acciones que deberían ser realizadas por los métodos de los objetos. Los mensajes no se asocian a los objetos que los generan, y no pueden acceder a su información estructural.

**Modelo de Reificación de Canal** Este modelo extiende al anterior, definiendo un canal lógico que asocia a los objetos involucrados en la comunicación. Un canal se define por la tripla compuesta por los objetos que conecta y por el tipo de meta-computación que realiza.

#### Programación Lógica

En la comunidad de programación lógica se han estudiado conceptos similares a los de reflexión, pero de una forma bastante independiente a las anteriores. En lenguajes como Prolog existen mecanismos que permiten manipular programas en tiempo de ejecución, de manera de poder referirse explícitamente a las teorías y discutir su derivabilidad. Mediante el uso de predicados como `clause/1`, `assert/1`, `retract/1` y `call/1`, los objetivos y las cláusulas son tratados como objetos de primera clase representados por términos.

### 2.3.2. Por Tiempo

Dependiendo del momento en el que se aplique la reflexión podemos distinguir entre reflexión en tiempo de **compilación**, **carga**, **enlace** y **ejecución**.

Llamaremos reflexión en tiempo de compilación a la que puede ser llevada a cabo completamente durante la fase de análisis estático. Claramente este tipo de lenguajes limita mucho la reflexión que se pueda brindar. Algunos lenguajes de este tipo son CRML [HS93] y Template Haskell [SP02].

Una mayor capacidad de reflexión se obtiene cuando la misma puede ser llevada a cabo mientras el programa se está ejecutando. De esta manera se puede, por ejemplo, realizar computación reflexiva que depende de entradas provistas en tiempo de ejecución. En este trabajo se estudiará este tipo de reflexión.

La clasificación entre tiempo de compilación y ejecución es importante por motivos de diseño y eficiencia, aunque es claro que la primera puede ser implementada con la segunda [Ala04].

### 2.3.3. Por Tipado

Los lenguajes con **sistemas de tipos** tienen la capacidad de restringir los valores a los que las expresiones pueden evaluar. De esta forma se logra detectar y evitar ciertos errores, así como mejorar la comprensión del código. Pero también se imponen restricciones a cómo los programas pueden ser escritos, por lo que se pierde cierta flexibilidad.

Dado que el concepto de reflexión está muy ligado a la idea de flexibilidad, la mayoría de los trabajos sobre reflexión se han realizado en lenguajes **no tipados**, o con **tipado dinámico**. En dichos lenguajes no se puede asegurar que las operaciones se completen satisfactoriamente, pudiendo existir errores de tipo en tiempo de ejecución.

Si se desea ceder flexibilidad para obtener la seguridad de un lenguaje con **tipado estático**, se deben restringir los tipos de operaciones reflexivas de manera que solamente acepten valores que el sistema de tipos puede garantizar. De esta forma se restringen algunos usos válidos de reflexión, como la inclusión de valores obtenidos por entradas en tiempo de ejecución.

### 2.3.4. Por Capacidad de Reificación

Con respecto a la capacidad de reificación se pueden distinguir entre dos tipos de reflexión, que no son mutuamente excluyentes, llamados de **comportamiento** y **estructural**. La primera refiere al proceso de ejecución de los programas, mientras que la segunda tiene que ver con sus aspectos estructurales. Sus definiciones son las siguientes [MJD96]:

> ***Reflexión de Comportamiento:*** *tiene que ver con la habilidad de un lenguaje de proveer una reificación completa de su propia semántica e implementación (procesador), así como de los datos e implementación del sistema de ejecución en el que se ejecuta.*
>
> ***Reflexión Estructural:*** *tiene que ver con la habilidad de un lenguaje de proveer una reificación completa tanto del programa que se está ejecutando en ese momento como de sus tipos de datos abstractos.*

La reflexión de comportamiento, impulsada por Smith en 3-Lisp, es la más potente y difícil de implementar, dado que permite que el comportamiento subyacente del sistema sea cambiado. Este tipo de sistemas es utilizado en ambientes en los que se da una alta prioridad a la flexibilidad y ha alcanzado cierto desarrollo en lenguajes orientados objetos.

A la reflexión estructural también se la llama reflexión lingüística, porque está asociada con la habilidad de un programa en funcionamiento de generar nuevos fragmentos de programa e integrarlos dentro de su propia ejecución [SMKC93].

## 2.4. Estructuras Internas

Uno interpreta el mundo en base a los conceptos y categorías de una determinada teoría. De esta misma forma se debe realizar la interpretación de uno mismo. Al decir de Smith [Smi82] *"cuando reflexionas sobre tu propio comportamiento, lo debes hacer inevitablemente de una manera relativa a una teoría algo arbitraria"*. Tanto el procesador como las estructuras internas son abstracciones que nos permiten describir el comportamiento de un proceso, por lo que no nos importa su implementación interna, sino que se debe poder proveer una interfaz que nos permita razonar sobre ella.

En este trabajo se considerarán estructuras desde una **perspectiva denotacional**, es decir, en base a las construcciones que se utilizan al definir la semántica denotacional de un lenguaje [Sch86], dado que de esta forma se puede brindar una interpretación matemática de los significados de los programas. Se define una función de significado que mapea frases del lenguaje (código) en sus denotaciones, las cuales se definen en base a ambientes, memorias y continuaciones. Existen otras perspectivas posibles para contemplar las estructuras internas, como ser: sintáctica, de la máquina y del intérprete. Por más detalles sobre las distintas perspectivas ver [Ala04, sección 4.2].

A continuación se presentarán ejemplos de implementación de mecanismos reflexivos en lenguajes funcionales para la reflexión de continuaciones, ambientes y código. Este trabajo profundizará en reflexión de código para lenguajes funcionales tipados, que es el tema que se tratará en el siguiente capítulo.

### 2.4.1. Continuaciones

Las continuaciones son una forma de representar el estado de la ejecución de un programa en un determinado momento, en otras palabras, una continuación es una abstracción

del resto del programa. Las continuaciones de primera clase son una abstracción que evolucionó de algunas estructuras de control no estándares de lenguajes de fines de los 60, como el operador `J` de Landin [Lan65] o el `escape` de Reynolds [Rey72].

Una forma de manipular continuaciones es utilizando las primitivas `catch` y `throw` de Lisp. Con la primera se captura el contexto, asignándolo a una etiqueta, y con la segunda se abandona el contexto actual y se vuelve al anterior. Por lo tanto, `catch` puede ser vista como una primitiva de reificación de continuaciones y `throw` como una de absorción.

Originalmente, Scheme [SS75] disponía de éstas construcciones para la manipulación de continuaciones, pero en 1982 Clinger y Friedman [CFW85] introdujeron una forma de reificar una continuación como un objeto de primera clase, mediante el procedimiento `call-with-current-continuation` (`call/cc`) que reifica la continuación actual como un procedimiento. La absorción se realiza invocando el procedimiento reificado.

Existen lenguajes tipados que proveen de mecanismos similares al anterior. En el Standard ML de New Jersey [AM91] se pueden manipular continuaciones tipadas de primera clase [DHM91] utilizando las siguientes funciones primitivas:

- `callcc` : $\forall\alpha.(\alpha$ `cont` $\rightarrow \alpha) \rightarrow \alpha$
- `throw` : $\forall\alpha\beta.(\alpha$ `cont`$) \rightarrow (\alpha \rightarrow \beta)$

Donde, por ejemplo, el tipo `int cont` es el tipo de una continuación que espera un valor entero. La función `callcc` es similar al `call/cc` de Scheme, mientras que la función `throw` absorve la continuación en la función que se le pasa como parámetro.

Es claro que contar solamente con esta clase de mecanismos resulta muy restrictivo y no hace a un lenguaje reflexivo, ya que no se provee ni análisis ni síntesis sobre las continuaciones. Marc Feeley [Fee01] propone una extensión a Scheme en la cual se representan las continuaciones como tipos de datos abstractos, que contiene como núcleo el siguiente conjunto de primitivas:

- `(continuation-capture r)`: crea un objeto que representa a la continuación actual e invoca al procedimiento `r` con la continuación como parámetro.
- `(continuation-graft c t)`: llama al procedimiento `t` sin parámetros y con la continuación implícita `c`.
- `(continuation-return c v1 ... vn)`: retorna los valores (`v1 ... vn`) a la continuación `c`.

Esta idea brinda mayor flexibilidad en su manipulación, permitiendo además que se introduzcan primitivas introspectivas, como la operación `(continuation-next c)`, que avanza un lugar en el programa apuntado por `c`.

### 2.4.2. Ambientes

El ambiente mapea las variables libres de una expresión a sus valores asociados. Por lo tanto al reificar un ambiente se tiene acceso a las ligaduras de las variables activas en ese punto, y al absorverlo el código se evalúa con sus ligaduras.

En 3-Lisp el ambiente es uno de los parámetros de los procedimientos reflexivos, y se puede manipular como una lista. En el lenguaje Brown [FW84] los ambientes reificados se representan como funciones que reciben un identificador y devuelven un valor.

Rascal [Jag92, Jag94] es un lenguaje que permite la reificación y absorción de ambientes, y provee de mecanismos para restringir el grado y alcance de estos procesos. El modelo

utilizado para brindar estos mecanismos se basa en partir el ambiente en dos componentes: el **ambiente léxico** y el **ambiente público**. El primero contiene todas las ligaduras léxicas y el segundo contiene las ligaduras que se pueden exportar fuera del ambiente léxico en el cual se crearon. En base a esta partición se definen las siguientes operaciones para manipular ambientes:

1. *Copiar* una ligadura del ambiente léxico al ambiente público. Se realiza con la operación `(make-public l e)`, que toma una lista de variables léxicas `l` y una expresión `e`, y evalúa `e` en el ambiente léxico actual y un nuevo ambiente público resultante de agregarle al actual las ligaduras de todas las variables en `l`.

2. *Utilizar* el valor asociado de una ligadura del ambiente público (`use x`).

3. *Recortar* o quitar ligaduras del ambiente público. Dada una expresión `e` que retorna un objeto que representa a un ambiente, `(barrier e)` le quita todas las ligaduras definidas en el ambiente público actual. Otra forma de recortar ambientes es mediante la operación `(restrict e l)`, que en lugar de quitar todas las ligaduras del ambiente público actual quita sólo las de la lista `l`.

4. *Reificar* las ligaduras del ambiente público a un objeto de datos (`reify`).

5. *Absorver* un objeto de datos en un ambiente público reemplazando las ligaduras definidas en el actual. `(reflect e1 e2)` evalúa `e2` en el ambiente léxico actual y un nuevo ambiente público resultante de reemplazar en el actual las ligaduras del ambiente reificado en la expresión `e1`.

Como se puede ver Rascal ofrece operaciones de reificación y absorción, pero sus funcionalidades de análisis y síntesis son algo limitadas.

En la propuesta de Queinnec y DeRoure [QD96] se ofrecen algunas características reflexivas de Rascal sin perjudicar la eficiencia en la compilación.

La reificación se realiza con la operación `(export l)` que devuelve un ambiente de primera clase con las variables nombradas en la lista `l`, o con todas las del ambiente léxico si la lista es vacía. La absorción se hace con la función `(import l e1 e2)`. Se evalúa primero `e1` para obtener un ambiente reificado, luego `e2` se evalúa con el ambiente resultante de extender el actual con las ligaduras del ambiente reificado para las variables que se mencionan en `l` (todas por defecto). El lenguaje provee operaciones que permiten realizar análisis sobre los ambientes reificados; éstas son `environment-present?` y `environment-get`. La primera sirve para determinar si un nombre se encuentra o no en el ambiente reificado, y la segunda permite obtener el valor asignado a la variable. Existen también dos operaciones de síntesis, una que permite renombrar ligaduras y la otra, extender ambientes. La operación `environment-rename` implementa la funcionalidad de renombrado. La extensión del ambiente se realiza con la operación `environment-enrich`, que recibe el nombre de una variable y un ambiente, y retorna un nuevo ambiente extendido con una ligadura asociando al nombre a un valor no inicializado.

El cálculo $\lambda\varepsilon$ [SSB01] es una extensión de un cálculo lambda tipado simple que agrega primitivas para la manipulación de ambientes de primera clase. Se definen los *tipos de ambientes* ($E$), que tienen la forma $\{x_1^{\tau_1}, ..., x_n^{\tau_n}\}$, indicando que un ambiente de ese tipo tiene las ligaduras de las variables $x_1, ..., x_n$ con tipos $\tau_1, ..., \tau_n$ respectivamente.

Se pueden crear nuevos ambientes (síntesis) utilizando la sintaxis $\{e_1/x_1^{\tau_1}, ..., e_n/x_n^{\tau_n}\}$, que genera un ambiente que liga cada variable ($x_i^{\tau_i}$) con la expresión correspondiente ($e_i$). Los nuevos ambientes pueden ser absorvidos mediante la operación de evaluación $a[\![e]\!]$, que evalúa el término $e$ en el ambiente $a$. No existen operaciones de análisis ni reificación de ambientes.

### 2.4.3. Código

Los programas en Lisp [McC62] pueden manipular código fuente como una estructura de datos. A este mecanismo se lo denomina *quotation*, que significa **citado**, en el sentido de la mención de las palabras dichas y escritas por alguien. Un programa, así como cualquier otro dato, se representa utilizando **expresiones simbólicas** (S-expresiones), las cuales se definen como símbolos atómicos (números o identificadores) y, recursivamente, listas de expresiones simbólicas.

Las S-expresiones le dan una estructura interna a la representación de programas, pero no aseguran la validez de la sintaxis ni del tipado de los programas objeto que representan. Para que represente a un programa sintácticamente válido, una S-expresión debe ser un átomo o una lista que contenga como primer elemento una función y como resto sus parámetros. De esta forma, al evaluarse un programa en Lisp, los átomos evalúan a sí mismos y las listas, llamando a la función representada por el primer elemento, con los siguientes elementos como parámetros.

La función `quote` (también se puede escribir con el caracter `'`) evita la evaluación de su argumento; es decir que la lista que representa al código es considerada un átomo para el intérprete del lenguaje. Por ejemplo, la expresión `(+ 4 5)` evalúa al número 9, pero si evaluamos `(quote (+ 4 5))` el resultado es la lista con los elementos `+`, `4` y `5`. De esta forma se permite que el programador pueda consultar y manipular el código fuente, teniendo entonces una operación de reificación de código.

La absorción del código se realiza mediante su evaluación. La función `eval` evalúa la expresión reificada que se le pase como parámetro. Por lo tanto evaluar la expresión `(eval (quote (+ 4 5)))` resulta en el número 9.

Dado que sólo el código es absorvido, su evaluación toma el ambiente y la continuación del punto en el que es llamada, permitiéndose que las ligaduras locales sean visibles para el código evaluado. Existen otros enfoques en los cuales a la evaluación se le pasa también un ambiente reificado. Otra posibilidad es que en la evaluación se absorva, junto a las demás estructuras, la continuación. Se tiene entonces una función de evaluación explícita completa, como `normalise` de 3-Lisp [Smi84] y `meaning` de Brown [FW84], que usualmente se acompaña de una operación de reificación que también abarca a todas las estructuras. En este trabajo nos inclinaremos por tratar la reificación y absorción de las estructuras por separado y no en una sola operación.

Se pueden construir expresiones reificadas a partir de otras utilizando ***quasi-quotation***. El término *quasi-quote* proviene de la lógica formal, y fue acuñado por Quine a principios de la década de los 40 como una forma de distinguir entre el significado denotado por una sintaxis y la sintaxis misma. Se basa en permitir escapes temporales del contexto citado pasando al lenguaje normal, en el cual se puede hablar sobre la sintaxis [Bar06]. Esta idea fue trasladada a los lenguajes de programación para definir al mecanismo que permite insertar "huecos" dentro de un código citado los cuales se "llenan" con el resultado de la evaluación de expresiones no citadas. Según Alan Bawden [Baw99] a mediados de 1970, sin estar construida en ningún dialecto de Lisp, muchos programadores utilizaban versiones personales de *quasi-quotes*. En *The evolution of Lisp* [SG93] se menciona a este concepto como *pseudo-quoting*, y se indica que el mismo fue estandarizado en el proyecto *Lisp Machine* [GKH$^+$84] del MIT.

Un caracter *back-quote* (`` ` ``) comienza una *quasi-quotation*, y una coma (`,`), dentro de una *quasi-quotation* y precediendo a una expresión, indica que esta expresión no será tratada como una cita pero que el resultado de su evaluación será insertado como una pieza de código en la *quasi-quotation* que la contiene.

Entonces, si queremos generar la expresión `(+ 4 i)`, donde `i` es un parámetro. Se

puede construir utilizando las operaciones de lista `(list '+ '4 i)` o con *quasi-quotation* `` `(+ 4 ,i) ``.

El siguiente es un ejemplo estándar de especialización de código. Utilizando la notación de coma y *back-quote*, se generan funciones de potencia especializadas con respecto a sus exponentes.

```
(define (pow n)
    `(lambda (x)
        ,(letrec ((pow-rec
            (lambda (n)
                (if (= n 0)
                    `1
                    `(* x ,(pow-rec (-n 1)))))))
          (pow-rec n))))
```

Evaluar este programa con `n` igual a dos, genera la siguiente S-expresión:
`(lambda (x) (* x (* x 1)))`.

Dado que las S-expresiones son listas, los objetos reificados pueden ser inspeccionados y mutados utilizando las primitivas usuales de listas; algunas de las operaciones provistas por Lisp son:

- `(cons x1 x2)`: construye una S-expresión compuesta a partir de las dos S-expresiones pasadas como parámetro.
- `(car x)`: devuelve la primera parte de la S-expression compuesta `x`.
- `(cdr x)`: devuelve la segunda parte de la S-expression compuesta `x`.
- `(null x)`: es verdadero si el parámetro es NIL.
- `(atom x)`: es verdadero si el parámetro es atómico.
- `(eq x1 x2)`: verifica la igualdad sobre símbolos atómicos.
- `(list x1 ... xn)`: construye la lista `(x1 ... xn)`.

Las construcciones presentadas en esta sección pertenecen a lenguajes no tipados. Dado que el objetivo de este trabajo es el estudio de mecanismos de reflexión de código en lenguajes funcionales tipados, se dedicará todo el siguiente capítulo a analizar lenguajes de este tipo que provean primitivas de reflexión de código, evaluando la capacidad de reflexión que brinda cada uno y la seguridad que provee su tipado.

# Capítulo 3

# Reflexión de Código en Lenguajes Funcionales Tipados

*Por el primer terceto voy entrando,*
*y parece que entré con pie derecho*
*pues fin con este verso le voy dando.*

*Ya estoy en el segundo y aún sospecho*
*que voy los trece versos acabando:*
*contad si son catorce y está hecho.*

Primer y segundo terceto del "Soneto de Repente" de Lope de Vega.

Las construcciones de los mecanismos de citado de código vistas en el capítulo anterior funcionan como anotaciones de etapas, dado que definen una división entre etapas de computación. Los resultados de etapas anteriores son código de las siguientes etapas. Esto significa que un programa puede generar una pieza de código y luego ejecutarla, y ese código puede a su vez generar más código y ejecutarlo, y así sucesivamente. Este tipo de metaprogramación, que permiten la generación de código en tiempo de ejecución, ha definido un paradigma llamado **programación multi-etapas** [Tah99], que ha evolucionado con el desarrollo de lenguajes como MetaML y MetaOCaml. Estos lenguajes deben ser homogéneos, en el sentido de que el meta-lenguaje debe ser el mismo que el lenguaje objeto, de manera de poder proveer múltiples niveles y construcciones de ejecución de código [She01].

Las primeras investigaciones formales sobre lenguajes para computación multi-etapas fueron realizadas por Davies y Pfenning [DP96, Dav96], que estudiaron dos lenguajes multi-etapas tipados basados en lógicas modales. La teoría de estos lenguajes ha tenido un desarrollo importante a partir del trabajo de doctorado de Taha [Tah99] y la implementación del lenguaje MetaML.

En este capítulo se examinarán varios lenguajes funcionales multi-etapas y se compararán sus características, analizando la capacidad de los mecanismos reflexivos que brinda cada uno.

## 3.1. Basados en Lógica Modal: $\lambda^{\Box}$ y $\lambda^{\bigcirc}$

Estas dos primeras propuestas están inspirados en la lógica modal, y se diferencian en que para el primero el código que se trata debe ser cerrado, mientras que el segundo

maneja siempre código abierto.

El cálculo $\lambda^{\Box}$ [DP96] es una interpretación lógica de la computación por etapas que se basa en la lógica modal S4 [HC96]. Existen dos formulaciones de $\lambda^{\Box}$ que son equivalentes entre sí: una explícita $\lambda^{\Box}_e$, donde el uso de código se debe introducir explícitamente y ligarse a un nombre, y otra implícita $\lambda^{\Box}_i$ donde la construcción de código puede ser colocada en el lugar donde se va a utilizar.

### 3.1.1. Formulación Explícita de $\lambda^{\Box}$

| | | | |
|---|---|---|---|
| Tipos Base | $b \in \mathsf{B}$ | ::= | $\texttt{nat}$ |
| Tipos | $\tau \in \mathsf{T}$ | ::= | $b \mid \tau \to \tau \mid \Box\tau$ |
| Contextos | $\Gamma, \Delta \in \mathsf{G}$ | ::= | $\cdot \mid \Gamma, x : \tau$ |
| Términos | $e \in \mathsf{E}$ | ::= | $\texttt{z} \mid \texttt{s}\ e \mid x \mid \lambda x : \tau.\ e \mid \texttt{fix}\ x : \tau.\ e \mid e_1\ e_2 \mid$ |
| | | | $\texttt{case}\ e_1\ \texttt{of z} \Rightarrow e_2 \mid \texttt{s}\ x \Rightarrow e_3 \mid$ |
| | | | $\texttt{box}\ e \mid \texttt{let box}\ x = e_1\ \texttt{in}\ e_2$ |

Figura 3.1: Sintaxis del Lenguaje $\lambda^{\Box}_e$

En este lenguaje aparece el nuevo constructor de tipo $\Box\tau$, el cual representa a un *código de tipo* $\tau$. Para poder manipular una expresión como código se define el nuevo constructor $\texttt{box}\ e$, que toma una expresión cerrada $e$ de tipo $\tau$ y resulta en un trozo de código de tipo $\Box\tau$. Los trozos de código cerrado se pueden mezclar para generar nuevos trozos de código también cerrado, mediante la aplicación de una sustitución en las variables "libres" de un trozo de código por códigos cerrados. Esto se puede asegurar utilizando dos contextos separados en el juicio de tipado $\Delta; \Gamma \vdash_e e : \tau$, donde $\Gamma$ es el contexto usual y $\Delta$ es el contexto modal, que contiene variables que serán ligadas sólo a código durante la evaluación, permitiéndose que sólo aparezcan libres dentro de una expresión `box` variables ligadas en $\Delta$. Se pueden introducir variables en el contexto modal mediante la operación de eliminación de `box`, la cual es `let box`. La evaluación de un trozo de código, por ejemplo, se puede escribir de la siguiente forma:

$$\lambda x : \Box\tau.\ \texttt{let box}\ y = x\ \texttt{in}\ y : \Box\tau \to \tau$$

Las reglas de tipado del fragmento modal del lenguaje descrito se encuentran en la Figura 3.2. Las reglas del fragmento del cálculo lambda son las usuales si ignoramos al contexto $\Delta$.

En la regla Box, se asegura que el constructor `box` se aplique sólo a términos cerrados obligando que el contexto $\Gamma$ sea vacío. La regla LetBox, de eliminación de $\Box$, es la única en la cual se agregan variables al contexto $\Delta$.

La semántica operacional para el fragmento modal se describe en la Figura 3.3. En la evaluación de $\texttt{box}\ e$ la expresión $e$ no es evaluada, de manera de ser manipulada como código. Al evaluarse una expresión $\texttt{let box}\ x = e_1\ \texttt{in}\ e_2$, todas las apariciones de la variable $x$ en $e_2$ son sustituídas por el código $e_1'$ resultante de evaluar $e_1$.

Con los elementos descritos se puede implementar una versión de la función `pow` del capítulo anterior en $\lambda^{\Box}_e$. Esta función tiene tipo $\texttt{nat} \to \Box(\texttt{nat} \to \texttt{nat})$, es decir que toma un entero y devuelve una expresión citada, que es una función de entero a entero.

$$\frac{x : \tau \in \Delta}{\Delta; \Gamma \vdash_e x : \tau}\ \text{GVar} \qquad \frac{\Delta; \cdot \vdash_e e : \tau}{\Delta; \Gamma \vdash_e \texttt{box}\ e : \Box\tau}\ \text{Box}$$

$$\frac{\Delta; \Gamma \vdash_e e_1 : \Box\tau_1 \quad (\Delta, x : \tau_1); \Gamma \vdash_e e_2 : \tau_2}{\Delta; \Gamma \vdash_e \texttt{let box}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{LetBox}$$

Figura 3.2: Reglas de Tipado del Fragmento Modal del Lenguaje $\lambda_e^{\Box}$

$$\frac{}{\texttt{box}\ e \hookrightarrow \texttt{box}\ e} \qquad \frac{e_1 \hookrightarrow \texttt{box}\ e_1' \quad e_2[x/e_1'] \hookrightarrow v_2}{\texttt{let box}\ x = e_1\ \texttt{in}\ e_2 \hookrightarrow v_2}$$

Figura 3.3: Reglas de Evaluación del Fragmento Modal del Lenguaje $\lambda_e^{\Box}$

```
pow = fix p: nat → □(nat → nat).
        λn: nat.
            case n of
                z  ⇒box (λx: nat. s z)
              | s m⇒let box q = p m in box (λx: nat. x × (q x))
```

Aplicada al natural dos (`s s z`) resulta en el trozo de código:

```
box (λx: nat.x × (λx: nat.x × (λx: nat. s z) x) x).
```

El código generado es menos eficiente que el del ejemplo de Lisp, ya que aparecen betas no reducidos. Esto se debe a que en $\lambda_e^{\Box}$ sólo se puede manipular código cerrado.

### 3.1.2. Formulación Implícita de $\lambda^{\Box}$

La formulación implícita tiene como ventaja sobre la anterior, que la definición de etapas de la computación suele requerir que sólo se tengan que agregar o quitar constructores modales al código, mientras que en el otro enfoque la estructura del código debe acompañar a la definición de etapas. Esto se debe a que en la formulación explícita sólo se pueden utilizar resultados de etapas anteriores mediante el constructor `let box`, que liga las variables de código por fuera del código a construir. En la formulación implícita, cuya sintaxis se muestra en la Figura 3.4, esto se sustituye por un constructor `unbox` que permite utilizar un código (generado por `box`) en la etapa actual y un constructor `pop` que, aplicado reiteradamente, lleva a futuras etapas. Se crea así una analogía con el *quasi-quotation* de Lisp, con la diferencia de que aquí se trata solamente con código cerrado, donde `box` corresponde a (‘) y `unbox (pop` $e$`)` corresponde a (`,`) para alguna expresión $e$.

Entonces, el ejemplo de la función generadora de potencias se puede escribir de la siguiente manera, dejando la computación de código en el lugar.

| | | | |
|---|---|---|---|
| Tipos Base | $b \in \mathsf{B}$ | ::= | $\mathtt{nat}$ |
| Tipos | $\tau \in \mathsf{T}$ | ::= | $b \mid \tau \to \tau \mid \Box\tau$ |
| Contextos | $\Gamma \in \mathsf{G}$ | ::= | $\cdot \mid \Gamma, x : \tau$ |
| Pilas de Contextos | $\Psi \in \mathsf{P}$ | ::= | $\cdot \mid \Psi; \Gamma$ |
| Términos | $e \in \mathsf{E}$ | ::= | $\mathtt{z} \mid \mathtt{s}\ e \mid x \mid \lambda x : \tau.\ e \mid \mathtt{fix}\ x : \tau.\ e \mid e_1\ e_2 \mid$ $\mathtt{case}\ e_1\ \mathtt{of}\ \mathtt{z} \Rightarrow e_2 \mid \mathtt{s}\ x \Rightarrow e_3 \mid$ $\mathtt{box}\ e \mid \mathtt{unbox}\ e \mid \mathtt{pop}\ e$ |

Figura 3.4: Sintaxis del Lenguaje $\lambda_i^{\Box}$

```
pow = fix p: nat → □(nat → nat).
        λn: nat.
            case n of
                z  ⇒ box (λx: nat. s z)
                | s m ⇒ box (λx: nat. x × ((unbox (pop (p m))) x))
```

Esta función es análoga a la definida en la subsección anterior.

Las reglas de tipado de este lenguaje utilizan una pila de contextos $\Psi$, donde cada elemento corresponde al contexto de una etapa de computación. El juicio tiene la forma $\Psi; \Gamma \vdash_i e : \tau$, que significa que el término $e$ tiene tipo $\tau$ en el contexto local $\Gamma$ bajo la pila $\Psi$.

$$\frac{\Psi; \Gamma; \cdot \vdash_i e : \tau}{\Psi; \Gamma \vdash_i \mathtt{box}\ e : \Box\tau}\ \text{Box} \qquad \frac{\Psi; \Gamma \vdash_i e : \Box\tau}{\Psi; \Gamma \vdash_i \mathtt{unbox}\ e : \tau}\ \text{UnBox}$$

$$\frac{\Psi; \Gamma_1 \vdash_i e : \Box\tau}{\Psi; \Gamma_1; \Gamma_2 \vdash_i \mathtt{pop}\ e : \Box\tau}\ \text{Pop}$$

Figura 3.5: Reglas de Tipado del Fragmento Modal del Lenguaje $\lambda_i^{\Box}$

Un constructor `box`, al congelar la evaluación, genera una nueva etapa, agregando un nuevo contexto vacío para el cuerpo del constructor. En cada etapa sólo se puede acceder a las variables del contexto actual, aunque el código generado en esta etapa o en etapas anteriores puede ser utilizado a través de las reglas UnBox y Pop.

En [DP96] no se define una semántica operacional para $\lambda_i^{\Box}$ directamente, sino que la misma depende de una traducción a $\lambda_e^{\Box}$. Esta traducción consiste básicamente en asociar los términos dentro de $n$ constructores `pop` anidados a nuevas variables, las cuales se ligan con un `let box` fuera del $n$-ésimo constructor `box`.

### 3.1.3. En base a Lógica Temporal: $\lambda^{\bigcirc}$

El cálculo $\lambda^{\bigcirc}$ [Dav96] se motiva por la modalidad de la lógica temporal $\bigcirc$, haciendo una correspondencia entre la noción de "un tiempo particular" y una etapa de com-

putación. Se provee de construcciones que permiten la generación de código, sin la restricción de que sea cerrado. No hay una construcción para evaluar el código generado. En la sintaxis de la Figura 3.6 la etapa se anota en los contextos como un número natural $n$, es decir que $x : \tau^n$ indica que la variable $x$ de tipo $\tau$ fue ligada en la etapa $n$. El constructor `next` es análogo a `box` sin la restricción de que la expresión sea cerrada, mientras que `prev` permite pasar nuevamente el control a la etapa actual siendo su resultado un valor de la siguiente etapa.

| | | | |
|---|---|---|---|
| Tipos Base | $b \in \mathsf{B}$ | ::= | `nat` |
| Tipos | $\tau \in \mathsf{T}$ | ::= | $b \mid \tau \to \tau \mid \bigcirc\tau$ |
| Contextos | $\Gamma \in \mathsf{G}$ | ::= | $\cdot \mid \Gamma, x : \tau^n$ |
| Términos | $e \in \mathsf{E}$ | ::= | $\mathtt{z} \mid \mathtt{s}\ e \mid x \mid \lambda x : \tau.\ e \mid \mathtt{fix}\ x : \tau.\ e \mid e_1\ e_2 \mid$ |
| | | | $\mathtt{case}\ e_1\ \mathtt{of}\ \mathtt{z} \Rightarrow e_2 \mid \mathtt{s}\ x \Rightarrow e_3 \mid$ |
| | | | $\mathtt{next}\ e \mid \mathtt{prev}\ e$ |

Figura 3.6: Sintaxis del Lenguaje $\lambda^{\bigcirc}$

El juicio de tipado es de la forma $\Gamma \vdash^n e : \tau$, que significa que el término $e$ tiene tipo $\tau$ en la etapa $n$ en el contexto $\Gamma$. Cuando se agrega una variable al contexto se le asigna la etapa actual $n$, por lo tanto sólo las variables ligadas con índice $n$ pueden ser utilizadas en esa misma etapa.

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \mathtt{next}\ e : \bigcirc\tau}\ \text{Next} \qquad \frac{\Gamma \vdash^{n} e : \bigcirc\tau}{\Gamma \vdash^{n+1} \mathtt{prev}\ e : \tau}\ \text{Prev}$$

Figura 3.7: Reglas de Tipado del Fragmento Temporal del Lenguaje $\lambda^{\bigcirc}$

El juicio de evaluación es de la forma $e \overset{n}{\hookrightarrow} v$, que se lee "la expresión $e$ con tiempo $n$ evalúa al valor $v$". Por lo tanto, la expresión $e$ tiene que tipar en la etapa $n$. El conjunto de valores se divide en subconjuntos $v^i$ de acuerdo al tiempo $i$ en que ocurren. Los valores $v^0$ son los usuales más la expresión `next` $v$, mientras que los valores $v^{n+1}$ son todas las expresiones posibles que no tengan sub-expresiones con tiempo 0.

Las reglas de evaluación para las construcciones no temporales son las usuales en el caso que se encuentren en tiempo 0. Para tiempos mayores que 0 (Figura 3.8) se construyen trozos de código acordes a la estructura sintáctica de la expresión.

Para la evaluación de `next` $e$ se evalúa $e$ en la siguiente etapa, quedando casi todas las construcciones intactas salvo por las apariciones de sub-expresiones `prev` $e$. Una expresión `prev` $e$ en tiempo $i + 1$ provoca la evaluación de $e$ en tiempo $i$, generando un trozo de código.

Si escribimos el ejemplo de la potencia, en este caso se puede lograr generar un código más eficiente que en $\lambda^{\Box}$, debido a la capacidad del lenguaje de manipular código abierto. La función `pow` tiene tipo `nat` $\to \bigcirc$`(nat` $\to$ `nat)` en el nivel 0.

$$\frac{}{x \overset{n+1}{\hookrightarrow} x} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\lambda x : \tau \,.e \overset{n+1}{\hookrightarrow} \lambda x : \tau \,.v} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2}$$

Figura 3.8: Reglas de Evaluación del Fragmento No Temporal de $\lambda^{\bigcirc}$ para Tiempo n+1

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{next}\ e \overset{n}{\hookrightarrow} \texttt{next}\ v} \qquad \frac{e \overset{0}{\hookrightarrow} \texttt{next}\ v}{\texttt{prev}\ e \overset{1}{\hookrightarrow} v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{prev}\ e \overset{n+2}{\hookrightarrow} \texttt{prev}\ v}$$

Figura 3.9: Reglas de Evaluación del Fragmento Temporal del Lenguaje $\lambda^{\bigcirc}$

```
pow = λn: nat. (next λx: nat. (prev
        (fix p: nat → ◯nat.
            λn′: nat.
                case n′ of
                    z  ⇒ next (s z)
                    | s m ⇒ next (x × (prev (p m)))) n))
```

Si, por ejemplo, se aplica esta función al natural dos (`s s z`) se invoca una función recursiva que retorna el código abierto `next` $(x \times (x \times$ `(s z)`$))$, que luego se inserta al final de `next` $\lambda x$, generando como resultado el código:
`next` $\lambda x$`: nat.` $(x \times (x \times$ `(s z)`$))$.

## 3.2. MetaML

MetaML [SH94] es un lenguaje tipo SML con construcciones especiales para programación multi-etapas. En lugar de estar inspirado en un sistema lógico en particular, este lenguaje proviene de la práctica de programación y la necesidad de proveer de operadores de etapas seguros y fáciles de usar [Yan99]. Desciende del lenguaje de dos etapas CRML [HS93], el cual es fuertemente tipado y está inspirado en TRPL [She91]. Existen otros lenguajes con características similares a MetaML, como son su dialecto compilado MetaOCaml [Tah03] y Omega [She04].

MetaML provee de construcciones para manipular código abierto utilizando anotaciones de etapas, las cuales son los **paréntesis**[1] $\langle e \rangle$ y el **escape** $\sim e$, que corresponden a `next` $e$ y `prev` $e$ de $\lambda^{\bigcirc}$ respectivamente. A diferencia del cálculo anterior, MetaML provee persistencia entre etapas (*csp*[2]), que es una de las características que distinguen a MetaML, y se basa en permitir que las variables ligadas en una etapa puedan ser utilizadas en etapas posteriores; por ejemplo, la expresión $(\lambda x.\langle x \rangle)5$ reduce a $\langle 5 \rangle$. También se provee de un operador `run` para la ejecución del código generado.

[1] En inglés *brackets*.
[2] Del inglés *cross-stage persistence*.

$$
\begin{array}{lrcl}
\text{Tipos} & \tau \in \mathsf{T} & ::= & b \mid \tau \to \tau \mid \langle \tau \rangle \\
\text{Contextos} & \Gamma \in \mathsf{G} & ::= & \cdot \mid \Gamma, x : (\tau, r)^n \\
\text{Términos} & e \in \mathsf{E} & ::= & c \mid x \mid \lambda x.\ e \mid e_1\ e_2 \mid \\
 & & & \langle e \rangle \mid \tilde{}e \mid \texttt{run}\ e
\end{array}
$$

Figura 3.10: Sintaxis del Lenguaje MetaML

En la Figura 3.10 se muestra la sintaxis del núcleo del lenguaje. En el siguiente ejemplo escribimos la función generadora de potencias considerando una sintaxis extendida tipo SML.

```
-| val pow =
    let val rec pow' = fn n => fn x =>
                        if n=0 then <1> else <~x * ~(pow' (n-1) x)>)
    in
        fn n => <fn a => ~(pow' n <a>)>
    end;
val pow = fn : int -> <int -> int>

-| run <~(pow 2) 5>;
val it = 25 : int
```

Como se puede ver en el ejemplo, existe la posibilidad de ejecutar el código generado utilizando el operador `run`. Se debe cumplir la condición de que estáticamente se pueda asegurar que el código generado es cerrado.

Las reglas de evaluación para el cálculo básico son iguales a las de $\lambda^{\bigcirc}$, donde el concepto de tiempo se cambia por el de etapa de ejecución. Las reglas Br y Esc son también esencialmente las mismas que las de sus análogas de $\lambda^{\bigcirc}$, como se puede notar al comparar las Figuras 3.11 y 3.9. La evaluación de `run` $e$ en la etapa 0 provoca la evaluación de la expresión citada $v'$, siendo $\langle v' \rangle$ el resultado de evaluar $e$.

$$
\frac{e \overset{n+1}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{\sim e \overset{1}{\hookrightarrow} v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\sim e \overset{n+2}{\hookrightarrow} \sim v}
$$

$$
\frac{e \overset{0}{\hookrightarrow} \langle v' \rangle \qquad v' \overset{0}{\hookrightarrow} v}{\texttt{run}\ e \overset{0}{\hookrightarrow} v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{run}\ e \overset{n+1}{\hookrightarrow} \texttt{run}\ v}
$$

Figura 3.11: Reglas de Evaluación de un Fragmento de MetaML

La existencia del `run` y su interacción con $\sim$ complican el sistema de tipos, ya que no todo el código que se puede generar puede ser evaluado. Por ejemplo, no es trivial

determinar que el término $\langle \lambda x.\ \sim (\texttt{run}\ \langle x \rangle) \rangle$, que reduce a $\langle \lambda x.\ \sim x \rangle$, es incorrecto. En el sistema de tipos de la Figura 3.12, presentado por Taha *et al.* [TBS98], se encara el problema de asegurar la clausura de los fragmentos de código a evaluar mediante un nuevo índice que cuenta el número de operadores `run` que rodean a una expresión, de manera que cada variable permitida sea rodeada estrictamente por más paréntesis que evaluaciones de código. El juicio de tipado es entonces de la forma: $\Gamma \vdash^n e : \tau, r$, indicando que en el contexto $\Gamma$, en el nivel $n$ y rodeado sintácticamente por $r$ ocurrencias de `run`, el término $e$ tiene tipo $\tau$. El nivel de un término es la resta del número de paréntesis que lo rodea menos los escapes.

$$\frac{\Gamma, x : (\tau_1, r)^n \vdash^n e : \tau_2, r}{\Gamma \vdash^n \lambda x.e : (\tau_1 \to \tau_2), r}\ \text{Abs} \qquad \frac{x : (\tau, r')^{n'} \in \Gamma \quad n' + r \leq n + r'}{\Gamma \vdash^n x : \tau, r}\ \text{Var}$$

$$\frac{\Gamma \vdash^{n+1} e : \tau, r}{\Gamma \vdash^n \langle e \rangle : \langle \tau \rangle, r}\ \text{Br} \qquad \frac{\Gamma \vdash^n e : \langle \tau \rangle, r}{\Gamma \vdash^{n+1} \sim e : \tau, r}\ \text{Esc} \qquad \frac{\Gamma \vdash^n e : \langle \tau \rangle, r+1}{\Gamma \vdash^n \texttt{run}\ e : \tau, r}\ \text{Run}$$

Figura 3.12: Reglas de Tipado de un Fragmento de MetaML

Como se puede ver en la regla Run, si `run` $e$ está rodeado por $r$ ocurrencias, entonces $e$ se rodea por $r+1$. La condición $n'+r \leq n+r'$ de la regla Var asegura que cada ocurrencia de una variable está rodeada por más paréntesis que operadores `run`, y si la misma fue ligada en un nivel $n$ pueda ser utilizada en un nivel $n'$ igual o posterior a $n$. De esta forma el término $\langle \lambda x.\ \sim (\texttt{run}\ \langle x \rangle) \rangle$ no es tipable, dado que la siguiente secuencia de aplicación de las reglas es la única posible:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(n+1)+(r+1) \not\leq (n+1)+r \quad x : (\tau_1, r)^{n+1} \in \Gamma, x : (\tau_1, r)^{n+1}}{\Gamma, x : (\tau_1, r)^{n+1} \not\vdash^{n+1} x : \tau_2, r+1}\ \text{Var}}{\Gamma, x : (\tau_1, r)^{n+1} \not\vdash^n \langle x \rangle : \langle \tau_2 \rangle, r+1}\ \text{Br}}{\Gamma, x : (\tau_1, r)^{n+1} \not\vdash^n (\texttt{run}\ \langle x \rangle) : \langle \tau_2 \rangle, r}\ \text{Run}}{\Gamma, x : (\tau_1, r)^{n+1} \not\vdash^{n+1} \sim (\texttt{run}\ \langle x \rangle) : \tau_2, r}\ \text{Esc}}{\Gamma \not\vdash^{n+1} \lambda x. \sim (\texttt{run}\ \langle x \rangle) : (\tau_1 \to \tau_2), r}\ \text{Abs}}{\Gamma \not\vdash^n \langle \lambda x. \sim (\texttt{run}\ \langle x \rangle) \rangle : \langle (\tau_1 \to \tau_2) \rangle, r}\ \text{Br}$$

Sin embargo existen también programas que aparentemente serían correctos pero que no son aceptados, limitando seriamente la expresividad del lenguaje. Por ejemplo, dado que `run` no puede ocurrir dentro de una lambda-abstracción, a términos como $(\lambda x.\texttt{run}\ x)\ \langle 1 \rangle$ no se les puede asignar tipo en este sistema.

### 3.2.1. AIM

El cálculo AIM[3] [MTBS99] encara el problema anterior de otra forma, intentando combinar las características de los sistemas $\lambda^{\Box}$ y $\lambda^{\bigcirc}$ además de pasar parte de la responsabilidad de verificar que un código es cerrado al programador.

[3] Del inglés *An Idealized MetaML*.

La sintaxis del lenguaje se muestra en la Figura 3.13. Se agrega a los tipos de MetaML un nuevo tipo $[\tau]$ (*box de* $\tau$), equivalente a $\Box\tau$ de $\lambda^{\Box}$, que representa a un trozo de código cerrado. Se agregan también las construcciones `box-with` y `unbox`, que no pertenecen a MetaML y están motivadas en las construcciones de $\lambda^{\Box}$.

$$
\begin{array}{llll}
\text{Tipos} & \tau \in \mathsf{T} & ::= & b \mid \tau \to \tau \mid \langle \tau \rangle \mid [\tau] \\
\text{Contextos} & \Gamma \in \mathsf{G} & ::= & \cdot \mid \Gamma, x : \tau^n \\
\text{Términos} & e \in \mathsf{E} & ::= & c \mid x \mid \lambda x.\ e \mid e_1\ e_2 \mid \\
 & & & \langle e \rangle \mid \tilde{}e \mid \texttt{run}\ e\ \texttt{with}\ \{x_i = e_i\ \texttt{|}\ i \in m\} \mid \\
 & & & \texttt{box}\ e\ \texttt{with}\ \{x_i = e_i\ \texttt{|}\ i \in m\} \mid \texttt{unbox}\ e
\end{array}
$$

Figura 3.13: Sintaxis del Lenguaje AIM

La construcción `run-with` es una generalización del `run` de MetaML que permite el uso de variables adicionales en el cuerpo de la expresión a evaluar. En su regla de tipado se incrementa el nivel de todas las variables del contexto, evitando así la notación extra para contar el número de `run`s que rodean al término[4].

De las reglas de tipado de la Figura 3.14 se puede deducir que se mantiene la persistencia entre etapas, la cual se asegura mediante la condición $n' \leq n$ de la regla Var. La regla BoxWith asegura que un código es cerrado si se verifica que todas las ligaduras en la cláusula `with` son trozos de código cerrado y que el término en el cuerpo del `box-with` es tipable en el nivel 0. En UnBox, al cambiarse el tipo de $[\tau]$ a $\tau$, la expresión deja de considerarse como código cerrado.

$$
\frac{x : \tau^{n'} \in \Gamma \quad n' \leq n}{\Gamma \vdash^n x : \tau}\ \text{Var}
\qquad
\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^n \langle e \rangle : \langle \tau \rangle}\ \text{Br}
\qquad
\frac{\Gamma \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^{n+1} {\sim} e : \tau}\ \text{Esc}
\qquad
\frac{\Gamma \vdash^n e : [\tau]}{\Gamma \vdash^n \texttt{unbox}\ e : \tau}\ \text{UnBox}
$$

$$
\frac{\Gamma \vdash^n e_i : [\tau_i] \quad \Gamma^{+1}, \{x_i : [\tau_i]^n | i \in m\} \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^n \texttt{run}\ e\ \texttt{with}\ x_i = e_i : \tau}\ \text{RunWith}
\qquad
\frac{\Gamma \vdash^n e_i : [\tau_i] \quad \{x_i : [\tau_i]^0 | i \in m\} \vdash^0 e : \tau}{\Gamma \vdash^n \texttt{box}\ e\ \texttt{with}\ x_i = e_i : [\tau]}\ \text{BoxWith}
$$

Figura 3.14: Reglas de Tipado de un Fragmento de AIM

La evaluación de las nuevas construcciones se muestra en la Figura 3.15. Al evaluarse una expresión $\texttt{box}\ e\ \texttt{with}\ x_i = e_i$ en la etapa 0, la expresión $e$ no se evalúa, sólo se sustituyen las variables $x_i$ por el resultado de la evaluación de cada $e_i$. Por lo tanto, además de los valores usuales y las expresiones entre paréntesis, las expresiones `box` pueden ser valores en esta etapa. Se consideran como resultados aceptables de una computación en la etapa 0 tanto a los términos que representan código cerrado como a los que representan código abierto. En la evaluación de $\texttt{unbox}\ e$, dado que $e$ resulta en un código cerrado

[4]En la regla se escribe $\Gamma^{+1}$ para representar el incremento de nivel en el contexto $\Gamma$.

`box` $e'$, se debe evaluar la expresión $e'$. En `run-with` se evalúa el código resultante de evaluar la expresión luego de realizar las sustituciones de variables. Para que la expresión a evaluar se encuentre en el nivel 0 se le aplica "degradación", que se anota $e \downarrow_n$ e indica que se baja el nivel de $e$ de $n+1$ a $n$.

Como en los casos de las secciones anteriores, para etapas mayores que cero se construye un trozo de código que tiene la estructura de la expresión, representándose así la postergación de su evaluación.

$$\frac{e_i \overset{0}{\hookrightarrow} v_i}{\texttt{box}\ e\ \texttt{with}\ x_i = e_i \overset{0}{\hookrightarrow} \texttt{box}\ e[v_i/x_i]} \qquad \frac{e_i \overset{n+1}{\hookrightarrow} v_i}{\texttt{box}\ e\ \texttt{with}\ x_i = e_i \overset{n+1}{\hookrightarrow} \texttt{box}\ e\ \texttt{with}\ x_i = v_i}$$

$$\frac{e \overset{0}{\hookrightarrow} \texttt{box}\ e' \qquad e' \overset{0}{\hookrightarrow} v}{\texttt{unbox}\ e \overset{0}{\hookrightarrow} v} \qquad \frac{e_i \overset{0}{\hookrightarrow} v_i \qquad e[v_i/x_i] \overset{0}{\hookrightarrow} \langle v' \rangle \qquad v' \downarrow_0 \overset{0}{\hookrightarrow} v}{\texttt{run}\ e\ \texttt{with}\ x_i = e_i \overset{0}{\hookrightarrow} v}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{unbox}\ e \overset{n+1}{\hookrightarrow} \texttt{unbox}\ v} \qquad \frac{e_i \overset{n+1}{\hookrightarrow} v_i \qquad e \overset{n+1}{\hookrightarrow} v}{\texttt{run}\ e\ \texttt{with}\ x_i = e_i \overset{n+1}{\hookrightarrow} \texttt{run}\ v\ \texttt{with}\ x_i = v_i}$$

Figura 3.15: Reglas de Evaluación de un Fragmento de AIM

### 3.2.2. MetaML con Clausuras: $\lambda^{BN}$

El cálculo $\lambda^{BN}$ [Tah99] puede ser visto como una versión recortada de AIM, que lo simplifica manteniendo casi toda su expresividad. La sintaxis del lenguaje se muestra en la Figura 3.16. El constructor de tipo $[\tau]$ es una versión estricta del tipo *Box de* $\tau$ de AIM. Este tipo se propone sólo para asegurar que un término sea cerrado y no para posponer su evaluación, lo cual se realiza solamente por el tipo $\langle \tau \rangle$. Se evita así la idea de que hay dos tipos que representan código, que puede llevar a confusiones como que un valor de tipo $[\langle \tau \rangle]$ se considere "código cerrado de código abierto de $\tau$".

| | | | |
|---|---|---|---|
| Tipos | $\tau \in \mathsf{T}$ | ::= | $b \mid \tau \to \tau \mid \langle \tau \rangle \mid [\tau]$ |
| Contextos | $\Gamma \in \mathsf{G}$ | ::= | $\cdot \mid \Gamma, x : \tau^n$ |
| Términos | $e \in \mathsf{E}$ | ::= | $c \mid x \mid \lambda x.\ e \mid e_1\ e_2 \mid$ |
| | | | $\langle e \rangle \mid \tilde{}e \mid \texttt{close}\ e\ \texttt{with}\ \{x_i = e_i \mid i \in m\} \mid$ |
| | | | $\texttt{open}\ e \mid \texttt{safe\_run}\ e \mid \texttt{up}\ e$ |

Figura 3.16: Sintaxis del Lenguaje $\lambda^{BN}$

La evaluación de `close-with` se diferencia de `box-with` en que la evaluación de la

expresión a la cual se aplica no se pospone, como se puede ver en la Figura 3.17[5]. Por lo tanto, en el caso de `open` $e$, análogo a `unbox` de AIM, la expresión cerrada a la que evalúa $e$ ya está evaluada.

La operación `safe_run` es una versión de `run` que aprovecha la interacción entre los tipos cerrado y código, para lograr una ejecución de código segura (sólo código cerrado). En $\lambda^{BN}$ no se provee de persistencia entre etapas implícitamente, como se puede ver en la regla Var de la Figura 3.18, sino que la misma se realiza con la operación `up`, la cual se limita a valores cerrados.

$$\frac{e_i \overset{0}{\hookrightarrow} v_i \qquad e[v_i/x_i] \overset{0}{\hookrightarrow} v}{\texttt{close}\ e\ \texttt{with}\ x_i = e_i \overset{0}{\hookrightarrow} \texttt{close}\ v} \qquad \frac{e \overset{0}{\hookrightarrow} \texttt{close}\ v}{\texttt{open}\ e \overset{0}{\hookrightarrow} v}$$

$$\frac{e \overset{0}{\hookrightarrow} \texttt{close}\ \langle v' \rangle \qquad v' \overset{0}{\hookrightarrow} v}{\texttt{safe\_run}\ e \overset{0}{\hookrightarrow} \texttt{close}\ v} \qquad \frac{e \overset{0}{\hookrightarrow} \texttt{close}\ v'}{\texttt{up}\ e \overset{1}{\hookrightarrow} \texttt{close}\ v'}$$

Figura 3.17: Reglas de Evaluación de un Fragmento del Lenguaje $\lambda^{BN}$

Las reglas de tipado Br, Esc, CloseWith y Open son análogas, respectivamente, a Br, Esc, BoxWith y UnBox del sistema anterior (Figura 3.14). La regla para `safe_run` permite eliminar el tipo código cuando ocurre bajo un tipo cerrado. En la regla Var se permite sólo el uso de variables ligadas en el mismo nivel. Por eso la regla Up habilita la persistencia entre etapas explícita de valores cerrados, permitiendo subir un valor de tipo $[\tau]$ de un nivel a otro.

$$\frac{x : \tau^n \in \Gamma}{\Gamma \vdash^n x : \tau}\ \text{Var} \qquad \frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^n \langle e \rangle : \langle \tau \rangle}\ \text{Br} \qquad \frac{\Gamma \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^{n+1} {\sim} e : \tau}\ \text{Esc}$$

$$\frac{\begin{array}{c}\Gamma \vdash^n e_i : [\tau_i] \\ \{x_i : [\tau_i]^0 | i \in m\} \vdash^0 e : \tau\end{array}}{\Gamma \vdash^n \texttt{close}\ e\ \texttt{with}\ x_i = e_i : [\tau]}\ \text{CloseWith} \qquad \frac{\Gamma \vdash^n e : [\tau]}{\Gamma \vdash^n \texttt{open}\ e : \tau}\ \text{Open}$$

$$\frac{\Gamma \vdash^n e : [\langle \tau \rangle]}{\Gamma \vdash^n \texttt{safe\_run}\ e : [\tau]}\ \text{SRun} \qquad \frac{\Gamma \vdash^n e : [\tau]}{\Gamma \vdash^{n+1} \texttt{up}\ e : [\tau]}\ \text{Up}$$

Figura 3.18: Reglas de Tipado del Fragmento Staged del Lenguaje $\lambda^{BN}$

La versión en $\lambda^{BN}$ de la función potencia es similar a la implementada en MetaML,

[5]Se utilizará la abreviatura `close` $e$ cuando en `close` $e$ `with` $\{x_i = e_i \mid i \in m\}$ el conjunto definido por $\{x_i = e_i \mid i \in m\}$ sea vacío.

con el agregado de las anotaciones `close-with` para marcar un término como cerrado, y `open` para poder manipularlo como código.

```
-| val pow = close
    let val rec pow' = fn n => fn x =>
                        if n=0 then <1> else <~x * ~(pow' (n-1) x)>)
    in
        fn n => <fn a => ~(pow' n <a>)>
    end;
val pow = fn : [int -> <int -> int>]

-| run  close
            let val opow = open pow
            in <~(opow 2) 5>
            end
        with pow = pow;
val it = 25 : int
```

### 3.2.3. Clasificadores de Ambiente

Los Clasificadores de Ambiente fueron propuestos por Taha y Nielsen [TN03] como una nueva forma de tipar lenguajes multi-etapas, que permite que `run` evalúe código abierto y también pueda ser lambda-abstraído. Su principal característica es la de proveer un tipo código $\langle\tau\rangle^\alpha$ con un **clasificador** $\alpha$ que restringe las variables que pueden ocurrir libres. Los clasificadores anotan el nivel en el cual se declara cada variable y un código de tipo $\langle\tau\rangle^\alpha$ sólo puede contener variables libres declaradas en el nivel anotado con $\alpha$. En [CMT04] se observó que la inferencia de tipos para esta propuesta falla, por lo que se realizó una nueva propuesta que utiliza clasificadores implícitos en la cual se puede realizar inferencia. Este nuevo cálculo, cuya sintaxis se muestra en la Figura 3.19, se llama $\lambda^i_{let}$.

| | | | |
|---|---|---|---|
| Clasificadores | $\alpha \in \mathsf{A}$ | | |
| Niveles Nombrados | $A \in \mathsf{A}^*$ | | |
| Tipos | $\tau \in \mathsf{T}$ | $::=$ | $b \mid \tau \to \tau \mid \langle\tau\rangle^\alpha \mid \langle\tau\rangle$ |
| Contextos | $\Gamma \in \mathsf{G}$ | $::=$ | $\cdot \mid \Gamma, x : \tau^A$ |
| Términos | $e \in \mathsf{E}$ | $::=$ | $c \mid x \mid \lambda x.\ e \mid e_1\ e_2 \mid$ |
| | | | $\langle e\rangle \mid \texttt{~}e \mid \texttt{\%}e \mid \texttt{open}\ e \mid \texttt{close}\ e \mid$ |
| | | | $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ |

Figura 3.19: Sintaxis del Lenguaje $\lambda^i_{let}$

En la Figura 3.20 se muestra el sistema de tipos de $\lambda^i_{let}$. El juicio de tipado es $\Gamma \vdash^A e : \tau$, donde $A$ es el **nivel nombrado** (que es una lista de clasificadores) donde ocurre la expresión. En la regla Abs el nivel nombrado de la abstracción se almacena en el ambiente. En la regla Var el nivel nombrado asociado a la variable tiene que ser el mismo que el nivel actual. En la regla Br, para chequear el tipo de $\langle e\rangle$ en el nivel nombrado $A$, se asigna un clasificador $\alpha$ para el tipo del código, y el cuerpo $e$ del código se chequea en el nivel nombrado $A, \alpha$. La regla Esc en el nivel nombrado $A, \alpha$ permite solamente la

incorporación de fragmentos de código de tipo $\langle\tau\rangle^\alpha$. La regla Csp, para la persistencia entre etapas, permite llevar una expresión $e$ a nivel superior. El operador `run` ejecuta solamente código con tipo código ejecutable $\langle\tau\rangle$, el cual es introducido por la regla Close, para `close` $e$, si $e$ puede ser clasificado por cualquier $\alpha$. En la regla Open se elimina el tipo código ejecutable, indicando que `open` $e$ puede ser clasificado por cualquier $\alpha$.

$$\frac{\Gamma, x : \tau_1^A \vdash^A e : \tau_2}{\Gamma \vdash^A \lambda x.e : (\tau_1 \to \tau_2)}\ \text{Abs} \qquad \frac{x : \tau^A \in \Gamma}{\Gamma \vdash^A x : \tau}\ \text{Var}$$

$$\frac{\Gamma \vdash^{A,\alpha} e : \tau}{\Gamma \vdash^A \langle e\rangle : \langle\tau\rangle^\alpha}\ \text{Br} \qquad \frac{\Gamma \vdash^A e : \langle\tau\rangle^\alpha}{\Gamma \vdash^{A,\alpha\sim} e : \tau}\ \text{Esc} \qquad \frac{\Gamma \vdash^A e : \tau}{\Gamma \vdash^{A,\alpha} \texttt{\%}e : \tau}\ \text{Csp}$$

$$\frac{\Gamma \vdash^A e : \langle\tau\rangle}{\Gamma \vdash^A \texttt{open}\ e : \langle\tau\rangle^\alpha}\ \text{Open} \qquad \frac{\begin{array}{c}\Gamma \vdash^A e : \langle\tau\rangle^\alpha \\ \alpha \notin \text{FV}(\Gamma, A, \tau)\end{array}}{\Gamma \vdash^A \texttt{close}\ e : \langle\tau\rangle}\ \text{Close} \qquad \frac{\Gamma \vdash^A e : \langle\tau\rangle}{\Gamma \vdash^A \texttt{run}\ e : \tau}\ \text{Run}$$

Figura 3.20: Reglas de Tipado de un Fragmento de $\lambda^i_{let}$

### 3.2.4. Tipado Dinámico

Además del chequeo de tipos estático, en MetaML [SBM00] se ofrece un mecanismo experimental de chequeo de tipos dinámico del código generado. La idea se basa en la propuesta de Shields *et al.* [SSP98], que consiste en realizar en etapas el chequeo de tipos del código generado, junto con su evaluación.

$$\begin{array}{llll} \text{Tipos} & \tau \in \mathsf{T} & ::= & b \mid \tau \to \tau \mid \langle\rangle \\ \text{Términos} & e \in \mathsf{E} & ::= & c \mid x \mid \lambda x.\ e \mid e_1\ e_2 \mid \\ & & & \langle e\rangle \mid \tilde{}e \mid \texttt{run}\ (e_1, e_2) \end{array}$$

Figura 3.21: Sintaxis de $\lambda_{dyn}$

Se utiliza un único tipo $\langle\rangle$ para todas la expresiones de tipo código, por lo que se remueve la información del tipo del código. La sintaxis que se muestra en la Figura 3.21 es un fragmento de los términos fuente de $\lambda_{dyn}$, los cuales son tipados implícitamente. Las reglas de tipado y la semántica operacional se realizan sobre términos anotados, los que se generan mediante la aplicación de un algoritmo de anotación sobre los términos fuente [SSP98, sección 5.4].

En la Figura 3.22 se muestra una simplificación de la semántica estática del fragmento de manipulación de código de $\lambda_{dyn}$. En la regla Defer se puede ver como se pierde la información del tipo de $e$. El chequeo de tipos del código generado se realiza cuando es ejecutado. En la regla Run se especifica el tipo que debe tener la ejecución del código

mediante la introducción de una segunda expresión $e_2$, la cual sirve a su vez de alternativa en caso de un error de tipos en tiempo de ejecución.

Dado que el código no tiene información de tipo, la aplicación de la regla Splice no asegura la generación de un código válido. Por ejemplo, la expresión $(\lambda x^{\langle\rangle}.\langle\sim x^{\mathtt{int}}\rangle\ \langle\mathtt{true}\rangle)$ está bien tipada, dado que el requerimiento de que el código ligado a $x$ tiene que ser entero no se puede verificar estáticamente. Al evaluarse una expresión como ésta, en lugar de generarse un error en tiempo de ejecución, se retorna un valor especial $\langle\mathtt{Fail}\rangle$ que expresa que el código generado está mal tipado.

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \langle e\rangle : \langle\rangle}\ \text{Defer} \qquad \frac{}{\Gamma \vdash^{n} \langle\mathtt{Fail}\rangle : \langle\rangle}\ \text{Fail}$$

$$\frac{\Gamma \vdash^{n} e : \langle\rangle}{\Gamma \vdash^{n+1} \sim e^{\tau} : \tau}\ \text{Splice} \qquad \frac{\Gamma \vdash^{n} e_1 : \langle\rangle \quad \Gamma \vdash^{n} e_2 : \tau}{\Gamma \vdash^{n} \mathtt{run}\ (e_1, e_2^{\tau}) : \tau}\ \text{Run}$$

Figura 3.22: Reglas de Tipado Simplificadas de un Fragmento de $\lambda_{dyn}$

En la Figura 3.23 se puede ver una simplificación de la semántica de la evaluación del `run` en la etapa 0. La idea general es evaluar $e_1$ a un código y realizar sobre el mismo un chequeo de tipos y una unificación con el tipo deseado. Por lo tanto, con $\cdot \vdash v' : \tau$ queremos decir que, en tiempo de ejecución verificamos que $v'$ está bien tipada en el contexto vacío y que tiene tipo $\tau$, mientras que con $\cdot \nvdash v' : \tau$ decimos que o no está bien tipada o no unifica con el tipo deseado. Si se da el primer caso se ejecuta el código; en el otro caso se evalúa la expresión $e_2$.

$$\frac{e_1 \overset{0}{\hookrightarrow} \langle v'\rangle \qquad \cdot \vdash v' : \tau \qquad v' \overset{0}{\hookrightarrow} v}{\mathtt{run}(e_1, e_2^{\tau}) \overset{0}{\hookrightarrow} \langle v\rangle} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle v'\rangle \qquad \cdot \nvdash v' : \tau \qquad e_2 \overset{0}{\hookrightarrow} v}{\mathtt{run}(e_1, e_2^{\tau}) \overset{0}{\hookrightarrow} \langle v\rangle}$$

Figura 3.23: Reglas de Evaluación Simplificadas del Run de $\lambda_{dyn}$

Al evaluarse una expresión $\sim e^{\tau}$, también se debe verificar que el tipo del código al que evalúa $e$ sea correcto y unifique con $\tau$. Si esto falla, todo el código que se está construyendo debe evaluar a $\langle\mathtt{Fail}\rangle$.

### 3.2.5. Análisis Intensional en MetaML

MetaML incluye una característica experimental que provee de un mecanismo de *pattern matching*[6] sobre piezas de código [SBM00]. Un ejemplo de uso es una función que verifica si un código es el literal 5:

[6] En este trabajo se utilizará **concordancia** como traducción de *matching*. Por lo tanto, *pattern matching* se traducirá como concordancia de patrones.

```
-| fun is5 <5> = true | is5 _ = false;
val is5 = fn : <int> -> bool

-| is5 <5>;
val it = true : bool

-| is5 <2>;
val it = false : bool
```

También se pueden incluir variables en los patrones a través de escapes en las mismas. En el siguiente ejemplo se muestra una función que retorna un par con los sumandos si se le pasa el código de una suma, o una indicación de falla en otro caso[7].

```
-| fun parts <~x + ~y> = SOME(x,y)
     | parts _ = NONE;
 val parts = fn : <int> -> (<int> * <int>) option

-| parts <2 + 3>;
val it = SOME(2,3) : (<int> * <int>) option

-| parts <2>;
val it = NONE : (<int> * <int>) option
```

Para concordar piezas de código en las que ocurren ligaduras de variables se utilizan patrones de alto orden, los cuales se indican como una aplicación escapada. La aplicación debe cumplir con las restricciones de que sea la aplicación de una variable a ciertos argumentos y de que estos argumentos sean variables citadas. Algunos ejemplos de patrones de alto orden válidos son `(f <x>)` y `(f <x><y>)`.

Mediante el uso de análisis intensional se puede, por ejemplo, reescribir la función generadora de potencias para que tome una función citada y devuelva una representación eficiente de la potencia de esa función.

```
-| val fpow  <fn w => ~(f <w>)> =
    let val rec fpow' = fn n => fn x =>
                        if n=0 then <1> else <~x * ~(fpow' (n-1) x)>)
    in
        fn n => <fn w => ~(fpow' n (f <w>))>
    end;
val fpow = fn  : ['a].<'a  -> int> -> int -> <'a  -> int>
```

Si por ejemplo se invoca esta función con los parámetros `<fn w =>w + 1>` y 2, devuelve la función citada `<fn w =>(w + 1) * (w + 1) * 1>`.

Al momento presente no existe ninguna formalización de esta característica de MetaML. Esto se debe a que, dado que fue concebido como un lenguaje de metaprogramación y que por lo tanto uno de sus principales objetivos es la generación de código optimizado, en MetaML se realizan ciertas reducciones en niveles superiores a cero. Entonces pueden ocurrir ciertas incongruencias como la siguiente:

[7] Para ello se usa el tipo `option`, cuya definición es la siguiente:

```
datatype 'a option = SOME of 'a | NONE
val SOME : ['a].'a ->'a option
val NONE : ['a].'a option
```

```
-| fun isApp <~x ~y> = true
     | isApp _ = false;
val isApp = fn  : ['a].<'a > -> bool

-| isApp <(fn x => x) (fn y => y)>;
val it = false : bool
```

que ocurre porque el código `<(fn x =>x) (fn y =>y)>` es reemplazado por el código "equivalente" `<(fn y =>y)>`.

## 3.3. El lenguaje $\nu^{\Box}$

Mientras que MetaML se puede ver como una extensión de $\lambda^{\bigcirc}$, el cálculo $\nu^{\Box}$ [Nan02a, Nan02b, Nan04, NP05] se basa en $\lambda^{\Box}$, extiendéndolo para poseer la habilidad de representar expresiones citadas abiertas y especificar **sustitución explícita** con captura. El principal objetivo de esta propuesta es soportar la inspección de código.

La idea de la extensión es emplear una nueva categoría semántica de **nombres** (de un conjunto infinito numerable $\mathsf{N}$) para referirse a las variables libres de expresiones de tipo código, y emplear sustituciones de nombre explícitas para su captura. Si una expresión citada depende de algunos nombres está parcialmente especificada, y no puede ser evaluada hasta que todos estos nombres sean sustituídos. Por lo tanto, al igual que en $\lambda^{\Box}$, una expresión de tipo código no puede contener variables libres, pero sí se permite que contenga nombres libres, siempre que los mismos sean listados en el tipo de la expresión.

| | | | |
|---|---|---|---|
| Nombres | $X, Y \in \mathsf{N}$ | | |
| Soportes | $C, D \in \mathsf{S}$ | $::=$ | $\cdot \mid C, X$ |
| Tipos | $\tau \in \mathsf{T}$ | $::=$ | $b \mid \tau \to \tau \mid \tau \nrightarrow \tau \mid \Box_C \tau$ |
| Sustituciones Explícitas | $\Theta \in \mathsf{SE}$ | $::=$ | $\cdot \mid X \to e, \Theta$ |
| Contextos Ordinarios | $\Gamma \in \mathsf{G}$ | $::=$ | $\cdot \mid \Gamma, x : \tau$ |
| Contextos Modales | $\Delta \in \mathsf{D}$ | $::=$ | $\cdot \mid \Gamma, u :: \tau[C]$ |
| Contexto de Nombres | $\Sigma \in \mathsf{CN}$ | $::=$ | $\cdot \mid \Sigma, X : \tau$ |
| Términos | $e \in \mathsf{E}$ | $::=$ | $X \mid x \mid \langle\Theta\rangle u \mid \lambda x : \tau.\ e \mid e_1\ e_2 \mid$ `box` $e \mid$ `let box` $u = e_1$ `in` $e_2 \mid$ $\nu X : \tau . e \mid$ `choose` $e$ |

Figura 3.24: Sintaxis de $\nu^{\Box}$

Como se muestra en la Figura 3.24, el tipo código es ahora $\Box_C \tau$, cuyos valores son las expresiones sintácticas cerradas que contienen los nombres del conjunto $C$. A este conjunto finito de nombres se lo llama **conjunto soporte**. Existen tres contextos: $\Gamma$ asocia tipos con las variables que almacenan valores ordinarios (llamadas variables de valor), $\Delta$ asocia tipos y soportes con variables modales, y $\Sigma$ asocia tipos con nombres.

Se requiere que todas las referencias a variables modales tengan como prefijo una sustitución explícita $\Theta$. La sustitución explícita $\langle\Theta\rangle u$ es un conjunto de pares $X \to e$, donde $X$ es un nombre y $e$ es un término, que provee definiciones para los nombres en la expresión ligada a $u$. La sustitución vacía $\langle\cdot\rangle u$ se puede escribir simplemente como $u$.

Los términos $\nu X : \tau_1.e$ y `choose` $e$ son la introducción y eliminación del constructor de tipo $\tau_1 \nrightarrow \tau_2$ respectivamente, y son utilizados para agregar nombres a las expresiones. El primer término agrega un nombre $X$ para que pueda ser usado en $e$, mientras que el segundo, aplicado a una $\nu$ abstracción, sustituye el nombre agregado por un nombre nuevo (no usado) y evalúa su cuerpo.

En el siguiente ejemplo, suponiendo una sintaxis tipo SML y la presencia de operadores `let` y funciones recursivas, se muestra una función `pow` que toma un entero, genera un nuevo nombre `X` entero y construye la expresión `v=X*...*X*1` de tipo entero y soporte {`X`}, y luego sustituye explícitamente el nombre `X` con una nueva variable `x`.

```
fun pow (n : int) : □ (int -> int) =
   choose (νX : int.
           let fun pow' (m : int) : □_X int =
               if m = 0 then box 1
               else
                   let box u = pow' (m - 1)
                   in
                       box(X * u)
                   end
           in
               let box v = pow' (n)
               in
                   box (λx : int. <X -> x> v)
               end
           end)
```

Por lo tanto invocar `exp 2` retornaría:

```
box (λx : int. x * (x * 1)) : □(int -> int).
```

El sistema de tipos de la Figura 3.25 consiste en dos juicios de tipado: un juicio para las expresiones $\Sigma; \Delta; \Gamma \vdash e : \tau[C]$, que significa que en los contextos nombrados la expresión $e$ tiene tipo $\tau$ y soporte $C$, y un juicio para las sustituciones explícitas $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, que indica que la sustitución $\Theta$ mapea de expresiones con soporte $[C]$ a expresiones con soporte $[D]$. Aplicar una sustitución vacía (EmpS) no cambia el término; es decir que cuando se aplica una sustitución vacía sobre un término que contiene nombres de $C$, el resultado contiene los mismos nombres, aunque se permite debilitar a un superconjunto $D$ de $C$. Cuando se aplica una sustitución explícita $\Theta : [C] \Rightarrow [D]$ a una expresión $e : \tau[C]$ entonces $\langle\Theta\rangle e$ tiene soporte $D$.

Las reglas de Hipótesis refieren a la relación correcta entre los distintos tipos de variables y sus contextos. De los tres tipos de contextos surgen las tres reglas de Hipótesis. La regla para los nombres dice que un nombre $X$ puede ser utilizado si fue declarado en $\Sigma$ y se encuentra en el conjunto soporte. La regla para las variables dice que el tipo $\tau$ de $x$ puede ser inferido si está declarado en $\Gamma$, y que el conjunto soporte puede ser cualquier conjunto soporte $C$ bien-formado. La regla para las variables modales indica que estas variables deben estar precedidas por una sustitución explícita que matchee del soporte declarado en $\Delta$ al soporte $D$ del tipo de la expresión.

Si obviamos los contextos $\Sigma$ y $\Delta$ y el conjunto soporte, las reglas de cálculo lambda son las usuales, salvo que en este caso se verifica también que se utilicen los mismos conjuntos

Sustituciones Explícitas

$$\frac{C \subseteq D}{\Sigma; \Delta; \Gamma \vdash \langle\rangle : [C] \Rightarrow [D]}\ \text{EmpS}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash \tau[D] \qquad \Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C \setminus \{X\}] \Rightarrow [D] \qquad X : \tau \in \Sigma}{\Sigma; \Delta; \Gamma \vdash \langle X \to e, \Theta\rangle : [C] \Rightarrow [D]}\ \text{CompS}$$

Hipótesis

$$\frac{X : \tau \in \Sigma}{\Sigma; \Delta; \Gamma \vdash X : \tau[X, C]} \qquad \frac{x : \tau \in \Gamma}{\Sigma; \Delta; \Gamma \vdash x : \tau[C]}$$

$$\frac{\Sigma; (\Delta, u :: \tau[C]); \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]}{\Sigma; (\Delta, u :: \tau[C]); \Gamma \vdash \langle\Theta\rangle u : \tau[D]}$$

$\lambda$-cálculo

$$\frac{\Sigma; \Delta; (\Gamma, x : \tau_1) \vdash e : \tau_2[C]}{\Sigma; \Delta; \Gamma \vdash \lambda x : \tau_1.\ e : (\tau_1 \to \tau_2)[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2[C] \qquad \Sigma; \Delta; \Gamma \vdash e_2 : \tau_1[C]}{\Sigma; \Delta; \Gamma \vdash e_1\ e_2 : \tau_2[C]}$$

Modal

$$\frac{\Sigma; \Delta; \cdot \vdash e : \tau[D]}{\Sigma; \Delta; \Gamma \vdash \texttt{box}\ e : \Box_D \tau[C]}\ \text{Box} \qquad \frac{\begin{array}{c}\Sigma; \Delta; \Gamma \vdash e_1 : \Box_D \tau_1[C] \\ (\Sigma; \Delta, u :: \tau_1[D]); \Gamma \vdash e_2 : \tau_2[C]\end{array}}{\Sigma; \Delta; \Gamma \vdash \texttt{let box}\ u = e_1\ \texttt{in}\ e_2 : \tau_2[C]}\ \text{LetBox}$$

Nombres

$$\frac{(\Sigma, X : \tau_1); \Delta; \Gamma \vdash e : \tau_2[C]}{\Sigma; \Delta; \Gamma \vdash \nu X : \tau_1.\ e : (\tau_1 \nrightarrow \tau_2)[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash e : (\tau_1 \nrightarrow \tau_2)[C]}{\Sigma; \Delta; \Gamma \vdash \texttt{choose}\ e : \tau_2[C]}$$

Figura 3.25: Reglas de Tipado de $\nu^{\Box}$

soporte.

El fragmento modal es análogo al de $\lambda^{\Box}$ extendido con los conjuntos soporte. En la regla Box se debe cumplir que el soporte de $e$ es el que se indica como subíndice del tipo $\Box$. El soporte de toda la expresión `box` $e$ en principio es vacío, pero se puede debilitar a cualquier conjunto bien-formado $C$. En LetBox se verifica que si $e_1$ tiene tipo $\Box_D\tau_1$, el soporte de $u$ sea $D$.

En cuanto al fragmento de nombres, el término $\nu X : \tau_1.e$ liga un nombre $X$ de tipo $\tau_1$ para que pueda ser utilizado en $e$. El término `choose` $e$ toma un nombre nuevo de tipo $\tau_1$, lo sustituye por el nombre ligado en la $\nu$-abstracción parámetro de tipo $\tau_1 \nrightarrow \tau_2$ y evalúa el cuerpo de la abstracción.

### 3.3.1. Análisis Intensional en $\nu^{\Box}$

Existe una extensión del cálculo con primitivas de concordancia de patrones sobre expresiones de código, utilizado para inspeccionar la estructura de un programa objeto y destruirlo en sus partes componentes. Esta extensión está definida sólo para realizar análisis intensional sobre el fragmento de cálculo lambda del lenguaje.

| | | | |
|---|---|---|---|
| Patrones Variable | $w \in \mathsf{W}$ | | |
| Patrones de Alto Orden | $\pi \in \mathsf{P}$ | ::= | $(w\ x_1\ ...\ x_n) : \tau[C] \mid X \mid x \mid \lambda x : \tau.\ \pi \mid \pi_1\ \pi_2$ |
| Asignaciones de Patrones | $\sigma \in \mathsf{AP}$ | ::= | $\cdot \mid w \rightarrow e, \sigma$ |
| Términos | $e \in \mathsf{E}$ | ::= | $... \mid$ `case` $e$ `of box` $\pi \Rightarrow e_1$ `else` $e_2$ |

Figura 3.26: Sintaxis de la Extensión para Análisis de $\nu^{\Box}$

La extensión, cuya sintaxis se encuentra en la Figura 3.26, consiste en una construcción `case` que concuerda expresiones de tipo código, y una serie de patrones de alto orden para realizar esta concordancia.

El patrón de alto orden $(w\ x_1\ ...\ x_n) : \tau[C]$ declara una variable de patrón $w$ que concuerda una expresión código si sus variables libres se encuentran entre $x_1, ..., x_n$. El patrón $\lambda x : \tau.\pi$ concuerda una $\lambda$-abstracción si la misma tiene dominio de tipo $\tau$ y su cuerpo concuerda con el patrón $\pi$, declarando una nueva variable $x$ local al patrón. La variable $x$ es una constante sintáctica y sólo puede concordar con el patrón $x$. El patrón $X$ concuerda un nombre $X$ y el patrón $\pi_1\ \pi_2$ concuerda una aplicación si su función y argumento concuerdan con los patrones $\pi_1$ y $\pi_2$ respectivamente.

El juicio para los patrones tiene la forma $\Sigma; \Gamma \vdash \pi : \tau[C] \Longrightarrow \Gamma_1$, que se lee: en el contexto de nombres $\Sigma$ y el contexto de variables locales $\Gamma$, el patrón $\pi$ tiene el tipo $\tau$, el conjunto soporte $C$ y produce el contexto residual $\Gamma_1$ de variables de patrones. Las reglas se muestran en la Figura 3.27. Notar que en el caso de las constantes sintácticas ($X$ y $x$) no se produce contexto residual, y que para $\lambda x : \tau.\pi$ y $\pi_1\ \pi_2$ el contexto es producido por sus sub-patrones. La regla que produce contexto residual es la del patrón variable $(w\ x_1\ ...\ x_n) : \tau[C]$, el cual concuerda expresiones de tipo $\tau$ con las variables dadas y los nombres declarados en un subconjunto $D$ de $C$. Para que el patrón esté bien tipado las variables $x_1 : \tau_1$, ..., $x_n : \tau_n$ tienen que estar declaradas en el contexto local $\Gamma$. En el contexto producido $w$ se tipa como una función sobre los tipos $\Box_P\tau_i$ con soporte polimórfico.

$$\frac{D \subseteq C \qquad p \notin \Sigma}{\Sigma; (\Gamma, x_1 : \tau_1, ..., x_n : \tau_n) \vdash ((w\ x_1\ ...\ x_n) : \tau[D]) : \tau[C] \Longrightarrow w : \forall p. \Box_p \tau_1 \to ... \to \Box_p \tau_n \to \Box_{p,D} \tau_n}$$

$$\frac{X : \tau \in \Sigma}{\Sigma; \Gamma \vdash X : \tau[X, C] \Longrightarrow \cdot} \qquad \frac{}{\Sigma; (\Gamma, x : \tau) \vdash x : \tau[C] \Longrightarrow \cdot}$$

$$\frac{\Sigma; (\Gamma, x : \tau_1) \vdash \pi : \tau_2[C] \Longrightarrow \Gamma_1}{\Sigma; \Gamma \vdash \lambda x : \tau_1.\ \pi : (\tau_1 \to \tau_2)[C] \Longrightarrow \Gamma_1}$$

$$\frac{\Sigma; \Gamma \vdash \pi_1 : \tau_1 \to \tau_2[C] \Longrightarrow \Gamma_1 \qquad \Sigma; \Gamma \vdash \pi_2 : \tau_1[C] \Longrightarrow \Gamma_2 \qquad \mathit{fn}(A) \subseteq \mathit{dom}(\Sigma)}{\Sigma; \Gamma \vdash \pi_1\ \pi_2 : \tau_2[C] \Longrightarrow (\Gamma_1, \Gamma_2)}$$

Figura 3.27: Reglas de Tipos de los Patrones de $\nu^{\Box}$

La regla de tipado del operador `case` se puede ver en la Figura 3.28. En este operador se evalúa el argumento $e$, de tipo $\Box_D \tau_1$ con soporte $C$, para obtener una expresión `box` $e'$. Luego se machea $e'$, de tipo $\tau_1$ con soporte $D$, con el patrón $\pi$. Notar que en el juicio del patrón el contexto de las variables de valor debe ser vacío, por lo que un patrón no puede contener variables del exterior. Si el macheo es exitoso se produce un contexto $\Gamma_1$, con las ligaduras de los patrones variable, y se evalúa la expresión $e_1$ en el ambiente compuesto por el actual y $\Gamma_1$. En otro caso, se evalúa $e_2$.

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma \vdash e : \Box_D\ \tau_1[C] \qquad \Sigma; \cdot \vdash \pi : \tau_1[D] \Longrightarrow \Gamma_1 \\ \Sigma; \Delta; (\Gamma, \Gamma_1) \vdash e_1 : \tau_2[C] \qquad \Sigma; \Delta; \Gamma \vdash e_2 : \tau_2[C]\end{array}}{\Sigma; \Delta; \Gamma \vdash \texttt{case}\ e\ \texttt{of box}\ \pi \Rightarrow e_1\ \texttt{else}\ e_2 : \tau_2[C]}\ \text{Case}$$

Figura 3.28: Reglas de Tipos de la Extensión para Análisis de $\nu^{\Box}$

Una versión en el cálculo $\nu^{\Box}$ del generador de potencias de funciones de la sección 3.2.5 es la siguiente:

```
fun fpow (f:□ (int->int)) (n : int) :□ (int -> int) =
   choose (νX : int.
           let fun fpow' (m : int) : □X int =
               if m = 0 then box 1
               else
                   let box u = fpow' (m - 1)
                   in
                       box(X * u)
                   end
```

```
in
    case f of
        box (λw:int [g w])=>
            let box v = exp' (n)
            in
                box (λx : int. <X -> g x> v)
            end
end)
```

## 3.4. *reFLect*

El lenguaje funcional *reFLect* [GMO03] fue desarrollado en Intel para aplicaciones de diseño y verificación de hardware. Es utilizado para la simulación de modelos de hardware, implementándolos como programas, y el razonamiento formal sobre los mismos. Por esto la posibilidad de proveer análisis intensional jugó un rol primario en el diseño del lenguaje, lo que lo diferencia de MetaML, cuyo principal objetivo es la generación de código y el control y optimización de su evaluación.

| | | | |
|---|---|---|---|
| Tipos | $\tau \in \mathsf{T}$ | ::= | $b \mid \tau \to \tau \mid \texttt{term}$ |
| Términos | $e \in \mathsf{E}$ | ::= | $x^\tau \mid \lambda e_1.e_2 \mid e_1 \cdot e_2 \mid$ |
| | | | $e_1 + e_2 \mid \langle\langle e \rangle\rangle \mid {}^\wedge e^\tau$ |

Figura 3.29: Sintaxis de *reFLect*

En [KM04] se define una versión minimal del lenguaje, que es la que presentaremos en esta sección y cuya sintaxis se describe en la Figura 3.29. Existe un único tipo `term` para todas las expresiones de tipo código, las cuales se construyen con $\langle\langle e \rangle\rangle$ y se escapan con ${}^\wedge e^\tau$. Notar que el último operador requiere de una anotación explícita del tipo de la expresión citada, debido a que el tipo `term` no provee de esta información. En este sentido la propuesta es similar a la extensión de tipado dinámico de MetaML (sección 3.2.4), con la excepción de que no existe un operador `run` con expresiones alternativas. Por lo tanto, pueden verificarse errores de tipos en tiempo de ejecución.

Algo que diferencia a este lenguaje de los vistos anteriormente es que en este caso la abstracción no liga un sola variable. Sintácticamente en *reFLect* cualquier expresión puede aparecer en la posición de ligado de una abstracción de manera de ligar todas sus variables libres, aunque en realidad estas expresiones tienen ciertas restricciones que se verán más adelante. Esta forma de abstracción fue agregada para proveer de análisis intensional, mediante un mecanismo de concordancia de patrones sobre las expresiones de código. Dado que la concordancia de patrones puede fallar se provee de una construcción de alternación (+), tal que evaluar la expresión $(\lambda e_p.e_1 + e_2) \cdot e_3$ resulta en la evaluación de $(\lambda e_p.e_1) \cdot e_3$, si $e_p$ machea con $e_3$, o de $e_2 \cdot e_3$ en otro caso.

Para explicar la semántica de *reFLect* se debe introducir la noción de **contexto**, que representa a una expresión con un cierto número de agujeros que ocurren en lugar de algunas subexpresiones en el árbol de sintaxis abstracta. Se definen utilizando la misma

gramática que se utiliza para los términos (que se describe en la Figura 3.29) extendida con una nueva producción para representar los huecos $e ::= ... \mid \_^{\tau}$. Se utiliza la notación $\mathcal{C}[\_^{\tau_1}, ..., \_^{\tau_n}]$ para indicar que el contexto $\mathcal{C}$ tiene los $n$ agujeros que se muestran, con los tipos $\tau_1, ..., \tau_n$ respectivamente. Para representar una expresión obtenida de llenar los $n$ agujeros de $\mathcal{C}$ con expresiones $e_1, ..., e_n$, cuyos tipos son $\tau_1, ..., \tau_n$ respectivamente, se escribe $\mathcal{C}[e_1, ..., e_n]$. Se utiliza el término *nivel* para referirse al número de *brackets* menos el número de *escapes* que rodean a una expresión. Una expresión se dice **consistente por niveles** si ninguna porción ocurre en un nivel negativo. De una forma más general, un contexto es consistente por niveles si además de la condición anterior, se cumple que todos los agujeros ocurren en el nivel 0.

$$\frac{}{\vdash x^{\tau} : \tau}\ \text{Var} \qquad \frac{\vdash e_1 : \tau_1 \to \tau_2 \quad \vdash e_2 : \tau_1}{\vdash e_1 \cdot e_2 : \tau_2}\ \text{App} \qquad \frac{\vdash e_1 : \tau_1 \quad \vdash e_2 : \tau_2}{\vdash \lambda e_1.e_2 : \tau_1 \to \tau_2}\ \text{Abs}$$

$$\frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1 + e_2 : \tau}\ \text{Alt} \qquad \frac{\vdash e_i : \texttt{term} \qquad \vdash \mathcal{C}[z_1^{\tau_1}, ..., z_k^{\tau_k}] : \tau \quad (1 \leq i \leq k, \mathcal{C} \text{ nivel consistente}, z_i \text{ fresca})}{\vdash \langle\!\langle \mathcal{C}[\hat{}\, e_1^{\tau_1}, ..., \hat{}\, e_k^{\tau_k}] \rangle\!\rangle : \texttt{term}}\ \text{Quote}$$

Figura 3.30: Reglas de Tipos de *reFLect*

Una expresión está bien tipada si es nivel consistente y se puede derivar el juicio $\vdash e : \tau$ de las reglas de la Figura 3.30. En [KM04] se menciona que dada una expresión $e$, existen un único contexto nivel consistente $\mathcal{C}[\_^{\tau_1}, ..., \_^{\tau_n}]$ y un único conjunto de expresiones $e_1, ..., e_n$, tal que $e$ es sintácticamente idéntica a $\mathcal{C}[\hat{}\, e_1^{\tau_1}, ..., \hat{}\, e_k^{\tau_k}]$. En base a lo anterior, en la regla Quote las expresiones de tipo código se expresan en forma de contextos con sus agujeros llenados con escapes. En esta regla se indican dos cosas a destacar. Primero, que si una expresión $e$ con tipo $\tau$ es bien-tipada, entonces la expresión $\langle\!\langle e \rangle\!\rangle$ es también bien-tipada y tiene tipo `term`, no figurando en éste el tipo $\tau$. Y, por último, que una expresión de escape $\hat{}\, e$ es bien-tipada si $e$ tiene tipo `term`, sin importar el tipo anotado.

En [GMO03] se presenta una función de evaluación de código citado $\texttt{value}^{\tau} : \texttt{term} \to \tau$ como una extensión del lenguaje. De forma similar a MetaML, esta función está restringida a operar sólo sobre expresiones cerradas.

### 3.4.1. Análisis Intensional en *reFLect*

La expresión que aparece en la parte de ligadura de una abstracción es tratada como un patrón. Los patrones en *reFLect* pueden ser variables o de código, los cuales son expresiones citadas de la forma $\langle\!\langle \mathcal{C}[\hat{}\, (x_1^{\texttt{term}})^{\tau_1}, ..., \hat{}\, (x_k^{\texttt{term}})^{\tau_k}] \rangle\!\rangle$, donde $\mathcal{C}$ es un contexto nivel consistente. Los patrones de código permiten inspeccionar y transformar código.

En la Figura 3.31 se muestra una simplificación de las reglas de relación de concordancia de patrones, cuya notación $\theta : e_p \Rightarrow e$ expresa que el patrón $e_p$ machea la expresión bien formada $e$ vía la sustitución $\theta$; siendo entonces $(\lambda e_p.e_b) \cdot e \hookrightarrow e_b\theta$ la reducción para la abstracción, que significa que si $e_p$ machea con $e$ vía la sustitución $\theta$ la abstracción reduce a aplicar esa sustitución a $e_b$.

Cuando el patrón es una variable, el patrón $x^{\tau}$ machea con cualquier expresión $e$ con tipo $\tau$, bajo una sustitución que mapea $x$ con $e$, teniendo el mismo efecto que las abstrac-

$$\frac{e : \tau \qquad \theta = [e/x^{\tau}]}{\theta : x^{\tau} \Rightarrow e} \qquad \frac{\theta = [\langle\!\langle e_1 \rangle\!\rangle / x_1^{\texttt{term}}, ..., \langle\!\langle e_k \rangle\!\rangle / x_k^{\texttt{term}}] \qquad e_i : \tau_i \qquad (i = 1, ..., k)}{\theta : \langle\!\langle \mathcal{C}[{}^{\wedge}(x_1^{\texttt{term}})^{\tau_1}, ..., {}^{\wedge}(x_k^{\texttt{term}})^{\tau_k}] \rangle\!\rangle \Rightarrow \langle\!\langle \mathcal{C}[e_1, ..., e_k] \rangle\!\rangle}$$

Figura 3.31: Concordancia de Patrones en *reFLect*

ciones normales. En el caso de los patrones de código, si la expresión que se quiere machear y el patrón pueden expresarse con un mismo contexto nivel consistente $\mathcal{C}[\_^{\tau_1}, ..., \_^{\tau_k}]$, entonces ambos tienen una misma estructura sintáctica. Por lo tanto, si la anotación del escape de cada variable de tipo código $x_i$ coincide con el tipo de la subexpresión $e_i$ correspondiente, la expresión machea bajo la sustitución que mapea cada variable $x_i$ con $\langle\!\langle e_i \rangle\!\rangle$.

Por ejemplo, si suponemos la existencia de un operador de multitplicación, la expresión $(\lambda\langle\!\langle {}^{\wedge}(x^{\texttt{term}})^{\texttt{int}} *^{\wedge} (y^{\texttt{term}})^{\texttt{int}} \rangle\!\rangle.\langle\!\langle {}^{\wedge}(y^{\texttt{term}})^{\texttt{int}} *^{\wedge} (x^{\texttt{term}})^{\texttt{int}} \rangle\!\rangle)$ conmuta los argumentos de una multiplicación citada. Entonces, si se aplicara a $\langle\!\langle 1 * 2 \rangle\!\rangle$, al aplicarse la sustitución $[\langle\!\langle 1 \rangle\!\rangle / x^{\texttt{term}}, \langle\!\langle 2 \rangle\!\rangle / y^{\texttt{term}}]$ resultaría en $\langle\!\langle 2 * 1 \rangle\!\rangle$.

La combinación entre el análisis intensional y la existencia de un único tipo para el código permite la construcción de funciones que atraviesen expresiones citadas y manipulen su estructura. Por lo tanto, si suponemos la existencia de un operador de punto fijo en el lenguaje, se puede extender el ejemplo anterior a uno que conmute los argumentos de todas las multiplicaciones de una expresión.

$$\begin{aligned}
&\texttt{fix } p. \\
&\quad \lambda\langle\!\langle {}^{\wedge}(x^{\texttt{term}})^{\texttt{int}} *^{\wedge} (y^{\texttt{term}})^{\texttt{int}} \rangle\!\rangle.\langle\!\langle {}^{\wedge}(p \cdot y^{\texttt{term}})^{\texttt{int}} *^{\wedge} (p \cdot x^{\texttt{term}})^{\texttt{int}} \rangle\!\rangle + \\
&\quad \lambda\langle\!\langle {}^{\wedge}(f^{\texttt{term}})^{\tau_1 \to \tau_2} .^{\wedge} (x^{\texttt{term}})^{\tau_1} \rangle\!\rangle.\langle\!\langle {}^{\wedge}(p \cdot f^{\texttt{term}})^{\tau_1 \to \tau_2} .^{\wedge} (p \cdot x^{\texttt{term}})^{\tau_1} \rangle\!\rangle + \\
&\quad \lambda\langle\!\langle {}^{\wedge}(x^{\texttt{term}})^{\tau} +^{\wedge} (y^{\texttt{term}})^{\tau} \rangle\!\rangle.\langle\!\langle {}^{\wedge}(p \cdot x^{\texttt{term}})^{\tau} +^{\wedge} (p \cdot y^{\texttt{term}})^{\tau} \rangle\!\rangle + \\
&\quad \lambda\langle\!\langle \lambda^{\wedge}(x^{\texttt{term}})^{\tau_1} .^{\wedge} (y^{\texttt{term}})^{\tau_2} \rangle\!\rangle.\langle\!\langle \lambda^{\wedge}(p \cdot x^{\texttt{term}})^{\tau_1} .^{\wedge} (p \cdot y^{\texttt{term}})^{\tau_2} \rangle\!\rangle + \\
&\quad \lambda x.x
\end{aligned}$$

## 3.5. Comparación

En esta sección se realizará un análisis rápido y corto de la capacidad de reflexión que brindan los lenguajes vistos, comparándolos en función del tipo de código que pueden manipular, el tiempo en que se realiza el chequeo de tipos, la capacidad de análisis intensional y de persistencia entre etapas. Los información presentada en esta comparación se resume en el Cuadro 3.1.

### 3.5.1. Código Abierto/Cerrado

Mientras que en $\lambda^{\Box}$ sólo se permite la manipulación de código cerrado, en $\lambda^{\bigcirc}$ se puede manipular código abierto. Es por esto que el código generado por el primero puede ser ejecutado y el del segundo no.

En MetaML y en el cálculo $\nu^{\Box}$ se intenta permitir manipular código abierto y a su vez tener un operador de ejecución del mismo, que es lo deseable en un lenguaje con reflexión de código. En MetaML existen varios enfoques para tratar de hacer esto de forma segura. En el cálculo $\nu^{\Box}$ la idea es basarse en el cálculo $\lambda^{\Box}$ y utilizar una nueva categoría de

nombres para referirse a las variables libres dentro de una expresión de tipo código. Estos nombres son capturados de forma simple mediante sustitución explícita.

En este sentido *reFLect* se comporta de forma similar a MetaML.

### 3.5.2. Tipado

En los lenguajes $\lambda^{\square}$, $\lambda^{\bigcirc}$, MetaML y $\nu^{\square}$ el tipo para código tiene asociado el tipo de la expresión citada. De esta forma se asegura estáticamente que el código generado sea bien tipado. El chequeo estático de tipos limita bastante la capacidad de reflexión que provee un lenguaje. Por esto en una propuesta experimental de MetaML y en *reFLect* se sigue un enfoque que aplica chequeo estático en el código no citado y realiza un chequeo en tiempo de ejecución del código citado sólo cuando es estrictamente necesario. La propuesta de Shields *et al* [SSP98] implementada en MetaML resulta interesante porque no se generan errores en tiempo de ejecución cuando el chequeo de tipos del código generado falla.

### 3.5.3. Análisis Intensional

El análisis intensional es una característica esencial cuando se habla de reflexión, dado que permite la introspección y puede resultar útil también para ciertos tipos de intercesión. Por ejemplo, la implementación de analizadores de sistemas, optimizadores y transformadores de código requiere de esta característica.

Ni $\lambda^{\square}$ ni $\lambda^{\bigcirc}$ proveen construcciones para permitir análisis intensional. Tampoco lo hace MetaML de una manera formal. Esto se debe a que el lenguaje fue concebido para la ejecución optimizada del código generado.

En el cálculo $\nu^{\square}$ se puede realizar análisis intensional sobre el fragmento lambda del cálculo mediante un mecanismo de concordancia de patrones. En *reFLect* también se provee de un mecanismo de concordancia de patrones sobre expresiones citadas que, sumado al tipado dinámico de las expresiones de tipo código, brinda un mecanismo de análisis intensional muy potente.

### 3.5.4. Persistencia entre Etapas

Una de las principales características de MetaML es la capacidad de brindar persistencia entre etapas, ya sea de forma implícita o explícita, como sucede en algunas de sus variantes. Esta característica no se repite en ninguno de los demás lenguajes estudiados. Taha [Tah99] observó que en aquellos lenguajes que proveen análisis intensional, y que por lo tanto se puede permitir realizar reducciones sólo en el nivel cero, no se puede proveer de persistencia entre etapas sin una pérdida de confluencia. Por ejemplo, la expresión MetaML $(\lambda x.\langle x\rangle)\ ((\lambda y.y)\ 5)$ puede reducir tanto a $\langle((\lambda y.y)\ 5)\rangle$ como a $\langle 5\rangle$.

Cuadro 3.1: Tabla Comparativa de Capacidad de Reflexión

| **Lenguaje** | **Reificación de Código Abierto** | **Absorción (evaluación)** | **Tipo Dinámico del Código** | **Análisis Intensional** | **CSP** |
|---|---|---|---|---|---|
| $\lambda^{\Box}$ | No | Si | No (tipo $\Box_\tau$) | No | No |
| $\lambda^{\bigcirc}$ | Si | No | No (tipo $\bigcirc_\tau$) | No | No |
| MetaML | Si | Si | No (tipo $\langle\tau\rangle$). Existe una extensión con tipado dinámico (tipo $\langle\rangle$). | No. Existe una extensión experimental limitada. | Si |
| $\nu^{\Box}$ | Si. Se utiliza una categoría de nombres para variables libres. | Si | No (tipo $\Box_C\tau$) | Concordancia de patrones sobre fragmento de lambda cálculo. | No |
| *reFLect* | Si | No. Se extiende con una función $\texttt{value}^\tau$ de código a $\tau$ | Si (tipo `term`) | Si. Concordancia de patrones. | No |

# Capítulo 4

# Lenguaje Reflexivo

*...me dijo, en lengua que en toda la Berbería, y aun en Costantinopla,*
*se halla entre cautivos y moros, que ni es morisca, ni castellana,*
*ni de otra nación alguna, sino una mezcla de todas las lenguas*
*con la cual todos nos entendemos; digo, pues,*
*que en esta manera de lenguaje me preguntó que qué buscaba...*

Fragmento de "Don Quijote de la Mancha".

En este capítulo se presentan la sintaxis y semántica del lenguaje propuesto, para permitir la generación y el análisis del código citado de una manera segura tal como fue planteado en el objetivo del trabajo. Las principales características de este lenguaje están inspiradas en varias de las características de los lenguajes analizados en el capítulo anterior. Algunas de éstas son la manipulación de código abierto de MetaML y *reFLect*, el tipado dinámico presentado por Shields *et al.* [SSP98], el análisis de código mediante un mecanismo de concordancia de patrones de *reFLect* y $\nu^{\Box}$, y el operador de sustituciones explícitas de este último.

## 4.1. Lenguaje

En esta sección se presenta el lenguaje mostrando su sintaxis y su sistema de tipos. Se da también una idea intuitiva de su mecanismo de operación.

### 4.1.1. Cálculo Básico

El núcleo del lenguaje está formado por un cálculo lambda simplemente tipado, a la Church [Bar92], con la siguiente sintaxis:

| | | | |
|---|---|---|---|
| Tipos | $\tau \in \mathsf{T}$ | ::= | $\texttt{int} \mid \texttt{bool} \mid \tau \to \tau \mid \tau \times \tau$ |
| Contextos | $\Gamma \in \mathsf{G}$ | ::= | $\cdot \mid \Gamma, \tau$ |
| Pilas de Contextos | $P \in \mathsf{GS}$ | ::= | $\cdot \mid (P, \Gamma)$ |
| Variables | $x \in \mathsf{X}$ | ::= | $\texttt{z} \mid \texttt{s}\ x$ |
| Términos | $e \in \mathsf{E}$ | ::= | $b \mid i \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid$ |
| | | | $\lambda^{\tau}.\ e \mid \texttt{fix}\ e \mid x^{\tau} \mid e_1\ e_2 \mid$ |
| | | | $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ |

Un tipo $\tau$ puede ser un tipo base (`int` ó `bool`), un tipo función $\tau \to \tau$ o un producto binario $\tau \times \tau$. Se cuenta con literales (booleanos $b$ y enteros $i$), lambda-abstracciones

y aplicaciones. Además de estas construcciones comunes del cálculo lambda se incluyen el constructor y los destructores (`fst` y `snd`) del producto, un operador de punto fijo (`fix`) y un operador de condición (`if-then-else`), para brindar la posibilidad de realizar funciones recursivas[1]. Se utilizan índices de de Bruijn [dB72] para codificar las ligaduras de variables, por lo que las variables son números naturales y los contextos de tipos son secuencias de tipos. De esta manera se evitan conflictos de nombres de variables asociados a $\alpha$-equivalencia

La utilización de los índices de de Bruijn y de las anotaciones de tipo hacen que el código se vuelva muy verboso y difícil de leer. Por lo tanto en la mayoría de los ejemplos se utilizará una sintaxis tipo ML en lugar de la definida para la formalización. Como se muestra a continuación, en la sintaxis de los ejemplos se nombrará a las variables, se omitirán algunas anotaciones de tipo, y se utilizará recursión explícita. Por ejemplo, en lugar de la siguiente expresión se escribe la que está a su derecha.

```
fix                                                  | val rec p =
  λ^int. if (z^int = 0)                              |   fn x:int => if (x = 0)
         then 1                                      |         then 1
         else z^int * (s z)^(int→int) (z^int − 1)    |         else x * p (x - 1)
```

En este ejemplo, escrito con ambas sintaxis, se define una función recursiva que calcula el factorial de un número.

El juicio de tipado es de la forma $P; \Gamma \vdash e : \tau$, que se lee "la expresión $e$ tiene tipo $\tau$ en un contexto local $\Gamma$ bajo la pila $P$"[2]. La presencia de las pilas de contextos en las reglas de tipado de la Figura 4.1, y el uso de un juicio para tipar variables, son las únicas diferencias con el cálculo lambda estándar. Las pilas de contextos pueden ser ignoradas hasta que se explique su uso en la extensión multi-etapas.

En las reglas de tipado para las variables (Figura 4.2), la regla Base proyecta el tipo de índice 0 y la regla Weak, para `s` $n$, proyecta recursivamente el tipo $(n + 1)$. En los ejemplos en los que se utilicen índices de de Bruijn se simplificará la notación de las variables, expresando en números el índice de la variable. Por ejemplo, en lugar de escribir `(s s z)` se escribirá #2. Entonces, si tenemos el contexto $(\Gamma, \texttt{int}, \texttt{bool})$, las variables $\#0^{\texttt{bool}}$ y $\#1^{\texttt{int}}$ están bien tipadas, mientras que $\#0^{\texttt{int}}$ no lo está.

### 4.1.2. Extensión Multi-etapas

Para poder construir y combinar piezas de código se incluyen anotaciones de etapas como parte del lenguaje, las cuales particionan la ejecución de los programas en etapas de computación. La extensión de la sintaxis del núcleo del lenguaje con las anotaciones de etapas es la siguiente:

| | | | |
|---|---|---|---|
| Tipos | $\tau \in \mathsf{T}$ | ::= | $\ldots \mid \texttt{cod}^{\Gamma}$ |
| Sustitutiones Explícitas | $\Theta \in \mathsf{S}$ | ::= | $\Uparrow(\Theta) \mid e/ \mid \uparrow^{\tau}$ |
| Términos | $e \in \mathsf{E}$ | ::= | $\ldots \mid \texttt{[\|}e\texttt{\|]\{}\Gamma\texttt{\}} \mid \texttt{\$}(e)^{\tau} \mid e[\Theta] \mid \texttt{run}\ e_1 \mid e_2$ |

En la extensión se incluyen los paréntesis (`[|e|]`), el escape (`$`$(e)$), el operador de evaluación de código (`run`) y la sustitución explícita ($\Theta$). Los primeros se corresponden con $\langle e \rangle$, $\sim e$ y `run` de MetaML respectivamente, aunque con ciertas diferencias que se describirán en las próximas subsecciones.

[1] Las funciones recursivas son importantes cuando se agrega análisis intensional, dado que la mayoría de las operaciones de análisis consisten en descomponer una expresión y analizar recursivamente sus partes componentes.

[2] En el apéndice A se resumen todos las juicios de tipado del lenguaje y la forma en que se leen.

$$\frac{}{P;\Gamma \vdash i : \texttt{int}}\ \text{LInt} \qquad \frac{}{P;\Gamma \vdash b : \texttt{bool}}\ \text{LBool}$$

$$\frac{P;\Gamma \vdash e_1 : \tau_1 \quad P;\Gamma \vdash e_2 : \tau_2}{P;\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \text{Pair} \qquad \frac{P;\Gamma \vdash e : \tau_1 \times \tau_2}{P;\Gamma \vdash \texttt{fst}\ e : \tau_1}\ \text{Fst} \qquad \frac{P;\Gamma \vdash e : \tau_1 \times \tau_2}{P;\Gamma \vdash \texttt{snd}\ e : \tau_2}\ \text{Snd}$$

$$\frac{P;\Gamma, \tau_1 \vdash e : \tau_2}{P;\Gamma \vdash \lambda^{\tau_1}.e : \tau_1 \to \tau_2}\ \text{Abs} \qquad \frac{P;\Gamma, \tau \vdash e : \tau}{P;\Gamma \vdash \texttt{fix}\ e : \tau}\ \text{Fix}$$

$$\frac{P;\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad P;\Gamma \vdash e_2 : \tau_2}{P;\Gamma \vdash e_1\ e_2 : \tau_1}\ \text{App} \qquad \frac{\Gamma \vdash x : \tau}{P;\Gamma \vdash x^{\tau} : \tau}\ \text{Var}$$

$$\frac{\begin{array}{c} P;\Gamma \vdash e_1 : \texttt{bool} \\ P;\Gamma \vdash e_2 : \tau \quad P;\Gamma \vdash e_3 : \tau \end{array}}{P;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ \text{Cond}$$

Figura 4.1: Reglas de Tipado del Cálculo Lambda

$$\frac{}{\Gamma, \tau \vdash \texttt{z} : \tau}\ \text{Base} \qquad \frac{\Gamma \vdash x : \tau_1}{\Gamma, \tau_2 \vdash \texttt{s}\ x : \tau_1}\ \text{Weak}$$

Figura 4.2: Reglas de Tipado de las Variables

No se incluye persistencia entre etapas en el lenguaje. Al igual que en *reFLect* [GMO03], esta decisión fue tomada en base a las observaciones de Taha [Tah99] de que el análisis intensional requiere que ciertas reducciones no sean permitidas en niveles superiores, lo que lleva a una pérdida de confluencia si se incluye persistencia entre etapas.

Las reglas de tipado para los términos que manipulan etapas (Figura 4.3) están inspiradas en la propuesta de Sheard [She05a] para MetaML de utilizar una "banda corrediza" $P;\Gamma;F$ de contextos de tipos. El fragmento lambda liga y busca variables en el contexto del "presente" $\Gamma$. La pila de contextos $P$, del "pasado", contiene a los contextos de las etapas pasadas, que pueden ser accedidas al aplicar escapes al contexto actual. En este caso no se incluye la pila $F$ del "futuro", la cual es innecesaria dado que a la misma se accede al aplicar algún operador de persistencia entre etapas.

$$\frac{(P,\Gamma);\Gamma^f \vdash e : \tau}{P;\Gamma \vdash [|e|]\{\Gamma^f\} : \texttt{cod}^{\Gamma^f}}\ \text{Br} \qquad \frac{P;\Gamma^p \vdash e : \texttt{cod}^{\Gamma}}{(P,\Gamma^p);\Gamma \vdash \$(e)^\tau : \tau}\ \text{Esc}$$

$$\frac{P;\Gamma \vdash e_1 : \texttt{cod}^{(\cdot)} \qquad P;\Gamma \vdash e_2 : \tau}{P;\Gamma \vdash \texttt{run}\ e_1\,|\,e_2 : \tau}\ \text{Run} \qquad \frac{\Gamma' \vdash \Theta \Rightarrow \Gamma'' \qquad P;\Gamma \vdash e : \texttt{cod}^{\Gamma'}}{P;\Gamma \vdash e[\Theta] : \texttt{cod}^{\Gamma''}}\ \text{Subst}$$

Figura 4.3: Reglas de Tipado de la Extensión Multi-etapas

### Tipado Dinámico y Contextos Explícitos

El tipo para las expresiones citadas es $\texttt{cod}^{\Gamma^f}$, donde $\Gamma^f$ es un contexto de tipos que refleja las variables libres de la expresión[3]. Cuando una expresión es citada, un contexto que incluya sus variables libres debe ser anotado explícitamente. Notar que $\Gamma^f$ no tiene por qué ser minimal. Esto significa que, si $\Gamma^p$ representa a todas las variables libres de la expresión citada, $\Gamma^f$ debe cumplir la relación $\Gamma^f = (\Gamma^p, \Gamma^c)$, que significa que todas las variables libres de la expresión ($\Gamma^c$) deben estar en $\Gamma^f$ pero algunas otras ($\Gamma^p$) pueden ser agregadas. Entonces, tanto la expresión `[|#0`$^{\texttt{int}}$`|]{int}` como `[|#0`$^{\texttt{int}}$`|]{bool,int}` cumplen con la regla Br, y, por lo tanto, están bien tipadas.

Una versión de la función generadora de potencias que se mostró como ejemplo a lo largo del capítulo 3 sería la siguiente:

```
val pow =
    let val rec pow':int->cod{x:int} =
        fn n:int =>
        fn x:cod{a:int} =>
                    if (n=0)
                        then < 1 >{a:int}
                        else [| $(x):int * $(pow' (n-1) x):int |]{a:int}
    in
        fn n:int => [| fn a:int => $(pow' n [|a|]{a:int}) |]{}
```

[3]La notación $\Gamma^f$ no expresa nada en especial sobre el contexto. Se trata de un contexto común, salvo que en él tipa una expresión que se evalúa en el "futuro".

```
end
```

Si se aplica esta función, por ejemplo, al literal entero `2`, retornaría la expresión de tipo $\texttt{cod}^{(\cdot)}$:

```
[| fn a:int => a * a * 1 |].
```

A diferencia de la mayoría de los lenguajes multi-etapas el tipo que representa al código no incluye el tipo de la expresión citada, por lo tanto el juicio del escape podría tipar expresiones mal formadas. Por ejemplo, la expresión

```
(fn x:cod{} => [| $(x):int + 1 |]{}) [| true |]{}
```

está bien tipada, dado que el requerimiento de que el código ligado sea de una expresión entera no puede ser verificado estáticamente. El chequeo de tipos de esta clase de expresiones se pospone para realizarse en tiempo de ejecución, y las expresiones mal tipadas evalúan al valor bien tipado `[|fail|]`.

El operador `run` es similar al propuesto por Shields *et al.* [SSP98], donde se realiza un chequeo y unificación de tipos en tiempo de ejecución para decidir si el código puede ser ejecutado. Para evitar que la invocación de `run` se detenga debido a errores en tiempo de ejecución se agrega una nueva expresión, llamada expresión de excepción, que es la que se evalúa en caso de que se detecte un error en la expresión a ejecutar. En la regla Run, el tipo de la expresión citada a ejecutar $e_1$ debe ser el tipo de la expresión de excepción $e_2$. Si su tipo no es el esperado o el chequeo de tipos falla entonces se evalúa la expresión $e_2$. Esta regla asegura que sólo se permita evaluar código cerrado, permitiendo sólo expresiones cuyo tipo sea $\texttt{cod}^{(\cdot)}$, o sea, sin variables libres. Algunos ejemplos de uso de `run`, y sus resultados, son los siguientes:

`run [| 1 + 1 |]{} | 0`
: evalúa a 2.

`run (fn x:cod{} => [| $(x):int + 1 |]{}) [| true |]{} | 0`
: evalúa a 0, porque hay un error de tipos en el código.

`run [| true |]{} | 0`
: evalúa a 0, porque el tipo del código no unifica con el de la expresión de excepción.

`run [| x + 1 |]{x:int} | 0`
: está mal tipada, porque la expresión a ejecutar no es cerrada (tiene tipo $\texttt{cod}^{\cdot,\texttt{int}}$).

Notar que como el contexto anotado en una expresión citada no tiene por qué ser minimal, la restricción de que el tipo de las expresiones a ejecutar sea $\texttt{cod}^{(\cdot)}$ puede excluir la ejecución de algunas expresiones cerradas. Por ejemplo, la expresión `[| 1 + 1 |]{x:int}` no se puede ejecutar, aunque es claro que es cerrada. El control de este tipo de situaciones queda en manos del programador.

### Sustituciones Explícitas

Se incluye un operador de sustitución explícita sobre expresiones citadas, para proveer de una forma simple de captura de variables libres. Se utiliza la notación de $\lambda\nu$ [BBLR96] para las sustituciones, que son barra ($e/$), levantar ($\Uparrow$) y desplazar ($\uparrow$). En este caso se agrega una anotación explícita del nuevo tipo en el desplazamiento ($\uparrow^{\tau}$).

El juicio de tipado para las sustituciones es de la forma $\Gamma \vdash \Theta \Rightarrow \Gamma'$. Relaciona a un contexto de tipos y a una sustitución con un contexto de tipos "resultante". Por lo tanto, una sustitución $\Theta$ sobre una expresión tipada en un contexto local $\Gamma$ resulta en una

$$\frac{P;\Gamma \vdash e : \tau}{\Gamma, \tau \vdash e/ \Rightarrow \Gamma}\ \text{Slash} \qquad \frac{}{\Gamma \vdash \uparrow^{\tau} \Rightarrow \Gamma, \tau}\ \text{Shift} \qquad \frac{\Gamma \vdash \Theta \Rightarrow \Gamma'}{\Gamma, \tau \vdash \Uparrow(\Theta) \Rightarrow \Gamma', \tau}\ \text{Lift}$$

Figura 4.4: Reglas de Tipado de las Sustituciones Explícitas

expresión tipada en un contexto local $\Gamma'$. Las reglas de tipado se muestran en la Figura 4.4.

Dada una expresión $e$ con tipo $\tau$ en una contexto $\Gamma$ bajo cualquier pila de "pasado" $P$, una barra ($e/$) reemplaza a la primer variable por $e$ y decrementa los índices de todas las demás variables en uno. El desplazamiento ($\uparrow^{\tau}$) incrementa los índices de todas las variables en uno y agrega el tipo $\tau$ en el índice 0. Al aplicar levantar ($\Uparrow$), el tipo $\tau$ con índice 0 permanece invariado y la sustitución $\Theta$ se aplica al resto del contexto. Algunos ejemplos de sustituciones explícitas son los siguientes:

$([|(\#0^{\texttt{int}}, \#1^{\texttt{bool}})|]\{\texttt{bool,int}\})[9/]$
evalúa a la expresión citada $[|(9, \#0^{\texttt{bool}})|]$ de tipo $\texttt{cod}^{\cdot,\texttt{bool}}$

$([|(\#0^{\texttt{int}}, \#1^{\texttt{bool}})|]\{\texttt{bool,int}\})[\Uparrow(\texttt{True}/)]$
evalúa a la expresión citada $[|(\#0^{\texttt{int}}, \texttt{True})|]$ de tipo $\texttt{cod}^{\cdot,\texttt{int}}$

$([|(\#0^{\texttt{int}}, \#1^{\texttt{bool}})|]\{\texttt{bool,int}\})[\uparrow^{\texttt{bool}\to\texttt{int}}]$
evalúa a la expresión citada $([|(\#1^{\texttt{int}}, \#2^{\texttt{bool}})|]$ de tipo $\texttt{cod}^{\cdot,\texttt{bool},\texttt{int},\texttt{bool}\to\texttt{int}}$

En la sintaxis a la ML se debe especificar el nombre de la variable a sustituir o agregar. El desplazamiento, o agregado de variables, se denota con el símbolo ^ seguido del nombre y tipo de la nueva variable. La versión en esta sintaxis de los ejemplos anteriores es la siguiente:

```
[| (x , y) |]{x:int,y:bool} [9/x]
[| (x , y) |]{x:int,y:bool} [True/y]
[| (x , y) |]{x:int,y:bool} [^z:bool->int]
```

Notar que la aplicación de la sustitución explícita cambia el tipo de la expresión citada, y que puede ser utilizada tanto para cerrar una expresión como para que sea considerada abierta. Por ejemplo:

`run [| x + 1 |]{x:int}[2/x] | 0`
evalúa a 3.

`run [| 1 + 1 |]{x:int}[2/x] | 0`
evalúa a 2.

`run [| 1 + 1 |]{}[^x:bool] | 0`
está mal tipada, porque la sustitución hace que la expresión a ejecutar no quede cerrada (tiene tipo $\texttt{cod}^{\cdot,\texttt{bool}}$).

### 4.1.3. Extensión para Análisis Intensional

Con el objetivo de proveer análisis intensional de código, el cálculo formulado es extendido con una primitiva de alternación. La misma es similar a la propuesta por Grundy *et al.* [GMO03], donde las variables son ligadas por un mecanismo de concordancia de patrones (*pattern matching*).

| | | | |
|---|---|---|---|
| Términos | $e \in \mathsf{E}$ | ::= | $\ldots \mid \lambda p.\ e_1 \mid e_2$ |
| Patrones | $p \in \mathsf{P}$ | ::= | $i \mid b \mid (p_1, p_2) \mid \bullet^\tau \mid \_ \mid [\mid pc \mid]$ |
| Patrones de Código | $pc \in \mathsf{PC}$ | ::= | $\$(\bullet)^\tau \mid \$(\mathtt{lit})^\tau \mid \_ \mid \mathtt{fail} \mid$ |
| | | | $i \mid b \mid (pc_1, pc_2) \mid \mathtt{fst}\ pc \mid \mathtt{snd}\ pc \mid$ |
| | | | $\lambda^\tau.\ pc \mid \mathtt{fix}^\tau\ pc \mid x^\tau \mid pc_1\ pc_2 \mid$ |
| | | | $\mathtt{if}\ pc_1\ \mathtt{then}\ pc_2\ \mathtt{else}\ pc_3 \mid$ |
| | | | $[\mid \_ \mid]\{\Gamma\} \mid pc[\Theta] \mid \mathtt{run}\ pc_1 \mid pc_2 \mid$ |
| | | | $\lambda p^\tau.\ pc_1 \mid pc_2$ |

La semántica de los patrones se inspira en el mecanismo de concordancia de patrones definido por Pasalic y Linger [PL04]. En dicho trabajo, el juicio para los patrones $\Gamma \vdash p : \tau \Rightarrow \Gamma'$ involucra un contexto de tipos "entrada" $\Gamma$, un patrón $p$, que puede concordar con un valor de tipo $\tau$, y un contexto de tipos "resultado" $\Gamma'$. Este último extiende $\Gamma$ con los tipos de las variables del patrón. En base al hecho de que el único cambio que se puede realizar a un contexto de entrada es su extensión con las variables libres de $p$, y para simplificar la semántica dinámica de las sustituciones sobre alternaciones, se ha omitido el contexto de entrada en el juicio para los patrones. Por lo tanto el juicio es de la forma $\vdash p : \tau \Rightarrow \Gamma$, que significa que un patrón $p$ (concordando a un valor de tipo $\tau$) tiene las variables libres contenidas en $\Gamma$.

Para una alternación de tipo $\tau_1 \to \tau_2$, la regla Alt (Figura 4.5) relaciona a un patrón $p$, el cual debería concordar con un valor de tipo $\tau_1$ extendiendo un contexto con $\Gamma'$, una expresión $e_1$ con tipo $\tau_2$ en un contexto local $\Gamma, \Gamma'$ ($\Gamma$ extendido con $\Gamma'$), y una expresión alternativa $e_2$ con tipo $\tau_1 \to \tau_2$. Si $p$ concuerda con el valor, entonces se evalúa $e_1$ en el contexto local $\Gamma, \Gamma'$, en otro caso se evalúa $e_2$ en el contexto local $\Gamma$ y se aplica al valor al que se quería concordar.

$$\frac{\vdash p : \tau_1 \Rightarrow \Gamma' \qquad P; \Gamma, \Gamma' \vdash e_1 : \tau_2 \qquad P; \Gamma \vdash e_2 : \tau_1 \to \tau_2}{P; \Gamma \vdash \lambda p.\ e_1 \mid e_2 : \tau_1 \to \tau_2}\ \text{Alt}$$

Figura 4.5: Regla de Tipado para la Alternación

El patrón más simple es el patrón **cualquiera** ($\_$) que concuerda cualquier valor de tipo $\tau$ y deja el contexto invariado. Otro patrón básico es el de **ligado de variable** ($\bullet^\tau$), que difiere del anterior en la anotación del tipo y en que extiende el contexto ligando el valor concordado. Un patrón podría ligar a más de una variable. La regla PPair muestra cómo las variables se relacionan a los índices del contexto resultante; dado un patrón **par** $(p_1, p_2)$, donde $p_1$ y $p_2$ se relacionan con $\Gamma'$ y $\Gamma''$ respectivamente, sus variables libres son $\Gamma', \Gamma''$, y entonces, las variables del sub-patrón de más a la derecha ($p_2$) serán las de menor índice en el contexto. Por ejemplo, el patrón $(\bullet^{\mathtt{int}}, \bullet^{\mathtt{bool}})$ concuerda con cualquier valor

de tipo `(int,bool)` y tiene las variables libres contenidas en el contexto $\cdot, \texttt{int}, \texttt{bool}$. Éste puede ser considerada como un principio general para los patrones con múltiples variables.

$$\frac{}{\vdash \_ : \tau \Rightarrow \cdot}\ \text{PAny} \qquad \frac{}{\vdash \bullet^{\tau} : \tau \Rightarrow \cdot, \tau}\ \text{PVar} \qquad \frac{}{\vdash i : \texttt{int} \Rightarrow \cdot}\ \text{PLInt} \qquad \frac{}{\vdash b : \texttt{bool} \Rightarrow \cdot}\ \text{PLBool}$$

$$\frac{\vdash p_1 : \tau_1 \Rightarrow \Gamma' \quad \vdash p_2 : \tau_2 \Rightarrow \Gamma''}{\vdash (p_1, p_2) : \tau_1 \times \tau_2 \Rightarrow \Gamma', \Gamma''}\ \text{PPair} \qquad \frac{\Gamma^f \Vdash pc \Rightarrow \Gamma'}{\vdash \texttt{[|}pc\texttt{|]} : \texttt{cod}^{\Gamma f} \Rightarrow \Gamma'}\ \text{PCod}$$

Figura 4.6: Reglas de Tipado de los Patrones

El análisis de código es entonces llevado a cabo con la ayuda de los **patrones de código**. Sus reglas de tipado se muestran en la Figura 4.7. El juicio $\Gamma^f \Vdash pc \Rightarrow \Gamma'$ expresa que un patrón $pc$, que podría concordar con un código de tipo $\texttt{cod}^{\Gamma f}$, tiene las variables contenidas en $\Gamma'$.

La mayoría de los patrones de código consisten en descomponer una expresión y aplicar otros patrones de código a las sub-expresiones resultantes. El patrón de código **cualquier código**(`_`) concuerda cualquier código, mientras que **falla** (`fail`) concuerda solamente código cuyo chequeo de tipo dinámico haya fallado. Ambos patrones dejan el contexto invariado.

La sintaxis de los patrones **ligado de variable** ($\texttt{\$(}\bullet\texttt{)}^{\tau}$) y **ligado de literal** ($\texttt{\$(lit)}^{\tau}$) sugiere su semántica, en el sentido de que sólo concuerda con expresiones que al ser escapadas tienen tipo $\tau$. Siempre que se haya cumplido la restricción de que el primero concuerda cualquier valor, mientras que el segundo concuerda solamente expresiones literales citadas. Ambos patrones extienden el contexto con los valores de código concordados.

El patrón **variable** ($x^{\tau}$) se comporta como los **constantes literales** ($i$ y $b$). Concuerda con un código cuya expresión citada es exactamente la variable $x^{\tau}$, y no cambia el contexto. El patrón **cualquier paréntesis** ($\texttt{[|\_|]}\{\Gamma^{ff}\}$) concuerda con cualquier código cuya expresión citada tenga la forma $\texttt{[|}e\texttt{|]}$ y tenga como variables libres $\Gamma^{ff}$. Dada una sustitución explícita citada $e[\Theta']$, el patrón de código **sustitución** ($pc[\Theta]$) requiere que $\Theta$ sea igual a $\Theta'$ y concuerda el patrón de código $pc$ con el código de $e$.

En los patrones de **punto fijo** ($\texttt{fix}^{\tau}.pc$) y **alternación** ($\lambda p^{\tau}.pc_1 \,|\, pc_2$), a diferencia que en la sintaxis de las expresiones de este tipo, se agregan anotaciones de tipo para poder reconstruir los tipos de los sub-patrones. Para que un código concuerde con el mismo debe ser una alternación exactamente de esa forma, cuyas ramas concuerden con los sub-patrones. El hecho de exigir que los patrones sean idénticos puede disminuir la expresividad del análisis del código de alternación. Se deja para un trabajo futuro agregar construcciones de concordancia de patrones sobre la clase de los patrones, tratando de superar los problemas de circularidad que esto conlleva[4].

Un ejemplo de análisis de código es el siguiente:

$$\begin{array}{l}\lambda\texttt{[|if }\#0^{\texttt{bool}}\texttt{ then \$(}\bullet\texttt{)}^{\texttt{int}}\texttt{ else \$(}\bullet\texttt{)}^{\texttt{int}}\texttt{|]}\ .\\ \qquad \#1^{\texttt{cod}^{\cdot,\texttt{bool}}}\texttt{[true/]}\\ \quad \mid\ \lambda^{\texttt{cod}^{\cdot,\texttt{bool}}}.\ \ \#0^{\texttt{cod}^{\cdot,\texttt{bool}}}\texttt{[true/]}\end{array}$$

[4]Como indica Nanevski [Nan04], cualquier agregado requiere de extensiones a la concordancia de patrones contra esos agregados, las cuales requieren ellas mismas nuevas extensiones y así infinitamente.

$$\frac{}{\Gamma^f \Vdash \_ \Rightarrow \cdot}\ \text{PCPAny} \qquad \frac{}{\Gamma^f \Vdash \texttt{fail} \Rightarrow \cdot}\ \text{PCPFail} \qquad \frac{}{\Gamma^f \Vdash i \Rightarrow \cdot}\ \text{PCLInt} \qquad \frac{}{\Gamma^f \Vdash b \Rightarrow \cdot}\ \text{PCLBool}$$

$$\frac{}{\Gamma^f \Vdash \texttt{\$(}\bullet\texttt{)}^\tau \Rightarrow \cdot, \texttt{cod}^{\Gamma^f}}\ \text{PCPVar} \qquad \frac{}{\Gamma^f \Vdash \texttt{\$(lit)}^\tau \Rightarrow \cdot, \texttt{cod}^{\Gamma^f}}\ \text{PCPLit}$$

$$\frac{\Gamma^f \Vdash pc_1 \Rightarrow \Gamma' \quad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma''}{\Gamma^f \Vdash (pc_1, pc_2) \Rightarrow \Gamma', \Gamma''}\ \text{PCPair} \qquad \frac{\Gamma^f \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash \texttt{fst}\ \ pc \Rightarrow \Gamma'}\ \text{PCFst} \qquad \frac{\Gamma^f \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash \texttt{snd}\ \ pc \Rightarrow \Gamma'}\ \text{PCSnd}$$

$$\frac{\Gamma^f, \tau \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash \lambda^\tau.pc \Rightarrow \Gamma'}\ \text{PCAbs} \qquad \frac{\Gamma^f, \tau \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash \texttt{fix}^\tau\ .pc \Rightarrow \Gamma'}\ \text{PCFix}$$

$$\frac{\Gamma^f \Vdash pc_1 \Rightarrow \Gamma' \qquad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma''}{\Gamma^f \Vdash pc_1\ pc_2 \Rightarrow \Gamma', \Gamma''}\ \text{PCApp} \qquad \frac{}{\Gamma^f \Vdash x^\tau \Rightarrow \cdot}\ \text{PCVar}$$

$$\frac{\Gamma^f \Vdash pc_1 \Rightarrow \Gamma' \quad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma'' \qquad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma'''}{\Gamma^f \Vdash \texttt{if}\ \ pc_1\ \ \texttt{then}\ \ pc_2\ \ \texttt{else}\ \ pc_3 \Rightarrow \Gamma', \Gamma'', \Gamma'''}\ \text{PCCond}$$

$$\frac{}{\Gamma^f \Vdash \texttt{[|\_|]}\{\Gamma^{ff}\} \Rightarrow \cdot}\ \text{PCBr} \qquad \frac{\Gamma^f \Vdash pc \Rightarrow \Gamma'}{\Gamma^f \Vdash pc[\Theta] \Rightarrow \Gamma'}\ \text{PCSubst}$$

$$\frac{\Gamma^f \Vdash pc_1 \Rightarrow \Gamma' \qquad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma''}{\Gamma^f \Vdash \texttt{run}\ \ pc_1 \,|\, pc_2 \Rightarrow \Gamma', \Gamma''}\ \text{PCRun}$$

$$\frac{\vdash p : \tau \Rightarrow \Gamma^p \qquad \Gamma^f, \Gamma^p \Vdash pc_1 \Rightarrow \Gamma' \qquad \Gamma^f \Vdash pc_2 \Rightarrow \Gamma''}{\Gamma^f \Vdash \lambda p^\tau.pc_1 \,|\, pc_2 \Rightarrow \Gamma', \Gamma''}\ \text{PCAlt}$$

Figura 4.7: Reglas de Tipado de los Patrones de Inspección de Código

Esta expresión toma un valor de código con tipo $\texttt{cod}^{\cdot,\texttt{bool}}$ (esto es, con una variable libre booleana) y retorna otro con tipo $\texttt{cod}^{(\cdot)}$, resultante de sustituir en el código la variable libre por el literal `true`. Si el código pasado es una cita de una expresión entera `if-then-else`, que tiene como condición la variable libre a sustituir, se optimiza el código devolviendo sólo la sub-expresión `then` con el literal `true` en cada ocurrencia de la variable libre $\#0^{\texttt{bool}}$. En este ejemplo se podría haber sustituido el patrón variable que liga a la sub-expresión `else` por un patrón cualquier código (`_`), dado que esta no se utiliza. Se incluyó el segundo patrón variable para mostrar de que forma se ligan las concordancias en los patrones. Notar que la subexpresión `then` se liga en el número #1 y la subexpresión `else` en el #0.

A continuación se reescribe el ejemplo anterior con la sintaxis tipo SML, con la diferencia de que se cambia el patrón variable por un patrón cualquier código (`_`) en la subexpresión `else`.

```
val sust = fn [| if x then $(et:cod{x:bool}):int else _ |]
            => et [true/x]
            |  fn e:cod{x:bool} => e [true/x]
```

Para los patrones de ligado de código (tanto de variable como de literal) se anotan nombres para las variables libres del código concordado, de manera de poder utilizarlas luego. Por ejemplo, cuando se concuerda la expresión `if-then-else`, a la variable libre se la llama `x`, que es a la cual se le aplica la sustitución. Algunos ejemplos de aplicaciones de esta función y sus resultados son:

`sust [| if x then fst (2,x) else 3 |]{x:bool}`
evalúa a `[| fst (2,true) |]`.

`sust [| x |]{x:bool}`
evalúa a `[| true |]`.

`sust [| if a then 1 else 2 |]{a:bool}`
evalúa a `[| 1 |]`.

`sust [| if true then fst (2,x) else 3 |]{x:bool}`
evalúa a `[| if true then fst (2,true) else 3|]`.

Una versión en este lenguaje del ejemplo del generador de potencias de funciones del capítulo 3 es la siguiente:

```
val fpow =
    fn [| fn a:int => $(f:cod{a:int}):int |]
    =>  let val rec fpow':int->cod{x:int} =
            fn n:int =>
            fn x:cod{a:int} =>
                        if (n=0)
                            then [| 1 |]{a:int}
                            else [| $(x):int * $(fpow' (n-1) x):int |]{a:int}
        in
            fn n:int => [| fn a:int => $(fpow' n f:cod{a:int}) |]{}
        end
    |   fn x:cod{} => x
```

## 4.2. Semántica Operacional

En esta sección se describirá la semántica operacional del lenguaje como una semántica **natural** o de **paso-largo** (*big-step*), es decir, mediante reglas de inferencia que describen la relación directa entre las expresiones y los valores a los que evalúan.

Para esto se debe definir un conjunto de expresiones evaluables $\mathsf{E}$, un conjunto de valores $\mathsf{V}$, los cuales no tienen por qué ser un subconjunto de $\mathsf{E}$, y una relación de evaluación $\hookrightarrow \subseteq \mathsf{E} \times \mathsf{V}$. El hecho que una expresión $e$ evalúe a un valor $v$ se denota $e \hookrightarrow v$.

En nuestro caso, para evitar por ejemplo que $\$(e)^\tau$ se evalúe en el nivel 0, se debe aplicar una clasificación más fina de las expresiones. Para esto se define una **familia de expresiones**, que es una colección de conjuntos $\mathsf{E}^0, \mathsf{E}^1, \mathsf{E}^2$,etc. indexada por números naturales, que indican la etapa en la que ese conjunto de expresiones es válido.

$$
\begin{array}{lcl}
e^0 \in \mathsf{E}^0 & ::= & b \mid i \mid \texttt{(}e^0\texttt{,}e^0\texttt{)} \mid \texttt{fst}\ e^0 \mid \texttt{snd}\ e^0 \mid \\
 & & \lambda^\tau.\ e^0 \mid \texttt{fix}\ e^0 \mid x^\tau \mid e^0\ e^0 \mid \\
 & & \texttt{if}\ e^0\ \texttt{then}\ e^0\ \texttt{else}\ e^0 \mid \\
 & & \texttt{[|}e^1\texttt{|]\{}\Gamma\texttt{\}} \mid e^0[\Theta] \mid \texttt{run}\ e^0\texttt{|}e^0 \mid \lambda p.\ e^0\texttt{|}e^0 \\
e^{n+1} \in \mathsf{E}^{n+1} & ::= & b \mid i \mid \texttt{(}e^{n+1}\texttt{,}e^{n+1}\texttt{)} \mid \texttt{fst}\ e^{n+1} \mid \texttt{snd}\ e^{n+1} \mid \\
 & & \lambda^\tau.\ e^{n+1} \mid \texttt{fix}\ e^{n+1} \mid x^\tau \mid e^{n+1}\ e^{n+1} \mid \\
 & & \texttt{if}\ e^{n+1}\ \texttt{then}\ e^{n+1}\ \texttt{else}\ e^{n+1} \mid \\
 & & \texttt{[|}e^{n+2}\texttt{|]\{}\Gamma\texttt{\}} \mid e^{n+1}[\Theta] \mid \texttt{run}\ e^{n+1}\texttt{|}e^{n+1} \mid \lambda p.\ e^{n+1}\texttt{|}e^{n+1} \mid \$(e^n)^\tau
\end{array}
$$

Como se puede ver en su definición, las expresiones evaluables en el nivel cero se diferencian de las de niveles superiores en que estas últimas incluyen al escape.

Los valores representan los resultados de las computaciones. Debido a la naturaleza de los paréntesis y los escapes, para definir a los valores se debe volver a utilizar una familia de conjuntos.

$$
\begin{array}{lcl}
v^0 \in \mathsf{V}^0 & ::= & b \mid i \mid \texttt{(}v^0\texttt{,}v^0\texttt{)} \mid \lambda^\tau.\ e^0 \mid \\
 & & \texttt{[|}v^1\texttt{|]\{}\Gamma\texttt{\}} \mid \lambda p.\ e^0\texttt{|}e^0 \\
v^1 \in \mathsf{V}^1 & ::= & b \mid i \mid \texttt{(}v^1\texttt{,}v^1\texttt{)} \mid \lambda^\tau.\ v^1 \mid \texttt{fst}\ v^1 \mid \texttt{snd}\ v^1 \mid \texttt{fix}\ v^1 \mid \\
 & & x^\tau \mid v^1\ v^1 \mid \texttt{if}\ v^1\ \texttt{then}\ v^1\ \texttt{else}\ v^1 \mid \\
 & & \texttt{[|}v^2\texttt{|]\{}\Gamma\texttt{\}} \mid \lambda p.\ v^1\texttt{|}v^1 \mid v^1[\Theta] \mid \texttt{run}\ v^1\texttt{|}v^1 \mid \\
 & & \texttt{fail} \\
v^{n+2} \in \mathsf{V}^{n+2} & ::= & b \mid i \mid \texttt{(}v^{n+2}\texttt{,}v^{n+2}\texttt{)} \mid \lambda^\tau.\ v^{n+2} \mid \texttt{fst}\ v^{n+2} \mid \texttt{snd}\ v^{n+2} \mid \texttt{fix}\ v^{n+2} \mid \\
 & & x^\tau \mid v^{n+2}\ v^{n+2} \mid \texttt{if}\ v^{n+2}\ \texttt{then}\ v^{n+2}\ \texttt{else}\ v^{n+2} \mid \\
 & & \texttt{[|}v^{n+3}\texttt{|]\{}\Gamma\texttt{\}} \mid \lambda p.\ v^{n+2}\texttt{|}v^{n+2} \mid v^{n+2}[\Theta] \mid \texttt{run}\ v^{n+2}\texttt{|}v^{n+2} \mid \\
 & & \texttt{fail} \mid \$(v^{n+1})^\tau
\end{array}
$$

Los valores en el nivel 0 son el resultado de evaluar un término en el nivel 0. Pueden ser una constante, un par, una lambda-abstracción, una alternación o un valor entre paréntesis. Esto quiere decir que un valor en el nivel 0 puede ser tanto un valor de los usuales del cálculo lambda como un término que representa código.

Los valores en niveles superiores son el resultado de reconstruir un término en ese nivel. Dado que los valores en niveles superiores representan código, que son expresiones cuya evaluación se ha diferido, los valores en estos niveles son casi idénticos a las expresiones correspondientes, con la excepción del valor especial `fail` y de que en el nivel 1 no hay escapes. Entonces $\mathsf{V}^1 = \mathsf{E}^0 \cup \{\texttt{fail}\}$.

En las siguientes sub-secciones se presentará la relación $\overset{n}{\hookrightarrow} \subseteq \mathsf{E}^n \times \mathsf{V}^n$, entendida como "en el nivel $n$ evalúa a", definiendo así una relación de evaluación diferenciada por nivel. En el apéndice A se resumen todas las notaciones utilizadas para especificar la evaluación.

### 4.2.1. Cálculo Básico

La semántica de paso-largo para el cálculo básico se describe en la Figura 4.8. Las mismas están definidas en el nivel 0 y corresponden a la semántica de un cálculo lambda con pasaje de parámetros por valor.

$$b \overset{0}{\hookrightarrow} b \quad i \overset{0}{\hookrightarrow} i \quad \frac{e_1 \overset{0}{\hookrightarrow} v_1 \quad e_2 \overset{0}{\hookrightarrow} v_2}{(e_1, e_2) \overset{0}{\hookrightarrow} (v_1, v_2)} \quad \frac{e \overset{0}{\hookrightarrow} (v_1, v_2)}{\texttt{fst}\ e \overset{0}{\hookrightarrow} v_1} \quad \frac{e \overset{0}{\hookrightarrow} (v_1, v_2)}{\texttt{snd}\ e \overset{0}{\hookrightarrow} v_2}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \texttt{true} \quad e_2 \overset{0}{\hookrightarrow} v_2}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \overset{0}{\hookrightarrow} v_2} \quad \frac{e_1 \overset{0}{\hookrightarrow} \texttt{false} \quad e_3 \overset{0}{\hookrightarrow} v_3}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \overset{0}{\hookrightarrow} v_3}$$

$$\lambda^\tau.e \overset{0}{\hookrightarrow} \lambda^\tau.e \quad \frac{e[\texttt{fix}\ e/] \overset{0}{\hookrightarrow} v}{\texttt{fix}\ e \overset{0}{\hookrightarrow} v} \quad \frac{e_1 \overset{0}{\hookrightarrow} \lambda^\tau.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[v_1/] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2}$$

Figura 4.8: Evaluación del Cálculo Básico

Las reglas para enteros, booleanos y lambda-abstracciones evalúan a sí mismos, dado que pertenecen a $\mathsf{V}^0$, y por lo tanto son expresiones canónicas en el nivel 0. En la regla para la aplicación se evalúa el operador, para obtener una lambda-abstracción, y se evalúa el operando, para obtener un valor, que luego se sustituye por la primera variable (#0) del cuerpo de la lambda-abstracción[5]. El resultado de evaluar la expresión sustituída es el resultado de la evaluación. La evaluación del constructor de pares consiste en el par que contiene el resultado de la evaluación de las dos expresiones. En las proyecciones de pares se evalúa la expresión para obtener un par y se devuelve su proyección.

En la evaluación de la condición se evalúa la primera o la segunda rama dependiendo del resultado de evaluar la condición. En el caso del punto fijo se debe seguir una estrategia perezosa, evaluando $e[\texttt{fix}\ e/]$, sin una evaluación previa de $\texttt{fix}\ e$, para evitar que se evalúe $\texttt{fix}\ e$ infinitamente.

Por ejemplo, la evaluación de la expresión

$(\texttt{fix}\ (\lambda^{\texttt{int}}.\texttt{if}\ \#0^{\texttt{int}} = 0\ \texttt{then}\ 1\ \texttt{else}\ (1 + \#1^{\texttt{int}\to\texttt{int}}\ (\#0^{\texttt{int}} - 1))))\ 1$

procedería del siguiente modo (llamaremos $e$ al cuerpo de la expresión fix):

$(\texttt{fix}\ e)\ 1$

$\overset{0}{\hookrightarrow}$ {*regla de evaluación del punto fijo*}

$e[\texttt{fix}\ e/]\ 1$

$\Rightarrow$ {*sustitución*}

[5]Dado que las ligaduras de variables se codifican con índices de de Bruijn, en la sustitución no se nombra la variable a sustituir. Más adelante se explicará una sintaxis para el caso de la sustitución explícita sobre trozos de código de este tipo. En este caso siempre se sustituye la primera variable (#0).

$(\lambda^{\texttt{int}}.\texttt{if}\ \#0^{\texttt{int}} = 0\ \texttt{then}\ 1\ \texttt{else}\ (1 + (\texttt{fix}\ e))\ (\#0^{\texttt{int}} - 1)))\ 1$
$\overset{0}{\hookrightarrow}$ {*regla de evaluación de la aplicación*}
$\texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ (1 + (\texttt{fix}\ e))\ (1 - 1))$
$\overset{0}{\hookrightarrow}$ {*evaluación de* `=`, `-` *y regla de evaluación del* `if-then-else`}
$1 + (\texttt{fix}\ e)\ 0$
$\overset{0}{\hookrightarrow}$ {*regla de evaluación del punto fijo*}
$1 + e[\texttt{fix}\ e/]\ 0$
$\Rightarrow$ {*sustitución*}
$1 + ((\lambda^{\texttt{int}}.\texttt{if}\ \#0^{\texttt{int}} = 0\ \texttt{then}\ 1\ \texttt{else}\ (1 + (\texttt{fix}\ e))\ (\#0^{\texttt{int}} - 1)))\ 0)$
$\overset{0}{\hookrightarrow}$ {*regla de evaluación de la aplicación*}
$1 + \texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ (1 + (\texttt{fix}\ e))\ (0 - 1))$
$\overset{0}{\hookrightarrow}$ {*evaluación de* `=` *y regla de evaluación del* `if-then-else`}
$1 + 1$
$\overset{0}{\hookrightarrow}$ {*evaluación de* `+`}
2.

### 4.2.2. Extensión Multi-etapas

En la Figura 4.9 se muestra la evaluación en nivel 0 de la extensión multi-etapas del lenguaje. Se presentan reglas de evaluación para `run`, los paréntesis y la sustitución explícita, pero no para el escape debido a que el mismo no es válido en el nivel 0, y por lo tanto no está incluido en $\mathsf{E}^0$.

$$\frac{e_1 \overset{0}{\hookrightarrow} [\,|\,v\,|\,]\{\} \qquad v \overset{0}{\hookrightarrow} v_1}{\texttt{run}\ e_1\,|\,e_2 \overset{0}{\hookrightarrow} v_1}(v \neq \texttt{fail}) \wedge (type(v) = type(e_2))$$

$$\frac{e_1 \overset{0}{\hookrightarrow} [\,|\,v\,|\,]\{\} \qquad e_2 \overset{0}{\hookrightarrow} v_2}{\texttt{run}\ e_1\,|\,e_2 \overset{0}{\hookrightarrow} v_2}(v = \texttt{fail}) \vee (type(v) \neq type(e_2))$$

$$\frac{e \overset{1}{\hookrightarrow} v}{[\,|\,e\,|\,]\{\Gamma\} \overset{0}{\hookrightarrow} [\,|\,v\,|\,]\{\Gamma\}}$$

$$\frac{e \overset{0}{\hookrightarrow} [\,|\,v_1\,|\,]\{\Gamma_1\} \qquad \Gamma_1 \vdash \Theta : v_1 \Longrightarrow v_2 \qquad \Gamma_1 \vdash \Theta \Rightarrow \Gamma_2}{e[\Theta] \overset{0}{\hookrightarrow} [\,|\,v_2\,|\,]\{\Gamma_2\}}$$

Figura 4.9: Evaluación de la Extensión Multi-etapas

**Evaluación de Código**

Para la evaluación del `run` se evalúa la primera expresión. Si su resultado no falla en el chequeo de tipos y el tipo de la expresión citada unifica con el de la expresión de excepción, entonces esta expresión $v$ pertenece a $\mathsf{E}^0$ (no es `fail`). Por lo tanto el resultado es el de la evaluación de $v$ en el nivel 0. En otro caso, si el resultado de evaluar la primera expresión es `[|fail|]` o no unifican los tipos, se evalúa la expresión de excepción.

En el siguiente ejemplo se ejecuta la expresión citada 9. Dado que su tipo coincide con el de la expresión de excepción y su evaluación no es `fail`, el resultado de la evaluación es 9.

$$\frac{\dfrac{9 \overset{1}{\hookrightarrow} 9}{\texttt{[|9|]}\{\} \overset{0}{\hookrightarrow} \texttt{[|9|]}\{\}} \qquad 9 \overset{0}{\hookrightarrow} 9}{\texttt{run [|9|]}\{\}\texttt{|0} \overset{0}{\hookrightarrow} 9}(9 \neq \texttt{fail}) \wedge (\mathit{type}(\texttt{9}) = \mathit{type}(\texttt{0}))$$

En este otro ejemplo se intenta ejecutar la expresión citada `true`. Dado que el tipo no coincide con el de la expresión de excepción, es esta última la que se evalúa para obtener el resultado de la expresión.

$$\frac{\dfrac{\texttt{true} \overset{1}{\hookrightarrow} \texttt{true}}{\texttt{[|true|]}\{\} \overset{0}{\hookrightarrow} \texttt{[|true|]}\{\}} \qquad 0 \overset{0}{\hookrightarrow} 0}{\texttt{run [|true|]}\{\}\texttt{|0} \overset{0}{\hookrightarrow} 0}(\mathit{type}(\texttt{true}) \neq \mathit{type}(\texttt{0}))$$

La forma de evaluación de las expresiones entre paréntesis, y el chequeo dinámico de tipos que se realiza durante la misma, se verá en la siguiente sub-sección.

**Chequeo Dinámico de Tipos y Construcción de Código**

Al evaluar una expresión entre paréntesis $\texttt{[|}e\texttt{|]}\{\Gamma\}$, se debe construir un código a partir de la expresión $e$, quitando los escapes en nivel 1 y realizando el chequeo de tipos de la expresión resultante. Esta construcción de código se realiza con las funciones de evaluación en niveles superiores $\overset{n+1}{\hookrightarrow}$, que se describen en las Figuras 4.10 y 4.11.

En el caso del cálculo básico (Figura 4.10) se atraviesa la expresión generando el código de forma recursiva. Si alguna sub-expresión evalúa a `fail`, toda la expresión evalúa también a `fail`. Por ejemplo, al evaluar el par $(e_1, e_2)$ en el nivel $n+1$, se realiza la evaluación de $e_1$ y $e_2$ en el mismo nivel. Si evalúan a los códigos $v_1$ y $v_2$ respectivamente, el resultado es el par $(v_1, v_2)$, perteneciente a $\mathsf{V}^{n+1}$. Si $e_1$ ó $e_2$ evalúa a `fail`, entonces el resultado total es `fail`.

Para las extensiones del lenguaje (Figura 4.11), los casos de alternación, `run` y sustitución explícita son iguales al cálculo básico. En la sustitución explícita, el patrón queda invariado al reconstruir el código. Para el caso de $\texttt{[|}e\texttt{|]}\{\Gamma\}$ se evalúa la expresión citada $e$ en el siguiente nivel.

Cuando ocurre un escape en el nivel uno, éste se debe sustituir por el resultado de evaluar su expresión en el nivel 0, si el mismo es válido y su tipo es el requerido, o por `fail` en otro caso. Es sólo en este nivel en donde se realiza el chequeo de tipos dinámico. Por ejemplo, la expresión $\texttt{[|\$([|true|]\{\})}^{\texttt{int}}\texttt{|]}\{\}$ evalúa a `[|fail|]{}`, a través de la siguiente cadena de evaluaciones:

$$b \overset{n+1}{\hookrightarrow} b \qquad i \overset{n+1}{\hookrightarrow} i \qquad x \overset{n+1}{\hookrightarrow} x$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{(e_1, e_2) \overset{n+1}{\hookrightarrow} (v_1, v_2)} \begin{array}{l}(v_1 \neq \texttt{fail}) \\ \wedge (v_2 \neq \texttt{fail})\end{array} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{(e_1, e_2) \overset{n+1}{\hookrightarrow} \texttt{fail}} \begin{array}{l}(v_1 = \texttt{fail}) \\ \vee (v_2 = \texttt{fail})\end{array}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{fst}\ e \overset{n+1}{\hookrightarrow} \texttt{fst}\ v}(v \neq \texttt{fail}) \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{fst}\ e \overset{n+1}{\hookrightarrow} \texttt{fail}}(v = \texttt{fail})$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{snd}\ e \overset{n+1}{\hookrightarrow} \texttt{snd}\ v}(v \neq \texttt{fail}) \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{snd}\ e \overset{n+1}{\hookrightarrow} \texttt{fail}}(v = \texttt{fail})$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2 \quad e_3 \overset{n+1}{\hookrightarrow} v_3}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \overset{n+1}{\hookrightarrow} \texttt{if}\ v_1\ \texttt{then}\ v_2\ \texttt{else}\ v_3} \begin{array}{l}(v_1 \neq \texttt{fail}) \wedge (v_2 \neq \texttt{fail}) \\ \wedge (v_3 \neq \texttt{fail})\end{array}$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2 \quad e_3 \overset{n+1}{\hookrightarrow} v_3}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \overset{n+1}{\hookrightarrow} \texttt{fail}}(v_1 = \texttt{fail}) \vee (v_2 = \texttt{fail}) \vee (v_3 = \texttt{fail})$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\lambda^\tau.e \overset{n+1}{\hookrightarrow} \lambda^\tau.v}(v \neq \texttt{fail}) \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\lambda^\tau.e \overset{n+1}{\hookrightarrow} \texttt{fail}}(v = \texttt{fail})$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{fix}\ e \overset{n+1}{\hookrightarrow} \texttt{fix}\ v}(v \neq \texttt{fail}) \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{fix}\ e \overset{n+1}{\hookrightarrow} \texttt{fail}}(v = \texttt{fail})$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2} \begin{array}{l}(v_1 \neq \texttt{fail}) \\ \wedge (v_2 \neq \texttt{fail})\end{array} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} \texttt{fail}} \begin{array}{l}(v_1 = \texttt{fail}) \\ \vee (v_2 = \texttt{fail})\end{array}$$

Figura 4.10: Construcción de Código del Cálculo Básico

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{\lambda p.\ e_1 \,|\, e_2 \overset{n+1}{\hookrightarrow} \lambda p.\ v_1 \,|\, v_2} \begin{array}{l}(v_1 \neq \texttt{fail}) \\ \wedge (v_2 \neq \texttt{fail})\end{array} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{\lambda p.\ e_1 \,|\, e_2 \overset{n+1}{\hookrightarrow} \texttt{fail}} \begin{array}{l}(v_1 = \texttt{fail}) \\ \vee (v_2 = \texttt{fail})\end{array}$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{\texttt{run}\ e_1 \,|\, e_2 \overset{n+1}{\hookrightarrow} \texttt{run}\ v_1 \,|\, v_2} \begin{array}{l}(v_1 \neq \texttt{fail}) \\ \wedge (v_2 \neq \texttt{fail})\end{array} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{\texttt{run}\ e_1 \,|\, e_2 \overset{n+1}{\hookrightarrow} \texttt{fail}} \begin{array}{l}(v_1 = \texttt{fail}) \\ \vee (v_2 = \texttt{fail})\end{array}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{e[\Theta] \overset{n+1}{\hookrightarrow} v[\Theta]} (v \neq \texttt{fail}) \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{e[\Theta] \overset{n+1}{\hookrightarrow} \texttt{fail}} (v = \texttt{fail})$$

$$\frac{e \overset{n+2}{\hookrightarrow} v}{\texttt{[}\,|\,e\,|\,\texttt{]}\{\Gamma\} \overset{n+1}{\hookrightarrow} \texttt{[}\,|\,v\,|\,\texttt{]}\{\Gamma\}} (v \neq \texttt{fail}) \qquad \frac{e \overset{n+2}{\hookrightarrow} v}{\texttt{[}\,|\,e\,|\,\texttt{]}\{\Gamma\} \overset{n+1}{\hookrightarrow} \texttt{fail}} (v = \texttt{fail})$$

$$\frac{e \overset{0}{\hookrightarrow} \texttt{[}\,|\,v\,|\,\texttt{]}\{\Gamma\}}{\$(e)^\tau \overset{1}{\hookrightarrow} v} \begin{array}{l}(v \neq \texttt{fail}) \\ \wedge (type(v) = \tau)\end{array} \qquad \frac{e \overset{0}{\hookrightarrow} \texttt{[}\,|\,v\,|\,\texttt{]}\{\Gamma\}}{\$(e)^\tau \overset{1}{\hookrightarrow} \texttt{fail}} \begin{array}{l}(v = \texttt{fail}) \\ \vee (type(v) \neq \tau)\end{array}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\$(e)^\tau \overset{n+2}{\hookrightarrow} \$(v)^\tau} (v \neq \texttt{fail}) \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\$(e)^\tau \overset{n+2}{\hookrightarrow} \texttt{fail}} (v = \texttt{fail})$$

Figura 4.11: Construcción de Código de las Extensiones Multi-etapas y Análisis Intensional

$$\dfrac{\dfrac{\dfrac{\texttt{true} \overset{1}{\hookrightarrow} \texttt{true}}{[|\texttt{true}|]\{\} \overset{0}{\hookrightarrow} [|\texttt{true}|]\{\}}}{\$([|\texttt{true}|]\{\})^{\texttt{int}} \overset{1}{\hookrightarrow} \texttt{fail}}\, type(\texttt{true}) \neq \texttt{int}}{[|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\} \overset{0}{\hookrightarrow} [|\texttt{fail}|]\{\}}$$

en donde, al evaluar $\$([|\texttt{true}|]\{\})^{\texttt{int}}$ en el nivel 1, no se cumple la condición de que el tipo de `true` sea `int`. Por otro lado, si a la expresión anterior se la rodea con paréntesis, elevando así su nivel, la evaluación de la expresión $[|[|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\}|]\{\}$, resultará en $[|[|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\}|]\{\}$. Como se muestra en la siguiente cadena de evaluaciones, en este caso la evaluación del escape se realiza en el nivel 2. En este nivel no se sustituyen los escapes y por lo tanto no se realiza chequeo de tipos.

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{true} \overset{2}{\hookrightarrow} \texttt{true}}{[|\texttt{true}|]\{\} \overset{1}{\hookrightarrow} [|\texttt{true}|]\{\}}}{\$([|\texttt{true}|]\{\})^{\texttt{int}} \overset{2}{\hookrightarrow} \$([|\texttt{true}|]\{\})^{\texttt{int}}}}{[|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\} \overset{1}{\hookrightarrow} [|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\}}}{[|[|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\}|]\{\} \overset{0}{\hookrightarrow} [|[|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\}|]\{\}}$$

De esta forma se realiza el chequeo de tipos de las expresiones citadas por etapas sólo cuando es necesario. Se puede ver fácilmente que si se intenta ejecutar esta última expresión, invocando, por ejemplo, $\texttt{run}\ [|[|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\}|]\{\}\ |\ [|0|]\{\}$, ahora sí se realizará el chequeo de tipos y el resultado de la evaluación será $[|\texttt{fail}|]\{\}$. Esto se debe a que el `run` provoca que la subexpresión $[|\$([|\texttt{true}|]\{\})^{\texttt{int}}|]\{\}$ se evalúe en el nivel 0, llevando a la situación del primer ejemplo.

Es importante notar que la expresión $[|[|\$(\$([|\texttt{true}|]\{\})^{\texttt{cod}^{(\cdot)}})^{\texttt{int}}|]\{\}|]\{\}$ también evalúa a $[|\texttt{fail}|]\{\}$, como se muestra a continuación:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{true} \overset{1}{\hookrightarrow} \texttt{true}}{[|\texttt{true}|]\{\} \overset{0}{\hookrightarrow} [|\texttt{true}|]\{\}}}{\$([|\texttt{true}|]\{\})^{\texttt{cod}^{(\cdot)}} \overset{1}{\hookrightarrow} \texttt{fail}}\, type(\texttt{true}) \neq \texttt{cod}^{(\cdot)}}{\$(\$([|\texttt{true}|]\{\})^{\texttt{cod}^{(\cdot)}})^{\texttt{int}} \overset{2}{\hookrightarrow} \texttt{fail}}}{[|\$(\$([|\texttt{true}|]\{\})^{\texttt{cod}^{(\cdot)}})^{\texttt{int}}|]\{\} \overset{1}{\hookrightarrow} \texttt{fail}}}{[|[|\$(\$([|\texttt{true}|]\{\})^{\texttt{cod}^{(\cdot)}})^{\texttt{int}}|]\{\}|]\{\} \overset{0}{\hookrightarrow} [|\texttt{fail}|]\{\}}$$

Esto se debe a que el segundo escape sucede en el nivel 1 y en el mismo se requiere que la expresión citada `true`, de tipo `bool`, sea de tipo código. El error de tipos genera un `fail` que se propaga a lo largo de toda la evaluación.

### Sustituciones Explícitas

Retornando a la Figura 4.9, cuando se evalúa una sustitución explícita $e[\Theta]$, se debe evaluar la expresión $e$ a un valor $[|v_1|]\{\Gamma_1\}$. Luego se debe evaluar $\Gamma_1 \vdash \Theta : v_1 \Longrightarrow v_2$, que representa el efecto de la sustitución sobre el valor $v_1$ en el contexto $\Gamma_1$. Entonces, dado el juicio $\Gamma_1 \vdash \Theta \Rightarrow \Gamma_2$ que representa el efecto de la sustitución sobre el contexto $\Gamma_1$, el resultado de la sustitución es $[|v_2|]\{\Gamma_2\}$.

$$\Gamma \vdash \Theta : b \Longrightarrow b \quad \Gamma \vdash \Theta : i \Longrightarrow i \quad \Gamma \vdash \Theta : \texttt{fail} \Longrightarrow \texttt{fail}$$

$$\frac{\Gamma \vdash \Theta : v_1 \Longrightarrow v_1' \quad \Gamma \vdash \Theta : v_2 \Longrightarrow v_2'}{\Gamma \vdash \Theta : (v_1, v_2) \Longrightarrow (v_1', v_2')} \qquad \frac{\Gamma \vdash \Theta : v_1 \Longrightarrow v_1' \quad \Gamma \vdash \Theta : v_2 \Longrightarrow v_2'}{\Gamma \vdash \Theta : v_1 \ v_2 \Longrightarrow v_1' \ v_2'}$$

$$\frac{\Gamma \vdash \Theta : v \Longrightarrow v'}{\Gamma \vdash \Theta : \texttt{fst} \ v \Longrightarrow \texttt{fst} \ v'} \qquad \frac{\Gamma \vdash \Theta : v \Longrightarrow v'}{\Gamma \vdash \Theta : \texttt{snd} \ v \Longrightarrow \texttt{snd} \ v'}$$

$$\frac{\Gamma \vdash \Theta : v_1 \Longrightarrow v_1' \quad \Gamma \vdash \Theta : v_2 \Longrightarrow v_2' \quad \Gamma \vdash \Theta : v_3 \Longrightarrow v_3'}{\Gamma \vdash \Theta : \texttt{if} \ v_1 \ \texttt{then} \ v_2 \ \texttt{else} \ v_3 \Longrightarrow \texttt{if} \ v_1' \ \texttt{then} \ v_2' \ \texttt{else} \ v_3'}$$

$$\frac{\Gamma, \tau \vdash\Uparrow \Theta : v \Longrightarrow v'}{\Gamma \vdash \Theta : \lambda^\tau.v \Longrightarrow \lambda^\tau.v'} \qquad \frac{\Gamma, \tau \vdash\Uparrow \Theta : v \Longrightarrow v'}{\Gamma \vdash \Theta : \texttt{fix} \ v \Longrightarrow \texttt{fix} \ v'} (\tau = type(v))$$

$$\frac{\Gamma \vdash \Theta : v \Longrightarrow v'}{\Gamma \vdash \Theta : v[\Theta_2] \Longrightarrow v'[\Theta_2]} \qquad \frac{\Gamma \vdash \Theta : v_1 \Longrightarrow v_1' \quad \Gamma \vdash \Theta : v_2 \Longrightarrow v_2'}{\Gamma \vdash \Theta : \texttt{run} \ v_1 \,|\, v_2 \Longrightarrow \texttt{run} \ v_1' \,|\, v_2'}$$

$$\frac{\begin{array}{c}\Theta' = pats(p, \Theta) \qquad \vdash p : \tau_1 \Rightarrow \Gamma' \\ \Gamma, \Gamma' \vdash \Theta' : v_1 \Longrightarrow v_1' \quad \Gamma \vdash \Theta : v_2 \Longrightarrow v_2'\end{array}}{\Gamma \vdash \Theta : \lambda p. \ v_1 \,|\, v_2 \Longrightarrow \lambda p. \ v_1' \,|\, v_2'} (\tau_1 \to \tau_2 = type(v_1))$$

$$\frac{v' = bds(v, 0, \Theta, \Gamma)}{\Gamma \vdash \Theta : [\,|\, v \,|\,]\{\Gamma^f\} \Longrightarrow [\,|\, v' \,|\,]\{\Gamma^f\}} \qquad \Gamma, \tau \vdash\uparrow^{\tau'} : x^\tau \Longrightarrow (\texttt{s} \ x)^\tau$$

$$\Gamma, \tau \vdash e/ : \texttt{z}^\tau \Longrightarrow e \qquad \Gamma, \tau' \vdash e/ : (\texttt{s} \ x)^\tau \Longrightarrow x^\tau$$

$$\Gamma, \tau \vdash\Uparrow \Theta : \texttt{z}^\tau \Longrightarrow \texttt{z}^\tau \quad \frac{\Gamma \vdash \Theta \Rightarrow \Gamma' \quad \Gamma \vdash \Theta : x^\tau \Longrightarrow v \quad \Gamma' \vdash\uparrow^{\tau'} : v \Longrightarrow v'}{\Gamma, \tau' \vdash\Uparrow \Theta : (\texttt{s} \ x)^\tau \Longrightarrow v'}$$

Figura 4.12: Evaluación de la Sustitución

La evaluación de la sustitución sobre el valor citado se muestra en la Figura 4.12, y consiste esencialmente en aplicar recursivamente la evlauación sobre las sub-expresiones y reconstruir la expresión con los resultados. Cuando se llega a una variable se debe aplicar la sustitución indicada según el caso:

**desplazar ($\uparrow^{\tau'}$):** agrega una nueva variable en el índice 0 y aumenta en uno el índice de todas las demás variables. Por lo tanto, el resultado de evaluar la sustitución sobre una variable $x^{\tau}$ es $(\mathtt{s}\ x)^{\tau}$.
Por ejemplo: $\cdot, \mathtt{int} \vdash \uparrow^{\mathtt{bool}} : \#0^{\mathtt{int}} \Longrightarrow \#1^{\mathtt{int}}$.

**barra ($e/$):** reemplaza la variable con índice 0 por $e$, entonces aplicarla a $\mathtt{z}^{\tau}$ resulta en $e$. Al quitarse la primera variable se debe decrementar en uno el resto del contexto, por lo tanto al aplicar la sustitución sobre $(\mathtt{s}\ x)^{\tau}$, el resultado es $x^{\tau}$.
Por ejemplo: $\cdot, \mathtt{int}, \mathtt{int} \vdash 9/ : \#0^{\mathtt{int}} \Longrightarrow 9$ y $\cdot, \mathtt{int}, \mathtt{int} \vdash 9/ : \#1^{\mathtt{int}} \Longrightarrow \#0^{\mathtt{int}}$.

**levantar ($\Uparrow \Theta$):** como esta sustitución deja la variable con índice 0 sin cambios y aplica $\Theta$ sobre el resto del contexto, en el caso de $\mathtt{z}^{\tau}$ el resultado es $\mathtt{z}^{\tau}$. Para una variable $(\mathtt{s}\ x)^{\tau}$, se debe aplicar $\Theta$ para la variable con su índice decrementado en uno ($x^{\tau}$), habiéndole quitado la variable 0 al contexto, y luego aplicar la sustitución $\uparrow^{\tau'}$ al resultado, reconstruyendo así el contexto anterior.
Por ejemplo: $\cdot, \mathtt{int}, \mathtt{int}, \mathtt{bool} \vdash \Uparrow 9/ : \#1^{\mathtt{int}} \Longrightarrow 9$, resulta en las evaluaciones:

- $\cdot, \mathtt{int}, \mathtt{int} \vdash 9/ : \#0^{\mathtt{int}} \Longrightarrow 9$
- $\cdot, \mathtt{int}, \mathtt{int} \vdash 9/ \Rightarrow \cdot, \mathtt{int}$
- $\cdot, \mathtt{int} \vdash \uparrow^{\mathtt{bool}} : 9 \Longrightarrow 9$

Entonces, si se quiere evaluar la sustitución:

$$[\mid (\#2^{\mathtt{int}}, (\#0^{\mathtt{bool}}, \#1^{\mathtt{int}})) \mid]\{\mathtt{int}, \mathtt{int}, \mathtt{bool}\}[\Uparrow 9/]$$

como la expresión citada es un par, se debe aplicar la misma sustitución a las sub-expresiones $\#2^{\mathtt{int}}$ y $(\#0^{\mathtt{bool}}, \#1^{\mathtt{int}})$. El resultado para la primera sub-expresión es $\#1^{\mathtt{int}}$, dada la siguiente derivación:

$$\frac{\dfrac{\cdot, \mathtt{int}, \mathtt{int} \vdash 9/ : \#1^{\mathtt{int}} \Longrightarrow \#0^{\mathtt{int}} \qquad \cdot, \mathtt{int}, \mathtt{int} \vdash 9/ \Rightarrow \cdot, \mathtt{int}}{\cdot, \mathtt{int} \vdash \uparrow^{\mathtt{bool}} : \#0^{\mathtt{int}} \Longrightarrow \#1^{\mathtt{int}}}}{\cdot, \mathtt{int}, \mathtt{int}, \mathtt{bool} \vdash \Uparrow 9/ : \#2^{\mathtt{int}} \Longrightarrow \#1^{\mathtt{int}}}$$

La segunda sub-expresión es un par, por lo que se debe aplicar la sustitución a sus componentes, en donde se llega a las situaciones de los ejemplos vistos anteriormente:

$$\frac{\cdot, \mathtt{int}, \mathtt{int}, \mathtt{bool} \vdash \Uparrow 9/ : \#0^{\mathtt{bool}} \Longrightarrow \#0^{\mathtt{bool}} \qquad \dfrac{\begin{array}{c}\cdot, \mathtt{int}, \mathtt{int} \vdash 9/ : \#0^{\mathtt{int}} \Longrightarrow 9 \\ \cdot, \mathtt{int}, \mathtt{int} \vdash 9/ \Rightarrow \cdot, \mathtt{int} \\ \cdot, \mathtt{int} \vdash \uparrow^{\mathtt{bool}} : 9 \Longrightarrow 9\end{array}}{\cdot, \mathtt{int}, \mathtt{int}, \mathtt{bool} \vdash \Uparrow 9/ : \#1^{\mathtt{int}} \Longrightarrow 9}}{\cdot, \mathtt{int}, \mathtt{int}, \mathtt{bool} \vdash \Uparrow 9/ : (\#0^{\mathtt{bool}}, \#1^{\mathtt{int}}) \Longrightarrow (\#0^{\mathtt{bool}}, 9)}$$

Dados estos resultados y la siguiente derivación de las reglas de tipado de la sustitución explícita:

$$\frac{\cdot, \mathtt{int}, \mathtt{int} \vdash 9/ \Rightarrow \cdot, \mathtt{int}}{\cdot, \mathtt{int}, \mathtt{int}, \mathtt{bool} \vdash \Uparrow (9/) \Rightarrow \cdot, \mathtt{int}, \mathtt{bool}}\ \text{Lift}$$

se tiene que el resultado de la sustitución es : $\texttt{[|}(\#1^{\texttt{int}}, (\#0^{\texttt{bool}}, 9))\texttt{|]}\{\texttt{int}, \texttt{bool}\}$.

En el caso de la lambda-abstracción y del operador de punto fijo, al evaluar el resultado de la sustitución sobre su cuerpo, se debe agregar una sustitución levantar. Esto se realiza debido a que estas construcciones ligan una nueva variable en el índice 0, y por lo tanto las variables sobre las cuales se debe hacer la sustitución tienen sus índices incrementados en uno. Entonces, por ejemplo, al evaluarse el efecto de la sustitución $9/$ sobre la expresión $\lambda^{\texttt{bool}}.(\#2^{\texttt{int}}, (\#0^{\texttt{bool}}, \#1^{\texttt{int}}))$, se debe aplicar la sustitución $\Uparrow 9/$ sobre el cuerpo de la lambda-abstracción, que es equivalente a la sustitución vista en el ejemplo anterior.

$$\dfrac{\dfrac{\dots}{\cdot, \texttt{int}, \texttt{int}, \texttt{bool} \vdash \Uparrow 9/ : (\#2^{\texttt{int}}, (\#0^{\texttt{bool}}, \#1^{\texttt{int}})) \Longrightarrow (\#1^{\texttt{int}}, (\#0^{\texttt{bool}}, 9))}}{\cdot, \texttt{int}, \texttt{int} \vdash 9/ : \lambda^{\texttt{bool}}.(\#2^{\texttt{int}}, (\#0^{\texttt{bool}}, \#1^{\texttt{int}})) \Longrightarrow \lambda^{\texttt{bool}}.(\#1^{\texttt{int}}, (\#0^{\texttt{bool}}, 9))}$$

De forma similar a la lambda-abstracción, la alternación puede ligar nuevas variables, modificando el contexto de la expresión de la primera rama. Para capturar la semántica del ligado de variables de un patrón, y como esto modifica la sustitución a realizar, se define la función $pats : \mathsf{P} \times \mathsf{S} \to \mathsf{S}$, que dado un patrón y una sustitución, retorna una nueva sustitución.

$$\begin{array}{lcl}
pats(_\ , \Theta) & = & \Theta \\
pats(\bullet^{\tau}, \Theta) & = & \Uparrow \Theta \\
pats(i, \Theta) & = & \Theta \\
pats(b, \Theta) & = & \Theta \\
pats((p_1, p_2), \Theta) & = & pats(p_2, pats(p_1, \Theta)) \\
pats(\texttt{[|}pc\texttt{|]}, \Theta) & = & patcs(pc, \Theta)
\end{array}$$

En los casos en que no se ligan variables $(i, b, \_)$ esta función es la identidad, mientras que en el caso del patrón ligado de variable, al igual que en la lambda-abstracción, se agrega una sustitución levantar. Cuando el patrón es un par, se modifica la sustitución original con el sub-patrón de la izquierda y luego se modifica la sustitución resultante con el sub-patrón de la derecha, dado que las variables ligadas por este último son las de menor índice. Para los patrones de código, se invoca la función $patcs : \mathsf{PC} \times \mathsf{S} \to \mathsf{S}$, que se comporta de forma análoga a $pats$, agregando una sustitución levantar en los casos de patrones que ligan variables.

$$
\begin{array}{lcl}
patcs(\_\,, \Theta) & = & \Theta \\
patcs(\texttt{fail}, \Theta) & = & \Theta \\
patcs(i, \Theta) & = & \Theta \\
patcs(b, \Theta) & = & \Theta \\
patcs(\$(\bullet)^\tau, \Theta) & = & \Uparrow \Theta \\
patcs(\$(\texttt{lit})^\tau, \Theta) & = & \Uparrow \Theta \\
patcs((pc_1, pc_2), \Theta) & = & patcs(pc_2, patcs(pc_1, \Theta)) \\
patcs(\texttt{fst}\ pc, \Theta) & = & patcs(pc, \Theta) \\
patcs(\texttt{snd}\ pc, \Theta) & = & patcs(pc, \Theta) \\
patcs(\lambda^\tau.pc, \Theta) & = & patcs(pc, \Theta) \\
patcs(\texttt{fix}^\tau\ pc, \Theta) & = & patcs(pc, \Theta) \\
patcs(pc_1\ pc_2, \Theta) & = & patcs(pc_2, patcs(pc_1, \Theta)) \\
patcs(x^\tau, \Theta) & = & \Theta \\
patcs(\texttt{if}\ pc_1\ \texttt{then}\ pc_2\ \texttt{else}\ pc_3, \Theta) & = & patcs(pc_3, patcs(pc_2, patcs(pc_1, \Theta))) \\
patcs(\texttt{[|}\_\texttt{|]}\{\Gamma^f\}, \Theta) & = & \Uparrow \Theta \\
patcs(\texttt{run}\ pc_1\texttt{|}pc_2, \Theta) & = & patcs(pc_2, patcs(pc_1, \Theta)) \\
patcs(pc[\Theta'], \Theta) & = & patcs(pc, \Theta) \\
patcs(\lambda p^\tau.pc_1\texttt{|}pc_2, \Theta) & = & patcs(pc_2, patcs(pc_1, \Theta))
\end{array}
$$

Cuando se realiza una sustitución $e[\Theta]$, y $e$ evalúa a $\texttt{[|}v\texttt{|]}\{\Gamma\}$, $\Theta$ se aplica a todas las variables que están en el nivel 0 de $v$. Entonces, si $v$ es una expresión que tiene paréntesis, se debe realizar un conteo de los paréntesis y los escapes de la expresión para buscar la presencia de variables que estén en el nivel 0. Esto se realiza utilizando la función $bds : \mathsf{V}^n \times \texttt{int} \times \mathsf{S} \times \mathsf{C} \to \mathsf{V}^n$, que toma un valor, un entero que se inicia en 0, una sustitución y un contexto, y devuelve el valor recostruído. El entero lleva la cuenta de la cantidad de paréntesis menos la cantidad de escapes que hay en la expresión que se pasa. La idea se basa en recorrer y reconstruir la expresión. Cada vez que se encuentra un nuevo paréntesis el entero se incrementa en uno. En el caso de los escapes, si el entero es 0 se debe realizar la sustitución sobre el valor escapado, porque el mismo se encuentra en el nivel 0. Si el entero es mayor que 0, éste se decrementa en uno.

$$
\begin{array}{lcl}
bds(i, n, \Theta, \Gamma) & = & i \\
bds(b, n, \Theta, \Gamma) & = & b \\
bds((v_1, v_2), n, \Theta, \Gamma) & = & (bds(v_1, n, \Theta, \Gamma), bds(v_2, n, \Theta, \Gamma)) \\
bds(\texttt{fst}\ v, n, \Theta, \Gamma) & = & \texttt{fst}\ bds(v, n, \Theta, \Gamma) \\
bds(\texttt{snd}\ v, n, \Theta, \Gamma) & = & \texttt{snd}\ bds(v, n, \Theta, \Gamma) \\
bds(\lambda^\tau.v, n, \Theta, \Gamma) & = & \lambda^\tau.bds(v, n, \Theta, \Gamma) \\
bds(\texttt{fix}\ v, n, \Theta, \Gamma) & = & \texttt{fix}\ bds(v, n, \Theta, \Gamma) \\
bds(v_1\ v_2, n, \Theta, \Gamma) & = & bds(v_1, n, \Theta, \Gamma)\ bds(v_2, n, \Theta, \Gamma) \\
bds(x^\tau, n, \Theta, \Gamma) & = & x^\tau \\
bds(\lambda p.\ v_1\texttt{|}v_2, n, \Theta, \Gamma) & = & \lambda p.\ bds(v_1, n, \Theta, \Gamma)\texttt{|}bds(v_2, n, \Theta, \Gamma) \\
bds(\texttt{if}\ v_1 & & \texttt{if}\ bds(v_1, n, \Theta, \Gamma) \\
\quad \texttt{then}\ v_2 & = & \texttt{then}\ bds(v_2, n, \Theta, \Gamma) \\
\quad \texttt{else}\ v_3, n, \Theta, \Gamma) & & \texttt{else}\ bds(v_3, n, \Theta, \Gamma) \\
bds(v[\Theta'], n, \Theta, \Gamma) & = & bds(v, n, \Theta, \Gamma)[\Theta'] \\
bds(\texttt{run}\ v_1\texttt{|}v_2, n, \Theta, \Gamma) & = & \texttt{run}\ bds(v_1, n, \Theta, \Gamma)\texttt{|}bds(v_2, n, \Theta, \Gamma) \\
bds(\texttt{[|}v\texttt{|]}\{\Gamma^f\}, n, \Theta, \Gamma) & = & \texttt{[|}bds(v, n+1, \Theta, \Gamma)\texttt{|]}\{\Gamma^f\} \\
bds(\$(v)^\tau, n, \Theta, \Gamma) & = & \begin{cases} \$(v')^\tau, & \text{si } n = 0 \text{ y } \Gamma \vdash \Theta : v \Longrightarrow v'; \\ \$(bds(v, n-1, \Theta, \Gamma))^\tau, & \text{otro caso.} \end{cases}
\end{array}
$$

Entonces, al evaluarse la sustitución $\texttt{[|[|\$(\#0}^{\texttt{cod}^{(\cdot)}}\texttt{)}^{\texttt{int}}\texttt{|]\{\}|]\{cod}^{(\cdot)}\texttt{\}[[|1|]\{\}/]}$, se da la siguiente derivación:

$$\dfrac{\dfrac{\cdot, \texttt{cod}^{(\cdot)} \vdash \texttt{[|1|]\{\}/} : \texttt{\#0}^{\texttt{cod}^{(\cdot)}} \Longrightarrow \texttt{[|1|]\{\}}}{bds(\texttt{\$(\#0}^{\texttt{cod}^{(\cdot)}}\texttt{)}^{\texttt{int}}, 0, \texttt{[|1|]\{\}/}, \{\cdot, \texttt{cod}^{(\cdot)}\}) = \texttt{[|1|]\{\}}}(n = 0)}{\cdot, \texttt{cod}^{(\cdot)} \vdash \texttt{[|1|]\{\}/} : \texttt{[|\$(\#0}^{\texttt{cod}^{(\cdot)}}\texttt{)}^{\texttt{int}}\texttt{|]\{\}} \Longrightarrow \texttt{[|\$([|1|]\{\})}^{\texttt{int}}\texttt{|]\{\}}}$$

Como la variable $\texttt{\#0}^{\texttt{cod}^{(\cdot)}}$ se encuentra en el nivel de la sustitución, la misma es reemplazada por el valor $\texttt{[|1|]\{\}}$. Por lo que el resultado es $\texttt{[|[|\$([|1|]\{\})}^{\texttt{int}}\texttt{|]\{\}|]\{\}}$.

### 4.2.3. Extensión para Análisis Intensional

Para evaluar la alternación se agregan tres reglas, como lo muestra la Figura 4.13. La primera regla indica que una alternación, de forma similar a la lambda-abstracción, evalúa a sí misma. Las siguientes dos reglas son una extensión de la semántica de la aplicación, para el caso en que el operador sea una alternación. Con la notación $\Theta : p \triangleright v$ se expresa que el patrón $p$ concuerda con el valor $v$ vía la sustitución $\Theta$, la cual asigna los valores a las variables ligadas por el patrón. Mientras que $p \not\triangleright v$ significa que el patrón $p$ no concuerda con el valor $v$, lo que sucede si no se encuentra ningún caso en el que $\Theta : p \triangleright v$. En ambas reglas se evalúa el operador a una alternación $\lambda p.e_a \,|\, e_b$ y el operando a un valor $v_1$. En caso de concordancia $\Theta : p \triangleright v$, se evalúa la rama $e_a$ luego de realizarle la sustitución $\Theta$ de la concordancia. En otro caso se debe evaluar la aplicación de la rama $e_b$ al operando.

$$\lambda p.\ e_1 \,|\, e_2 \overset{0}{\hookrightarrow} \lambda p.\ e_1 \,|\, e_2$$

$$\dfrac{e_1 \overset{0}{\hookrightarrow} \lambda p.\ e_a \,|\, e_b \qquad e_2 \overset{0}{\hookrightarrow} v_1 \qquad \Theta : p \triangleright v_1 \qquad e_a\Theta \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2}$$

$$\dfrac{e_1 \overset{0}{\hookrightarrow} \lambda p.\ e_a \,|\, e_b \qquad e_2 \overset{0}{\hookrightarrow} v_1 \qquad p \not\triangleright v_1 \qquad e_b\ e_2 \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2}$$

Figura 4.13: Evaluación de la Extensión para Análisis Intensional

En la Figura 4.14 se encuentran las reglas de evaluación de la concordancia de patrones. El patrón cualquiera concuerda con cualquier valor $v$, vía la sustitución vacía $(\cdot)$, porque no liga ningún valor. También a través de la sustitución vacía, concuerdan los patrones de constantes literales $i$ y $b$ con los valores $i$ y $b$ respectivamente. En el patrón de ligado de variable $(\bullet^{\tau})$ se concuerda cualquier valor de tipo $\tau$, vía la sustitución que reemplaza la variable ligada por el valor concordado. Un patrón par $(p_1, p_2)$ concuerda con un par $(v_1, v_2)$, si lo hacen $p_1$ con $v_1$ y $p_2$ con $v_2$, vía las sustituciones $\Theta_1$ y $\Theta_2$ respectivamente. La sustitución de esta concordancia es la combinación de las dos anteriores. Con la notación $(\Theta_2, \Theta_1)$ se expresa que la sustitución resulta de aplicar primero $\Theta_2$ y sobre la expresión resultante aplicar $\Theta_1$.

$$\cdot : \_ \triangleright v \quad \cdot : i \triangleright i \quad \cdot : b \triangleright b \quad \frac{}{v/ : \bullet^{\tau} \triangleright v}(type(v) = \tau)$$

$$\frac{\Theta_1 : p_1 \triangleright v_1 \qquad \Theta_2 : p_2 \triangleright v_2}{\Theta_2, \Theta_1 : (p_1, p_2) \triangleright (v_1, v_2)} \qquad \frac{\Theta : pc \trianglerighteq v}{\Theta : \texttt{[|}pc\texttt{|]} \triangleright v}$$

Figura 4.14: Evaluación de los Patrones

La concordancia con expresiones citadas se realiza mediante patrones de código, en donde $\Theta : pc \trianglerighteq v$ indica que el patrón de código $pc$ concuerda con el código $v$ vía la sustitución $\Theta$, y $pc \not\trianglerighteq v$ que la concordancia no existe. Las reglas de evaluación de los patrones de código se muestran en la Figura 4.15.

En la mayoría de los casos se descompone la expresión citada en sus sub-expresiones, y se evalúan los sub-patrones con las sub-expresiones citadas en el mismo contexto que la original. En los casos de la lambda-abstracción y el operador de punto fijo, el sub-patrón $pc$ se evalúa con el cuerpo $v$ de la abstracción o punto fijo, citada en el contexto $\Gamma$ extendido con el tipo $\tau$ de la variable ligada. En el caso de la alternación, el segundo sub-patrón ($pc_2$) se evalúa con la segunda rama ($v_2$), citada en el contexto $\Gamma$, y $pc_1$ se evalúa con la rama $v_1$, citada en el contexto $\Gamma$ extendido con $\Gamma'$, que es el contexto "resultado" del patrón $p$ de la alternación.

Retomando el ejemplo de la sección 4.1.3, de una alternación con patrón de código:

$$\begin{array}{rl} \lambda\texttt{[|if } \#0^{\texttt{bool}} \texttt{ then \$(}\bullet\texttt{)}^{\texttt{int}} \texttt{ else \$(}\bullet\texttt{)}^{\texttt{int}}\texttt{|]} & \\ \quad . & \#1^{\texttt{cod}^{\cdot,\texttt{bool}}}\texttt{[true/]} \\ \quad | & \lambda^{\texttt{cod}^{\cdot,\texttt{bool}}}. \quad \#0^{\texttt{cod}^{\cdot,\texttt{bool}}}\texttt{[true/]} \end{array}$$

Si se quiere evaluar la aplicación de esta alternación a la expresión:

$$\texttt{[| if } \#0^{\texttt{bool}} \texttt{ then 1 else 2|]\{bool\}}$$

luego de evaluar ambas expresiones, las cuales evalúan a sí mismas, se debe realizar la concordancia entre el patrón $\texttt{[|if } \#0^{\texttt{bool}} \texttt{ then \$(}\bullet\texttt{)}^{\texttt{int}} \texttt{ else \$(}\bullet\texttt{)}^{\texttt{int}}\texttt{|]}$ y el valor resultante de la expresión.

$$\frac{\begin{array}{c}\cdot : \#0^{\texttt{bool}} \trianglerighteq \texttt{[|}\#0^{\texttt{bool}}\texttt{|]\{bool\}} \quad \texttt{[|1|]\{bool\}/} : \$(\bullet)^{\texttt{bool}} \trianglerighteq \texttt{[|1|]\{bool\}} \\ \texttt{[|2|]\{bool\}/} : \$(\bullet)^{\texttt{bool}} \trianglerighteq \texttt{[|2|]\{bool\}}\end{array}}{\begin{array}{c}\texttt{[|2|]\{bool\}/, [|1|]\{bool\}/ : if } \#0^{\texttt{bool}} \texttt{ then \$(}\bullet\texttt{)}^{\texttt{int}} \texttt{ else \$(}\bullet\texttt{)}^{\texttt{int}} \\ \trianglerighteq \texttt{[| if } \#0^{\texttt{bool}} \texttt{ then 1 else 2|]\{bool\}}\end{array}}$$

$$\frac{\begin{array}{c}\texttt{[|2|]\{bool\}/, [|1|]\{bool\}/ : [|if } \#0^{\texttt{bool}} \texttt{ then \$(}\bullet\texttt{)}^{\texttt{int}} \texttt{ else \$(}\bullet\texttt{)}^{\texttt{int}}\texttt{|]} \\ \triangleright\texttt{[| if } \#0^{\texttt{bool}} \texttt{ then 1 else 2|]\{bool\}}\end{array}}{}$$

Dado que el patrón y el valor concuerdan vía la sustitución ($\texttt{[|2|]\{bool\}/, [|1|]\{bool\}/}$), ésta se debe aplicar a la primera rama de la alternación. Esto significa que primero se debe realizar la sustitución $(\#1^{\texttt{cod}^{\cdot,\texttt{bool}}}\texttt{[true/]})\texttt{[[|2|]\{bool\}/]}$, y a su resultado realizar la sustitución $(\#0^{\texttt{cod}^{\cdot,\texttt{bool}}}\texttt{[true/]})\texttt{[[|1|]\{bool\}/]}$. Al evaluarse la expresión resultante $\texttt{[|1|]\{bool\}[true/]}$ se obtiene el resultado de la alternación, el cual es $\texttt{[|1|]\{bool\}}$.

$$\cdot : \_ \trianglerighteq [\![ v ]\!]\{\Gamma\} \quad \cdot : \texttt{fail} \trianglerighteq [\![ \texttt{fail} ]\!]\{\Gamma\} \quad \cdot : i \trianglerighteq [\![ i ]\!]\{\Gamma\} \quad \cdot : b \trianglerighteq [\![ b ]\!]\{\Gamma\}$$

$$\frac{}{[\![ v ]\!]\{\Gamma\} / : \texttt{\$(}\bullet\texttt{)}^{\tau} \trianglerighteq [\![ v ]\!]\{\Gamma\}} (type(v) = \tau)$$

$$[\![ i ]\!]\{\Gamma\} / : \texttt{\$(lit)}^{\texttt{int}} \trianglerighteq [\![ i ]\!]\{\Gamma\} \quad [\![ b ]\!]\{\Gamma\} / : \texttt{\$(lit)}^{\texttt{bool}} \trianglerighteq [\![ b ]\!]\{\Gamma\}$$

$$\frac{\Theta_1 : pc_1 \trianglerighteq [\![ v_1 ]\!]\{\Gamma\} \qquad \Theta_2 : pc_2 \trianglerighteq [\![ v_2 ]\!]\{\Gamma\}}{\Theta_2, \Theta_1 : (pc_1, pc_2) \trianglerighteq [\![ (v_1, v_2) ]\!]\{\Gamma\}}$$

$$\frac{\Theta : pc \trianglerighteq [\![ v ]\!]\{\Gamma\}}{\Theta : \texttt{fst}\ pc \trianglerighteq [\![ \texttt{fst}\ v ]\!]\{\Gamma\}} \qquad \frac{\Theta : pc \trianglerighteq [\![ v ]\!]\{\Gamma\}}{\Theta : \texttt{snd}\ pc \trianglerighteq [\![ \texttt{snd}\ v ]\!]\{\Gamma\}}$$

$$\frac{\Theta : pc \trianglerighteq [\![ v ]\!]\{\Gamma, \tau\}}{\Theta : \lambda^{\tau}.pc \trianglerighteq [\![ \lambda^{\tau}.v ]\!]\{\Gamma\}} \qquad \frac{\Theta : pc \trianglerighteq [\![ v ]\!]\{\Gamma, \tau\}}{\Theta : \texttt{fix}^{\tau}\ pc \trianglerighteq [\![ \texttt{fix}\ v ]\!]\{\Gamma\}} (type(v) = \tau)$$

$$\frac{\Theta_1 : pc_1 \trianglerighteq [\![ v_1 ]\!]\{\Gamma\} \qquad \Theta_2 : pc_2 \trianglerighteq [\![ v_2 ]\!]\{\Gamma\}}{\Theta_2, \Theta_1 : pc_1\ pc_2 \trianglerighteq [\![ v_1\ v_2 ]\!]\{\Gamma\}} \quad \cdot : x^{\tau} \trianglerighteq [\![ x^{\tau} ]\!]\{\Gamma\}$$

$$\frac{\Theta_1 : pc_1 \trianglerighteq [\![ v_1 ]\!]\{\Gamma\} \qquad \Theta_2 : pc_2 \trianglerighteq [\![ v_2 ]\!]\{\Gamma\} \qquad \Theta_3 : pc_3 \trianglerighteq [\![ v_3 ]\!]\{\Gamma\}}{\Theta_3, \Theta_2, \Theta_1 : \texttt{if}\ pc_1\ \texttt{then}\ pc_2\ \texttt{else}\ pc_3 \trianglerighteq [\![ \texttt{if}\ v_1\ \texttt{then}\ v_2\ \texttt{else}\ v_3 ]\!]\{\Gamma\}}$$

$$\cdot : [\![ \_ ]\!]\{\Gamma^f\} \trianglerighteq [\![ [\![ v ]\!]\{\Gamma^f\} ]\!]\{\Gamma\} \quad \frac{\Theta : pc \trianglerighteq [\![ v ]\!]\{\Gamma\}}{\Theta : pc[\Theta_1] \trianglerighteq [\![ v[\Theta_1] ]\!]\{\Gamma\}}$$

$$\frac{\Theta_1 : pc_1 \trianglerighteq [\![ v_1 ]\!]\{\Gamma\} \qquad \Theta_2 : pc_2 \trianglerighteq [\![ v_2 ]\!]\{\Gamma\}}{\Theta_2, \Theta_1 : \texttt{run}\ pc_1 | pc_2 \trianglerighteq [\![ \texttt{run}\ v_1 | v_2 ]\!]\{\Gamma\}}$$

$$\frac{\vdash p : \tau \Rightarrow \Gamma' \qquad \Theta_1 : pc_1 \trianglerighteq [\![ v_1 ]\!]\{\Gamma, \Gamma'\} \qquad \Theta_2 : pc_2 \trianglerighteq [\![ v_2 ]\!]\{\Gamma\}}{\Theta_2, \Theta_1 : \lambda p^{\tau}.pc_1 | pc_2 \trianglerighteq [\![ \lambda p.v_1 | v_2 ]\!]\{\Gamma\}}$$

Figura 4.15: Evaluación de los Patrones de Código

Si se aplica la alternación a la expresión $\texttt{[|\#0}^{\texttt{bool}}\texttt{|]\{bool\}}$, al intentar evaluar la concordancia del patrón de código con el valor, no se encuentra ninguna regla que los contenga. Por lo tanto los mismos no concuerdan.

$$\frac{\texttt{if \#0}^{\texttt{bool}}\texttt{ then \$(•)}^{\texttt{int}}\texttt{ else \$(•)}^{\texttt{int}} \not\triangleright \texttt{[|\#0}^{\texttt{bool}}\texttt{|]\{bool\}}}{\texttt{[|if \#0}^{\texttt{bool}}\texttt{ then \$(•)}^{\texttt{int}}\texttt{ else \$(•)}^{\texttt{int}}\texttt{|]} \not\triangleright \texttt{[|\#0}^{\texttt{bool}}\texttt{|]\{bool\}}}$$

Entonces se debe evaluar la aplicación de la segunda rama de la alternación a la expresión, cuyo resultado es `[|true|]{}`.

## 4.3. Ejemplos

En esta sección se mostrarán algunos ejemplos del uso del lenguaje. Estos ejemplos manipulan programas que representan funciones matemáticas de una variable. Cada programa representa a una función $f(x)$ de reales a reales[6]. Si $f(x) = e$, entonces su representación es dada por el código `[|`$e$`|]{x:real}` de tipo `cod{x:real}`. Las funciones pueden contener sumas, productos, literales, variables y composiciones de funciones $f(g(x))$ que se representan como `[|(fn a:real =>` $e$`)` $e'$`|]{x:real}`, si $f(a) = e$ y $g(x) = e'$. Por ejemplo, dado que las funciones $f(x) = x*2+1$ y $g(x) = x*x$ se representan con los programas `[|x:real * 2 + 1|]{x:real}` y `[|x:real * x:real|]{x:real}`, respectivamente, su composición $f(g(x))$ es `[|(fn a:real =>a:real * 2 + 1) (x:real * x:real) |]{x:real}`.

### 4.3.1. Diferenciación

A continuación se muestra un programa que calcula la derivada de una función, representada por un código de tipo `cod{x:real}`, utilizando las reglas de la Figura 4.16. Este es un ejemplo de como se pueden generar nuevos fragmentos de código, en base al análisis de otro código.

$$\frac{d}{dx}x = 1 \qquad \frac{d}{dx}c = 0$$

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$$

$$\frac{d}{dx}(f(x).g(x)) = f(x).\frac{d}{dx}g(x) + g(x).\frac{d}{dx}f(x)$$

$$\frac{d}{dx}f(g(x)) = \frac{d}{dg}f(g(x)).\frac{d}{dx}g(x)$$

Figura 4.16: Reglas de Diferenciación

```
val rec diff : cod{x:real} -> cod{x:real}
  = fn [| x:real |] => [| 1 |]{x:real}
  | fn [| $(c:lit):real |] => [| 0 |]{x:real}
  | fn [| $(a:cod{x:real}):real + $(b:cod{x:real}):real |]
    => [| $(diff a:cod{x:real}):real + $(diff b:cod{x:real}):real |]{x:real}
  | fn [| $(a:cod{x:real}):real * $(b:cod{x:real}):real |]
```

[6]Para los ejemplos se asume que el tipo `real` pertenece al lenguaje.

```
    => [| $(a:cod{x:real}):real * $(diff b:cod{x:real}):real
            + $(b:cod{x:real}):real * $(diff a:cod{x:real}):real |]{x:real}
  | fn [| (fn a:real => $(f:cod{a:real}):real) $(g:cod{x:real}):real |]
    => [| ((fn a:real => $(diff f:cod{a:real}):real) $(g:cod{x:real}):real)
            * $(diff g:cod{x:real}):real |]{x:real}
  | fn a:cod{x:real} => a:cod{x:real}
```

Se recorre recursivamente la estructura del código que representa a la función y se construye un nuevo código de acuerdo a cada caso. Si se encuentra una variable o un literal[7] se retorna un código que representa al literal 1 o el 0, respectivamente. Si es una suma, se retorna el código de la suma de las derivadas de los sumandos, que se calculan recursivamente. De forma similar, en el caso del producto se aplica la regla de diferenciación correspondiente. El caso más interesante es en el cual se aplica la regla de la cadena. En este caso, el patrón `[| (fn a:real =>` `$(f:cod{a:real}):real)` `$(g:cod{x:real}):real |]` concuerda con un código que represente una composición de funciones $f(g(x))$. Los subpatrones variable `$(f:cod{a:real}):real)` y `$(g:cod{x:real}):real` ligan a las representaciones de las funciones $f(x)$ y $g(x)$ con las variables `f` y `g`, respectivamente. Se calcula recursivamente la derivada de la función que representa `f` y se construye el código de la composición del resultado con `g`, de manera de tener la representación de $\frac{d}{dg}f(g(x))$. Por otro lado se calcula recursivamente la derivada de la función representada por `g`, obteniendo $\frac{d}{dg}g(x)$. Finalmente, se retorna el código del producto de ambos resultados.

Por ejemplo, para calcular la derivada de $(x+1).(x+1)$, se debe realizar la siguiente invocación de `diff`.

```
diff [|(fn g:real => g*g) (x+1)|]{x:real}
```

La variable `f:cod{a:real}` se liga al valor `[|a*a|]{a:real}` y la variable `g:cod{x:real}` se liga al valor `[|x+1|]{x:real}`.
Entonces, la derivada de `f` se calcula `diff [|(x*x)|]{x:real}`, que resulta en el código `[|x*1 + x*1|]{x:real}`. La derivada de `g` se calcula `diff [|(x+1)|]{x:real}`, y su resultado es `[|1+0|]{x:real}`. Por lo tanto, la derivada de la composición es el código `[|((fn a:real => a*1 + a*1) (x+1)) * (1+0)|]{x:real}`, que representa a la función $\frac{d}{dg}f(g(x)) = ((x+1).1 + (x+1).1).(1+0)$.
En base a lo anterior se puede calcular la derivada segunda, cuyo resultado es:

```
[|((fn a:real => a*1 + a*1) (x+1)) * (0+0)
  + (1+0) * (((fn a:real => 1*1+a*0 + 1*1+a*0) (x+1)) * (1+0))|]{x:real}
```

representando a la función $\frac{d^2}{dg}f(g(x)) = ((x+1).1 + (x+1).1).0 + (1+0).((1.1 + (x+1).0) + (1.1 + (x+1).0)).(1+0)$, que, realizando algunas reducciones básicas, equivale a $\frac{d^2}{dg}f(g(x)) = 2$.

La función `diff` se puede utilizar, entre otras cosas, para realizar una implementación del método Newton-Raphson, para encontrar raíces de cualquier ecuación $f(x)$. Dada una aproximación inicial a una raíz y una tolerancia máxima, el método utiliza la derivada de la función para encontrar la raíz.

```
val newton : real -> real -> cod{x:real} -> real
  = fn x0:real => fn tol:real => fn f:cod{real} =>
    aprox x0:real tol:real f:cod{real} (diff f:cod{real})
```

[7] En el patrón de ligado de literal (como se había visto anteriormente en el caso de ligado de variable) se debe explicitar el nombre con el que se liga el literal concordado.

```
val rec aprox : real -> cod{x:real} -> cod{x:real} -> real -> real
  = fn tol:real => fn f:cod{x:real} => fn df:cod{x:real} => fn x0:real =>
    let xi = x0:real -
            (run f:cod{x:real}[x0/x] | 0) / (run df:cod{x:real}[x0/x] | 1)
    in
        if (abs (xi:real-x0:real)<tol:real)
        then xi:real
        else (aprox tol:real f:cod{x:real} df:cod{x:real} xi:real)
```

La variable `x0` representa a la aproximación inicial $x_0$, `tol` a la tolerancia y `f` a la función $f(x)$. Dada además la derivada de $f$, se realiza la aproximación de forma recursiva. En cada paso se calcula $x_i = x_0 - \frac{f(x_0)}{f'(x_0)}$, que es la raíz buscada si $|x_i - x_0|$ es menor que la tolerancia.

### 4.3.2. Reducción

El siguiente es un ejemplo del uso de la inspección de código para la optimización del mismo. Como se vió anteriormente, los resultados de la función `diff` pueden ser optimizados mediante algunas reducciones básicas. Por ejemplo, en el código:

```
[|((fn a:real => a*1 + a*1) (x+1)) * (0+0)
  + (1+0) * (((fn a:real => 1*1+a*0 + 1*1+a*0) (x+1)) * (1+0))|]{x:real}
```

hay varios productos por 1 y por 0, y sumas a 0, que se pueden simplificar trivialmente. También se puede aplicar las composiciones para llevar la función a su forma $\beta$-normal. Si se aplica a ese código todas estas reducciones se puede obtener una función optimizada equivalente `[|1+0|]x:real`. La función `red` implementa esta optimización.

```
val rec red : cod{x:real} -> cod{x:real}
  = fn [| x:real |] => [| x:real |]{x:real}
  | fn [| $(a:lit):real |] => [| $(a:cod{x:real}):real |]{x:real}
  | fn [| $(a:cod{x:real}):real + $(b:cod{x:real}):real |]
    => let  ar = red a:cod{x:real}
            br = red b:cod{x:real}
       in   redsum [|$(ar:cod{x:real}):real
                     + $(br:cod{x:real}):real|]{x:real}
  | fn [| $(a:cod{x:real}):real * $(b:cod{x:real}):real |]
    => let  ar = red a:cod{x:real}
            br = red b:cod{x:real}
       in   redpro [|$(ar:cod{x:real}):real
                     * $(br:cod{x:real}):real|]{x:real}
  | fn [| (fn a:real => $(f:cod{a:real}):real) $(g:cod{x:real}):real |]
    => red (redfun g:cod{x:real} f:cod{a:real})
  | fn a:cod{x:real} => a:cod{x:real}
```

La función toma el código y retorna una optimización del mismo. En el caso de que el código represente una suma o un producto, se aplica recursivamente la reducción a sus componentes y luego, si es posible, se realiza la simplificación. Las simplificaciones de la suma y el producto están implementadas por las funciones `redsum` y `redpro`, respectivamente.

```
val redsum : cod{x:real} -> cod{x:real}
  = fn [| $(a:cod{x:real}):real + 0 |] => a:cod{x:real}
  | fn [| 0 + $(a:cod{x:real}):real |] => a:cod{x:real}
  | fn a:cod{x:real} => a:cod{x:real}

val redpro : cod{x:real} -> cod{x:real}
  = fn [| $(a:cod{x:real}):real * 0 |] => [| 0 |]{x:real}
  | fn [| 0 * $(a:cod{x:real}):real |] => [| 0 |]{x:real}
  | fn [| $(a:cod{x:real}):real * 1 |] => a:cod{x:real}
  | fn [| 1 * $(a:cod{x:real}):real |] => a:cod{x:real}
  | fn a:cod{x:real} => a:cod{x:real}
```

En `redsum` se eliminan las sumas a 0. En `redpro` se eliminan los productos por 1, y si se detectan productos por 0 se retorna directamente el código del literal 0.

En el caso de las composiciones, se realiza una $\beta$-reducción con la función `redfun`, y se aplica la optimización al código resultante. La función `redfun` toma dos códigos de tipos `cod{x:real}` y `cod{a:real}`, y retorna un código que resulta de sustituir en el segundo parámetro cada ocurrencia de la variable `a` por el primero.

```
val rec redfun : cod{x:real} -> cod{a:real} -> cod{x:real}
  = fn g:cod{x:real} =>
      fn [| a:real |] => g:cod{x:real}
    | fn [| $(c:lit):real |] => [| $(c:cod{a:real}[0/a]):real |]{x:real}
    | fn [| $(c:cod{a:real}):real + $(d:cod{a:real}):real |]
        => let  cr = redfun g:cod{x:real} c:cod{a:real}
                dr = redfun g:cod{x:real} d:cod{a:real}
           in   [|$(cr:cod{x:real}):real
                        + $(dr:cod{x:real}):real|]{x:real}
    | fn [| $(c:cod{a:real}):real * $(d:cod{a:real}):real |]
        => let  cr = redfun g:cod{x:real} c:cod{a:real}
                dr = redfun g:cod{x:real} d:cod{a:real}
           in   [|$(cr:cod{x:real}):real
                        * $(dr:cod{x:real}):real|]{x:real}
    | fn [| (fn b:real => $(f:cod{b:real}):real) $(gi:cod{a:real}):real |]
        => let  gr = redfun g:cod{x:real} gi:cod{a:real}
           in   [| (fn b:real => $(f:cod{b:real}):real)
                        $(gr:cod{x:real}):real |]
    | fn a:cod{x:real} => a:cod{x:real}
```

### 4.3.3. Extensión

Este ejemplo muestra un caso de adaptación de código en tiempo de ejecución. La función `diff`, definida en la subsección 4.3.1, calcula la derivada de funciones que se representan con un código de tipo `cod{x:real}`, que pueden contener sumas, productos, literales variables y composiciones. Si el tipo de funciones que se pueden diferenciar se quisiera extender, `diff` tendría que ser reescrita para poder incluir los nuevos casos. La siguiente es una versión de `diff` que puede ser extendida en tiempo de ejecución, que se llamará `cod_diff`, dado que ahora el código se encierra entre paréntesis para hacerlo manipulable.

```
val cod_diff = [|
  val rec diff : ((cod{x:real} -> cod{x:real}) -> cod{x:real} -> cod{x:real})
               -> cod{x:real} -> cod{x:real}
      = fn ext: (cod{x:real} -> cod{x:real}) -> cod{x:real} -> cod{x:real} =>
        fn [| x:real |] => [| 1 |]{x:real}
      | fn [| $(c:lit):real |] => [| 0 |]{x:real}
      | fn [| $(a:cod{x:real}):real + $(b:cod{x:real}):real |]
        => [| $(diff ext a:cod{x:real}):real
               + $(diff ext b:cod{x:real}):real |]{x:real}
      | fn [| $(a:cod{x:real}):real * $(b:cod{x:real}):real |]
        => [| $(a:cod{x:real}):real * $(diff ext b:cod{x:real}):real
               + $(b:cod{x:real}):real
               * $(diff ext a:cod{x:real}):real |]{x:real}
      | fn [| (fn a:real => $(f:cod{a:real}):real) $(g:cod{x:real}):real |]
        => [| ((fn a:real => $(diff ext f:cod{a:real}):real)
               $(g:cod{x:real}):real)
               * $(diff ext g:cod{x:real}):real |]{x:real}
      | fn a:cod{x:real}
        => ext (diff ext) a:cod{x:real}
  |]{}
```

Además del código que representa la función a diferenciar, se toma como parámetro una función `ext` de tipo `(cod{x:real} -> cod{x:real}) -> cod{x:real} -> cod{x:real}`, que representa a una posible extensión de la función. El cuerpo de la función es esencialmente el mismo que el de su versión anterior, excepto por el caso en el cual no se concuerda con ningún patrón. En este último caso se aplica la extensión, que toma como parámetro a la función `(diff ext)` para resolver posibles llamadas recursivas.

Por lo tanto, la nueva versión de `diff` es una función que toma como parámetro un código que representa a la extensión, y ejecuta un código que representa la aplicación de la función citada `cod_diff` a la extensión:

```
val run_diff = fn ext:cod{}
    => run ( [| $(cod_diff:cod{}):((cod{x:real} -> cod{x:real})
                                -> cod{x:real} -> cod{x:real})
                                -> cod{x:real} -> cod{x:real}
               $(ext:cod{}): (cod{x:real} -> cod{x:real})
                                -> cod{x:real} -> cod{x:real} |]{}
           | fn y:cod{x:real} => y:cod{x:real})
```

Si hay algún problema de tipos dentro de la extensión, o el tipo resultante no es el esperado, no se realiza modificación alguna a la función a diferenciar.

Todas las extensiones deben tomar una función de tipo `cod{x:real} -> cod{x:real}`, que resuelve las llamadas recursivas, y un código `cod{x:real}`, que representa a la función a diferenciar. El comportamiento de la versión original de `diff` se obtendría aplicando una extensión que sólo retorne el código abstraído.

```
val cod_ext = [| fn d: cod{x:real} -> cod{x:real} =>
                    fn y:cod{x:real} => y:cod{x:real} |]{}
```

Por lo tanto, la función resultante de aplicar `run_diff` a la extensión `cod_ext`, es equivalente a la función `diff` original.

```
val diff_ext = run_diff cod_ext
```

Para realizar extensiones en la función de diferenciación sólo se debe modificar la variable `cod_ext`. Si se quiere aceptar ahora funciones que puedan contener restas y divisiones, el siguiente código implementaría la extensión.

```
val cod_ext =
    [| fn d: cod{x:real} -> cod{x:real} =>
         fn [|$(a:cod{x:real}):real - $(b:cod{x:real}):real|]
                => [| $(d a:cod{x:real}):real
                       - $(d b:cod{x:real}):real |]{x:real}
         | fn [|$(a:cod{x:real}):real / $(b:cod{x:real}):real|]
             => [| ($(b:cod{x:real}):real
                       * $(d a:cod{x:real}):real
                       - $(a:cod{x:real}):real
                       * $(d b:cod{x:real}):real)
                       / ($(b:cod{x:real}):real * $(b:cod{x:real}):real)
                  |]{x:real}
         | fn a:cod{x:rel} => a:cod{x:rel}
    |]{}
```

Para el caso de la resta y de la división se aplican las reglas de diferenciación respectivas. La variable `d`, de tipo `cod{x:real} -> cod{x:real}`, tiene una función que resuelve la llamada recursiva a la función original.

### 4.3.4. Verificación

Este ejemplo muestra como se puede utilizar la capacidad de análisis de código para realizar ciertas pruebas sobre el mismo. La función `es_grado` verifica que el grado del polinomio representado por el código que se pasa como parámetro sea el deseado, que se recibe también como parámetro.

```
val es_grado : cod{x:real} -> int -> bool
  = fn f:cod{x:real} => fn n:int => ((cant (red f:cod{x:real})) == n:int)

val rec cant : cod{x:real} -> int
  = fn [| x:real |] => 1
  | fn [| $(c:lit):real |] => 0
  | fn [| $(a:cod{x:real}):real + $(b:cod{x:real}):real |]
    => max (cant a:cod{x:real}) (cant b:cod{x:real})
  | fn [| $(a:cod{x:real}):real * $(b:cod{x:real}):real |]
    => (cant a:cod{x:real}) + (cant b:cod{x:real})
  | fn a:cod{x:real} => - MAX_INT
```

Notar que en ese ejemplo que se analiza estrictamente la estructura del polinomio, sin considerar si los coeficientes evalúan o no a valores distintos de cero. Es decir, que el resultado de `(es_grado [|(x*x) - (x*x)|]{x:real} 2)` será `true`.

# Capítulo 5

# Codificación de las Semánticas

*Tengo muchas cosas que decirte*
*y mientras no las diga ni yo mismo sé*
*mientras no las diga no son cosas fijas*
*no son ni siquiera cosas*
*no son más que nubarrones*
*negros y marrones*

Fragmento de "Metalenguaje" de Leo Masliah.

En este capítulo se codificarán las semánticas del lenguaje definido en términos del lenguaje Ωmega (cuyas bases se describen en la siguiente sección) obteniendo así un intérprete para el mismo. En base a esta codificación se probará luego que un término bien tipado nunca termina mal.

## 5.1. Lenguaje Ωmega

El isomorfismo de Curry-Howard indica que los tipos pueden ser interpretados como proposiciones y los programas como sus pruebas. Pero los tipos expresables en los lenguajes funcionales comunes no pueden expresar muchas de las propiedades de los programas. Ωmega [PL04, SP04, She04] se propone como una extensión conservativa de Haskell[1] inspirada entre otras cosas en el trabajo sobre "tipos fantasma" realizado por Cheney y Hinze [CH02, CH03, Hin03], que permite indicar y forzar propiedades interesantes de los programas utilizando el sistema de tipos. Algunas de sus características más importantes son la inclusión de los llamados **Tipos de Datos Algebraicos Generalizados** (GADTs) y un **Sistema de Kinds Extensible**[2].

Tim Sheard [She05a] propuso un procedimiento para explorar el diseño de nuevos lenguajes utilizando Ωmega como meta-lenguaje, el cual consiste en codificar las semánticas del lenguaje en términos de construcciones del meta-lenguaje y así testearlos de forma interactiva. El procedimiento es el siguiente:

**Semántica Estática** El lenguaje se define como un GADT indexado por tipos. Un GADT representa un juicio, y sus constructores codifican las reglas de tipado. La definición de nuevos *kinds* permite que los índices del GADT tengan una estructura arbitraria que corresponda a propiedades de la semántica estática del lenguaje. Estas propiedades se chequean y mantienen por el sistema de tipos de Ωmega.

[1] Aunque, a diferencia de Haskell, Ωmega es estricto y no tiene un sistema de clases.
[2] Estas características se describirán en la siguiente sub-sección.

**Semántica Dinámica** Se puede definir una semántica de paso largo como un intérprete o una función de evaluación, o una semántica de paso corto en términos de sustituciones sobre el lenguaje de términos. El sistema de tipos del meta-lenguaje garantiza que los programas del meta-nivel mantengan la seguridad del tipado del nivel objeto.

Este fue el procedimiento empleado en la etapa de diseño del lenguaje propuesto en este trabajo. En las secciones subsiguientes se mostrará la codificación de las semánticas y las pruebas de que las mismas son correctas.

### 5.1.1. Semántica Estática

A continuación mostraremos como algunas de las principales características de Ωmega pueden ser útiles para la especificación de la semántica de lenguajes de programación.

#### Tipos de Datos Algebraicos Generalizados

La sintaxis abstracta de un lenguaje de programación puede ser representada mediante tipos de datos algebraicos. Como ejemplo se ofrece un cálculo lambda con números naturales con la siguiente descripción de sintaxis abstracta:

| | | | |
|---|---|---|---|
| Tipos | $\tau \in \mathsf{T}$ | ::= | $\mathtt{nat} \mid \tau \to \tau$ |
| Contextos | $\Gamma \in \mathsf{G}$ | ::= | $\cdot \mid \Gamma, \tau$ |
| Variables | $x \in \mathsf{X}$ | ::= | $\mathtt{vz} \mid \mathtt{vs}\ x$ |
| Términos | $e \in \mathsf{E}$ | ::= | $\mathtt{z} \mid \mathtt{s}\ e \mid x \mid \lambda.e \mid e_1\ e_2$ |

la misma puede ser codificada mediante la siguiente definición de tipos:

```
data Exp = NatZ
         | NatS Exp
         | V Var
         | Abs Exp
         | App Exp Exp

data Var = VZ
         | VS Var
```

Cada constructor del tipo `Exp` representa a un término del lenguaje y el tipo `Var` representa a la categoría sintáctica de las variables. Entonces, por ejemplo, la expresión $\lambda.(\mathtt{vz})$ se representa como `(Abs (V VZ))`. Las reglas de tipado de este lenguaje se muestran en la Figura 5.1.

Utilizando el tipo `Exp` se pueden construir solamente expresiones sintácticamente correctas, por lo que expresiones como `(NatZ NatZ)` son descartadas por el chequeador de tipos del meta-lenguaje. Sin embargo otras propiedades estáticas importantes, como el tipado de las expresiones, no pueden ser capturadas por `Exp`. Por ejemplo, se pueden definir expresiones mal tipadas como `(App NatZ NatZ)` o `(NatS (Abs NatZ))`.

Una forma de capturar estas propiedades sería parametrizando el tipo `Exp` con el tipo de la expresión. Así `Exp t` sólo representaría a términos de tipo `t`. Entonces, dado un sistema de tipos como el de la Figura 5.1, se debería de poder definir a los constructores de `Exp` de manera que representen a esas reglas. Por lo que sus tipos tendrían que ser:

$$\frac{}{\Gamma, \tau \vdash \texttt{vz} : \tau}\ \text{VZ} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma, \tau' \vdash \texttt{vs}\ x : \tau}\ \text{VS}$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat}}\ \text{NatZ} \qquad \frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{s}\ e : \texttt{nat}}\ \text{NatS}$$

$$\frac{\Gamma, \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda e : \tau_1 \to \tau_2}\ \text{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}\ \text{App}$$

Figura 5.1: Reglas de Tipos del cálculo lambda simple

```
NatZ  ::  Exp Int
NatS  ::  Exp Int -> Exp Int
V     ::  ∀t. Var t -> Exp t
Abs   ::  ∀t1,t2. Exp t2 -> Exp (t1->t2)
App   ::  ∀t1,t. Exp (t1->t) -> Exp t1 -> Exp t
```

Por ejemplo, el constructor `App` sólo puede ser aplicado a una expresión de tipo función `t1->t` y otra de tipo `t1`, resultando en una expresión de tipo `t`. Por lo que expresiones erróneas como `(App NatZ NatZ)` no podrían ser definidas.

Este tipo de signaturas no se puede representar utilizando tipos de datos algebraicos simples, dado que existe la restricción de que todos los constructores de un tipo de datos tienen que tener el mismo tipo resultado, que no es el caso del tipo que se quiere definir. Los **Tipos de Datos Algebraicos Generalizados** (GADTs[3]) son una generalización de los anteriores en que se remueve esta restricción. Esto se hace utilizando una sintaxis alternativa para la declaración `data`, donde se definen explícitamente el *kind* (clasificador de tipo) del nuevo tipo y las signaturas de sus constructores. De esta manera se pueden definir tipos como los siguientes:

```
data Exp :: *0 ~> *0 where
        NatZ    :: Exp Int
        NatS    :: Exp Int -> Exp Int
        V       :: Var t -> Exp t
        Abs     :: Exp t2 -> Exp (t1->t2)
        App     :: Exp (t1->t) -> Exp t1 -> Exp t

data Var :: *0 ~> *0 where
        VZ      :: Var t
        VS      :: Var t -> Var t
```

Las únicas restricciones puestas en los tipos de los constructores son que su rango debe ser un caso completamente aplicado del tipo que se define, y que el tipo en su totalidad debe ser clasificado por `*0`, el *kind* que clasifica tipos que clasifican valores. Esto se debe a

[3]Del inglés *Generalized Algebraic Data Types*.

que, como se explicará en la siguiente sub-sección, existen tipos que no clasifican a ningún valor.

### Sistema de Kinds Extensible

De la misma forma en que los tipos clasifican valores, los tipos son clasificados por **kinds**. En los lenguajes funcionales como Haskell los kinds son implícitos. Son utilizados por el chequeador de tipos pero nunca son mencionados por el programador, y pueden ser solamente el kind base (`*0`) o kinds de alto orden (kind `~>` kind), que clasifican constructores de tipo que requieren tipos como parámetro. Entonces, si utilizamos el operador infijo (`::`) para representar la relación de clasificación, tenemos que, por ejemplo, `9::Int` expresa que 9 tiene tipo entero e `Int::*0` que el tipo entero tiene kind `*0`.

El tipo `Exp` se clasifica por el kind `*0~>*0`, lo que indica que está parametrizado por un tipo, que es el que se utiliza para codificar la propiedad del tipado de las expresiones.

En Ωmega se permite la introducción de nuevos kinds, y de nuevos tipos que sean clasificados por éstos, utilizando la declaración `kind`. Esta declaración es análoga a `data` pero en lugar de introducir constructores de valores introduce constructores de tipos que producen tipos clasificados por este kind. Estos tipos no clasifican ningún valor, por ejemplo:

```
kind Nat = Z | S Nat
```

introduce dos constructores de tipos `Z` y `S` que codifican a los números naturales a nivel de tipos. Ni el tipo `Z` ni los tipos construídos con `S` clasifican valores. Los mismos pueden ser utilizados como tipos parámetro para otros constructores de tipo. Entonces, si por ejemplo se define un kind `Row`:

```
kind Row (x::*1) = RCons x (Row x) | RNil
```

que clasifica estructuras tipo lista a nivel de tipos[4], se puede redefinir el tipo `Exp` para agregar la restricción del contexto en el cual tipa una expresión. Esto se realiza con un nuevo índice, de kind `Row *0`, que corresponde a una lista de tipos que clasifican valores. Finalmente, el tipo múltiplemente indexado `Exp gamma t` codificaría al juicio $\Gamma \vdash e : \tau$, donde el índice `gamma` representa al contexto $\Gamma$ y `t` al tipo $\tau$.

```
data Exp :: Row *0 ~> *0 ~> *0 where
        NatZ    :: Exp gamma Int
        NatS    :: Exp gamma Int
                -> Exp gamma Int
        V       :: Var gamma t
                -> Exp gamma t
        Abs     :: Exp (RCons t1 gamma) t2
                -> Exp gamma (t1->t2)
        App     :: Exp gamma (t1->t) -> Exp gamma t1
                -> Exp gamma t

data Var :: Row *0 ~> *0 ~> *0 where
        VZ      :: Var (RCons t gp) t
        VS      :: Var gp t
                -> Var (RCons tp gp) t
```

[4] Un `Row` es una "lista incompleta" que puede completarse por unificación. La notación `*1` se utiliza para expresar sorts (clasificadores de kinds) que clasifican kinds que no son de alto órden.

Por ejemplo, el constructor `Abs`, para las lambda-abstracciones, toma un parámetro que representa al juicio de tipado del cuerpo de la abstracción $\Gamma, \tau_1 \vdash e : \tau_2$. El tipo del parámetro es entonces `Exp (RCons t1 gamma) t2`, indicando que el cuerpo tiene tipo `t2` en el contexto actual (`gamma`) con el agregado de una variable de tipo `t1` en el índice 0. Si el parámetro es correcto, se devuelve un valor de tipo `Exp gamma (t1->t2)`, que codifica el juicio $\Gamma \vdash e : \tau_1 \to \tau_2$, representándose entonces la regla (Abs). Los constructores del tipo `Var`, `VZ` y `VS`, representan a las reglas de mismo nombre en la Figura 5.1.

Por ejemplo, `App (V VZ) (V (VS VZ))` tiene tipo `(Exp {a->b,a;u} b)`[5], representando a una expresión que evalúa a un valor de tipo `b`, en cualquier contexto que tenga en el índice cero a una variable de tipo `a->b` y en el siguiente a una variable de tipo `a`. Por lo que se codifica la derivación:

$$\dfrac{\text{VZ}\ \dfrac{}{\Gamma, \tau_1, \tau_1 \to \tau_2 \vdash \texttt{vz} : \tau_1 \to \tau_2} \qquad \text{VS}\ \dfrac{\text{VZ}\ \dfrac{}{\Gamma, \tau_1 \vdash \texttt{vz} : \tau_1}}{\Gamma, \tau_1, \tau_1 \to \tau_2 \vdash \texttt{vs vz} : \tau_1}}{\Gamma, \tau_1, \tau_1 \to \tau_2 \vdash \texttt{vz (vs vz)} : \tau_2}\ \text{App}$$

### 5.1.2. Semántica Dinámica

Dado que los contextos se codifican con un tipo de kind `Row *0`, definiendo el tipo:

```
data Env :: Row *0 ~> *0 where
    EnvNil :: Env RNil
    EnvCons :: t -> Env r -> Env (RCons t r)
```

se puede representar un ambiente `Env gamma`, que tiene forma `gamma`. Por ejemplo, el valor `(EnvCons 9 (EnvCons True EnvNil))`, tiene tipo `Env {Int,Bool}`, representando a un ambiente que tiene forma `{Int,Bool}`.

Dados los GADTs `Exp` y `Env` se puede definir una función de evaluación `eval` con el siguiente tipo: `Exp gamma t -> Env gamma -> t`. Esta función toma un término bien tipado, con tipo `t` en el contexto `gamma`, un ambiente con forma `gamma` y retorna un valor de tipo `t`. Desde un punto de vista denotacional el valor devuelto es el significado del término $[\![e]\!]\rho$, es decir, la evaluación de una expresión $e$ en un ambiente $\rho$. Siempre que `eval` esté definida para todos los términos, se puede decir, en base al hecho de que la función está bien tipada, que en conjunto las semánticas proveen un valor de tipo `t` para cada término bien tipado en el nivel objeto. La siguiente es la función de evaluación del cálculo lambda simple visto en la subsección anterior:

```
eval :: Exp gamma t -> Env gamma -> t
eval NatZ g = 0
eval (NatS  exp) g = (eval exp e1 e2) + 1
eval (V var) gamma = evalVar var gamma
eval (Abs body) g = \v -> (eval body (EnvCons v g))
eval (App f x) g = (eval f g) (eval x g)

evalVar :: Var gamma t -> Env gamma -> t
evalVar VZ (EnvCons v g) = v
evalVar (VS var) (EnvCons v g) = evalVar var g
```

[5] La notación `{a->b,a;u}` es una sintaxis "azucarada" para imprimir `(RCons a->b (RCons a u))`.

Observar que, debido al uso de GADTs, el tipo resultado de `eval` es en cierta manera dependiente del tipo del argumento `Exp gamma t`. Por lo tanto, si el argumento representa una expresión de cierto tipo, el resultado de `eval` será de ese mismo tipo.

Como los números naturales del cálculo se codifican con el tipo `Int` de Ωmega, la evaluación de `NatZ` en cualquier ambiente es el entero 0 y la evaluación de `NatS` es el resultado de sumarle uno a la evaluación de la sub-expresión. La aplicación y la lambda-abstracción se evalúan como aplicaciones y lambda-abstracciones de Ωmega, respectivamente. En el caso de la lambda-abstracción se agrega el valor ligado al ambiente con el cual se evalúa el cuerpo. La evaluación de las variables (`evalVar`) busca en el ambiente el valor ligado para dicha variable.

Otra forma de codificar la semántica dinámica puede ser definirla en términos de sustitución.

```
eval_2 :: Exp gamma t -> Exp gamma t
eval_2 (App f x) = case (eval_2 f) of
                     (Abs e) -> eval_2 (subst (Slash (eval_2 x)) e)
eval_2 x = x
```

La función de evaluación utiliza un GADT (`Subst gamma delta`) para representar a las sustituciones y una función auxiliar

```
subst :: Subst gamma delta -> Exp delta t -> Exp gamma t
```

para su evaluación. No se profundizará más en este trabajo con respecto a esta alternativa, dado que no fue la que se tomó en el mismo. Sheard y Pasalic [SP04] describen una implementación de `Subst` y `subst`, las cuales fueron tomadas como base para la codificación de la sustitución explícita de nuestro lenguaje.

### 5.1.3. Corrección (*Soundness*)

La corrección o *soundness* de un sistema de tipos con respecto a la semántica operacional significa que, si un término está bien-tipado, entonces su evaluación o retorna un valor del mismo tipo u ocasiona una secuencia infinita de reducciones. En otras palabras, los términos bien-tipados nunca terminan mal. Para probar corrección se deben probar las propiedades de *subject reduction* y progreso. La primera significa que la reducción conserva el tipado, mientras que progreso significa que un programa bien-tipado o es un valor o puede seguir siendo reducido (la evaluación nunca queda en un estado indefinido).

De acuerdo al tipo de la función de evaluación, `Exp gamma t -> Env gamma -> t`, la evaluación de cualquier expresión que satisface el juicio de tipado $\Gamma \vdash e : \tau$ produce, si termina, un valor de tipo $\tau$. Esto significa que el sistema de tipos de Ωmega asegura automáticamente la propiedad de *subject reduction*.

Con respecto al progreso, se puede demostrar con una simple inducción completa que la función `eval` está definida para todos los términos bien-tipados. Es decir que todo término bien tipado siempre concuerda con una de las cláusulas de `eval`. Por lo tanto, si un término no es un valor, existe una regla de reducción que es aplicable al mismo.

Entonces, como corolario de lo anterior, la implementación en Ωmega de las semánticas del lenguaje permite demostrar la corrección de su sistema de tipos.

## 5.2. Semántica Estática como un GADT de Ωmega

En esta sección se codificarán los juicios de tipado de la sección 4.1 como GADTs de Ωmega. Un valor de cada uno de estos tipos representa entonces una derivación del juicio

codificado. Esto asegura que las propiedades de la semántica estática sean chequeadas y mantenidas por el sistema de tipos del meta-lenguaje.

El juicio para las expresiones $P; \Gamma \vdash e : \tau$ se codifica con el tipo múltiplemente indexado `Exp p n t`. La pila $P$ es representada por el primer índice, un tipo producto anidado, que contiene tipos de kind `Row *0` representando a cada uno de los contextos de tipos de la pila. El contexto actual $\Gamma$ se representa también con un `Row *0`, en el índice `n`. Finalmente, `t` representa al tipo $\tau$ del término.

La codificación en Ωmega de las reglas mostradas en las Figuras 4.1, 4.3 y 4.5 es la siguiente:

```
data Exp :: *0 ~> Row *0 ~> *0 ~> *0 where
    ELBool  :: Bool -> Exp p n Bool
    ELInt   :: Int -> Exp p n Int
    EPair   :: Exp p n t -> Exp p n s
            -> Exp p n (t,s)
    EPFst   :: Exp p n (t,s)
            -> Exp p n t
    EPSnd   :: Exp p n (t,s)
            -> Exp p n s
    EAbs    :: Rep s -> Exp p (RCons s n) t
            -> Exp p n (s->t)
    EFix    :: Exp p (RCons t n) t
            -> Exp p n t
    EApp    :: Exp p n (s->t) -> Exp p n s
            -> Exp p n t
    EVar    :: Var n t -> Rep t -> Exp p n t
    ECond   :: Exp p n Bool -> Exp p n t -> Exp p n t
            -> Exp p n t
    EBr     :: Exp (p,Env n) c t -> RepEnv c
            -> Exp p n (Cod c)
    ERun    :: Exp p n (Cod RNil) -> Exp p n t
            -> Exp p n t
    EEsc    :: Exp p b (Cod n) -> Rep t
            -> Exp (p, Env b) n t
    ESubst  :: Exp p n (Cod f) -> Subst f fc
            -> Exp p n (Cod fc)
    EAlt    :: Pat s c -> Exp p {eapp c n} t -> Exp p n (s->t)
            -> Exp p n (s->t)
```

Cada constructor tiene la estructura de un juicio formal. Para representar a los tipos `int` y `bool` del lenguaje se utilizan los tipos `Int` y `Bool` de Ωmega, respectivamente. También para la representación de los tipos del producto y las funciones se utilizan los tipos análogos en Ωmega.

En `EAbs`, `EVar` y `EEsc` el tipo debe ser anotado. Ésto se realiza con un parámetro de tipo `Rep t`, que contiene una representación de los tipos usada por Cheney y Hinze [CH02] y Baars y Swierstra [BS02] para tipado dinámico:

```
data Rep:: *0 ~> *0 where
    Int     :: Rep Int
    Bool    :: Rep Bool
```

```
    Arr     :: Rep a -> Rep b -> Rep(a->b)
    Prod    :: Rep a -> Rep b -> Rep (a,b)
    Cod     :: RepEnv n -> (Rep (Cod n))
```

Estas anotaciones se utilizan durante el chequeo de tipos en tiempo de ejecución, el cual se realiza de la misma manera que hacen los autores mencionados.

El constructor `EVar` incluye el juicio `Var n t`, donde `VZ` y `VS` codifican a las reglas Base y Weak de la Figura 4.2.

```
data Var :: Row *0 ~> *0 ~> *0 where
    VZ      :: Var (RCons t env) t
    VS      :: Var env t -> Var (RCons s env) t
```

Observar que una extensión de contexto $\Gamma, \tau$ se representa por el constructor `RCons` del kind `Row`. Entonces `(EVar VZ Int)` tiene tipo `Exp p {Int;n} Int`, indicando que la expresión $\#0^{\texttt{int}}$ tiene tipo `int` en cualquier contexto que tenga un entero en el índice 0.

Los constructores `EBr` y `EEsc` realizan operaciones sobre la pila de contextos `p` del "pasado". Mientras que `EBr` desplaza el contexto `n` del tope de la pila al contexto actual, el constructor `EEsc` hace lo contrario. Las pilas de contextos se representan como pares anidados. Dado que el constructor de pares toma sólo tipos de kind `*0`, se define el tipo `Env`, que es indexado por un `Row *0`, para poder apilar contextos.

```
data Env :: Row *0 ~> *0 where
    EnvNil  :: Env RNil
    EnvCons :: t -> Env r -> Env (RCons t r)
```

Las anotaciones multi-etapas involucran expresiones con tipo $\texttt{cod}^{\Gamma f}$. La codificación de este tipo en Ωmega tiene la siguiente definición:

```
data Cod :: Row *0 ~> *0 where
    Q   :: (forall p. Exp p n t) -> RepEnv n
        -> Cod n
    F   :: RepEnv n -> Cod n
```

Debido al tipado dinámico, puede ocurrir que una expresión evalúe a un código mal formado. Por esta razón, el tipo `Cod` tiene dos constructores: uno para expresiones citadas bien formadas y otro para aquellas fallidas (`[|fail|]`$\{\Gamma^f\}$). Un código bien formado es una expresión en el nivel 0, tipada en un ambiente dado. Un término en el nivel 0 no tiene escapes en ese nivel. Esto es capturado por el requerimiento de que la pila de contextos del "pasado" sea universalmente cuantificable, dado que el constructor `EEsc`, al apilar contextos en la pila del "pasado", fuerza que la misma tenga una cierta forma. Notar que el tipo resultado del constructor `EEsc` es `Exp (p, Env b) n t`, y que `(p, Env b)` no es universalmente cuantificable. Tanto en el caso de código mal formado como para el fallido, se pasa una representación del contexto como parámetro. Esta representación tiene tipo `RepEnv`.

```
data RepEnv:: Row *0 ~> *0 where
    REnvNil     :: RepEnv RNil
    REnvCons    :: Rep t -> RepEnv r
                -> RepEnv (RCons t r)
```

Este tipo clasifica listas de `Rep t` y, al igual que `Env`, es indexado por un `Row *0`. El tipo `RepEnv` se utiliza también en el constructor `EBr` para representar a las variables libres de la expresión a citar. Por ejemplo, la representación en Ωmega de la expresión $\texttt{[|\#0}^{\texttt{int}}\texttt{|]\{int\}}$ es `(EBr (EVar VZ Int) (REnvCons Int REnvNil))`. Dado que en cualquier contexto y bajo cualquier pila evalúa a un código de tipo $\texttt{cod}^{\cdot,\texttt{int}}$ el tipo de la representación es `Exp p n Cod{Int}`. De acuerdo al tipo del constructor `EBr`, el parámetro `(EVar VZ Int)` debe tener tipo `Exp (p,Env n) {Int} Int`. Para que esto se cumpla, entonces `(VZ Int)` tiene que ser de tipo `Var {Int} Int`. Esto último se cumple porque el tipo del constructor `VZ` es `Var {t',n'} t'` y se pueden unificar usando `t'=Int` y `n'=EnvNil`. Observar que lo anterior es una codificación de la derivación:

$$\dfrac{\dfrac{\dfrac{}{\cdot, \texttt{int} \vdash \#0 : \texttt{int}}\ \text{VZ}}{(P,\Gamma); \cdot, \texttt{int} \vdash \#0^{\texttt{int}} : \texttt{int}}\ \text{Var}}{P;\Gamma \vdash \texttt{[|}\#0^{\texttt{int}}\texttt{|]\{int\}} : \texttt{cod}^{\texttt{int}}}\ \text{Br}$$

El juicio de las sustituciones se codifica por el tipo de datos `Subst`, cuyos constructores representan a las reglas de la Figura 4.4. Como en el constructor `Q` para el tipo `Cod`, la expresión pasada al constructor `SSlsh` debe tener una pila de contextos del "pasado" universalmente cuantificada.

```
data Subst :: Row *0 ~> Row *0 ~> *0 where
    SSft    :: Rep t -> Subst n (RCons t n)
    SLft    :: Subst n c
            -> Subst (RCons t n) (RCons t c)
    SSlsh   :: (forall p. Exp p n t)
            -> Subst (RCons t n) n
```

Por ejemplo, la sustitución explícita $(\texttt{[|}(\#0^{\texttt{int}}, \#1^{\texttt{bool}})\texttt{|]\{bool,int\}})\texttt{[}\Uparrow(\texttt{True}/)\texttt{]}$, se codifica en Ωmega como:

```
(ESubst
    (EBr
        (EPair (EVar VZ Int) (EVar (VS VZ) Bool))
        (REnvCons Int (REnvCons Bool REnvNil)))
    (SLft (SSlsh (ELBool True))))
```

En el constructor `EAlt` se utiliza la función de tipos[6] `eapp` para codificar una restricción de concatenación de listas $(\Gamma', \Gamma'')$. Mediante una inducción simple en el primer parámetro se puede probar que esta función termina.

```
eapp :: Row *0 ~> Row *0 ~> Row *0
{eapp RNil ys} = ys
{eapp (RCons x xs) ys} = RCons x {eapp xs ys}
{eapp {eapp xs ys} zs} = {eapp xs {eapp ys zs}}
```

El juicio para los patrones $\vdash p : \tau \Rightarrow \Gamma$ se codifica por el tipo de datos `(Pat t n)`. Cada constructor del tipo equivale a una de las reglas de la Figura 4.6.

[6]Las funciones de tipos son funciones en el nivel de los tipos, que se definen mediante un conjunto de ecuaciones. De esta forma se generan y propagan restricciones de igualdad de tipos.

```
data Pat :: *0 ~> Row *0 ~> *0 where
    PLInt   :: Int -> Pat Int RNil
    PLBool  :: Bool -> Pat Bool RNil
    PPair   :: Pat t1 c1 -> Pat t2 c2
            -> Pat (t1,t2) {eapp c2 c1}
    PVar    :: Rep t -> Pat t (RCons t RNil)
    PAny    :: Pat t RNil
    PCod    :: PatCod f c
            -> Pat (Cod f) c
```

El constructor `PCod` incluye un juicio para los patrones de código. La definición del tipo `PatCod`, que representa al juicio de los patrones de código (Figura 4.7), es el siguiente:

```
data PatCod :: Row *0 ~> Row *0 ~> *0 where
    PCPAny  :: PatCod f RNil
    PCPFail :: PatCod f RNil
    PCPVar  :: Rep t -> PatCod f (RCons (Cod f) RNil)
    PCPLit  :: Rep t -> PatCod f (RCons (Cod f) RNil)
    PCLInt  :: Int -> PatCod f RNil
    PCLBool :: Bool -> PatCod f RNil
    PCPair  :: PatCod f c1 -> PatCod f c2
            -> PatCod f {eapp c2 c1}
    PCFst   :: PatCod f c
            -> PatCod f c
    PCSnd   :: PatCod f c
            -> PatCod f c
    PCAbs   :: Rep s -> PatCod (RCons s f) c
            -> PatCod f c
    PCFix   :: Rep s -> PatCod (RCons s f) c
            -> PatCod f c
    PCApp   :: PatCod f c1 -> PatCod f c2
            -> PatCod f {eapp c2 c1}
    PCVar   :: Var vn t -> Rep t -> PatCod f RNil
    PCCond  :: PatCod f c1 -> PatCod f c2 -> PatCod f c3
            -> PatCod f {eapp3 c3 c2 c1}

    PCBr    :: RepEnv fp -> PatCod f (RCons (Cod f) RNil)
    PCSubst :: PatCod fc c -> Subst f fc
            -> PatCod fc c
    PCRun   :: PatCod f c1 -> PatCod f c2
            -> PatCod f {eapp c2 c1}

    PCAlt   :: Rep s -> Pat s fp
            -> PatCod {eapp fp f} c1 -> PatCod f c2
            -> PatCod f {eapp c2 c1}
```

En el constructor `PCCond` que corresponde al patrón de código de las condiciones, se utiliza la función de tipos `eapp3`, con la siguiente implementación:

```
eapp3 :: Row *0 ~> Row *0 ~> Row *0 ~> Row *0
```

```
{eapp3 xs ys zs} = {eapp xs {eapp ys zs}}
```

Se puede demostrar que esta función termina, dado que `eapp` termina.

Si consideramos la expresión:

$$\begin{array}{l}\lambda\texttt{[|if } \#0^{\texttt{bool}} \texttt{ then \$(•)}^{\texttt{int}} \texttt{ else \$(•)}^{\texttt{int}}\texttt{|]} \\ \quad . \;\; \#1^{\texttt{cod}\cdot,\texttt{bool}}\texttt{[true/]} \\ \quad | \;\; \lambda^{\texttt{cod}\cdot,\texttt{bool}} . \;\; \#0^{\texttt{cod}\cdot,\texttt{bool}}\texttt{[true/]}\end{array}$$

su codificación en Ωmega es la siguiente:

```
(EAlt
    (PCod (PCCond (PCVar VZ Bool) (PCPVar Int) (PCPVar Int)))
    (ESubst (EVar (VS VZ) (Cod (REnvCons Bool REnvNil)))
        (SSlsh (ELBool True)))
    (EAbs (Cod (REnvCons Bool REnvNil))
        (ESubst (EVar VZ (Cod (REnvCons Bool REnvNil)))
            (SSlsh (ELBool True)))))
```

que tiene tipo `Exp p n (Cod {Bool} -> Cod RNil)`.

## 5.3. Semántica Dinámica como un Intérprete Ωmega

La función de evaluación tiene tipo `Exp p n t -> Env n -> t`. Dada una expresión bien tipada `Exp p n t` y un ambiente con forma `n`, `eval` retorna un valor con tipo `t`. En esta sección se mostrarán sólo fragmentos de la función de evaluación y sus funciones auxiliares, la implementación completa se encuentra en el Apéndice B.

```
eval :: Exp p n t -> Env n -> t
eval (ELInt i) env      = i
...
eval (EPair e1 e2) env  = (eval e1 env,eval e2 env)
eval (EPFst e) env      = fst (eval e env)
...
eval (EAbs t e) env     = \ v -> eval e (EnvCons v env)
eval (EApp f x) env     = (eval f env)(eval x env)
eval (EVar v t) env     = evalVar v t env
...
eval (EFix e) env       = lazy ((\ v -> (eval e (EnvCons v env)))
                                (eval (EFix e) env))
eval (EBr e renv) env   = case (bd (CountBrZ env) e) of
                            (x,True) -> Q x renv
                            (x,False) -> F renv
eval (ERun e1 e2) env   = case (eval e1 env) of
                            (Q e REnvNil) ->
                                case eqType (getType e) (getType e2) of
                                    Just Eq -> eval e EnvNil
                                    Nothing -> eval e2 env
                            _ -> eval e2 env
eval (ESubst e s) env   = case (eval e env) of
                            (Q eb rb) ->
```

```
                                  case (evalSub s eb rb) of
                                      (en,rn) -> (Q en rn)
                              (F rb) -> F (evalSubR s rb)
eval (EAlt p e1 e2) env = \ v -> case (evalPat p v env) of
                                      Just env2 -> eval e1 env2
                                      Nothing -> (eval e2 env) v
```

Esta función analiza todos los casos, con la excepción de `EEsc`, el cual no es evaluado. En una expresión en el nivel 0 no habrá escapes, así que dicha función debe ser definida para tomar expresiones en el nivel 0. Esto puede ser forzado definiendo una función de evaluación que sólo pueda ser aplicada sobre términos polimórficos en su pasado (en otras palabras, no dependientes de un pasado).

```
eval0 :: (forall p. Exp p n t) -> Env n -> t
eval0 exp env = eval exp env
```

Por ejemplo, la expresión `(EEsc (EBr (ELInt 9) REnvNil) Int)` no es aceptada por la función de evaluación `eval0`, dado que tiene tipo `Exp (a,Env b) RNil Int`, y que no tiene la forma `(forall p. Exp p n t)`.

Tanto el meta-lenguaje como el lenguaje objeto son estrictos, por esto se implementa la aplicación y la lambda-abstracción en términos de la aplicación y la lambda-abstracción de Ωmega. En el caso de la lambda-abstracción, el valor ligado se agrega al ambiente en el cual se evalúa el cuerpo. Al evaluarse las variables, utilizando la función `evalVar`, se busca su valor en el lugar correspondiente del ambiente.

```
evalVar :: Var n t -> Rep t -> Env n -> t
evalVar VZ t (EnvCons v vs)     = v
evalVar (VS v) t (EnvCons x xs) = evalVar v t xs
```

En la evaluación de `EFix`, se utiliza el constructor Ωmega de evaluación perezosa explícita (`lazy`) para evitar iteraciones infinitas.

### 5.3.1. Chequeo Dinámico de Tipos y Construcción de Código

El chequeo de tipos se implementa con la función de unificación `eqType`, la cual toma dos representaciones de tipo, verifica que tengan una igualdad estructural, y posiblemente retorna una prueba de su equivalencia. La función es:

```
eqType = eqTAux where
    eqTAux :: Rep a -> Rep b -> Maybe (Equal a b)
    eqTAux Int Int = Just Eq
    eqTAux Bool Bool = Just Eq
    eqTAux (Arr a b) (Arr m n) =
            do { Eq <- eqTAux a m
            ; Eq <- eqTAux b n
            ; return Eq }
    eqTAux (Prod a b) (Prod m n) =
            do { Eq <- eqTAux a m
            ; Eq <- eqTAux b n
            ; return Eq }
    eqTAux (Cod REnvNil) (Cod REnvNil) = Just Eq
```

```
eqTAux (Cod (REnvCons e1 r1)) (Cod (REnvCons e2 r2)) =
        do{ Eq <- eqTAux e1 e2
        ; Eq <- eqTAux (Cod r1) (Cod r2)
        ; return Eq }
eqTAux _ _ = Nothing
monad maybeM
```

La declaración `monad maybeM` al final de la cláusula `where` de la función indica que se utiliza la mónada `maybeM` dentro de la misma[7].

Un valor de tipo `Equal a b` es un testigo dinámico de que se cumple la propiedad estática de que el tipo `a` es igual al tipo `b`. Los testigos, que pueden ser implementados mediante GADTs de Ωmega, son valores con un tipo parametrizado, cuya existencia asegura que se cumple una cierta relación entre sus parámetros de tipo. Por lo tanto si los tipos unifican correctamente, se puede construir y retornar un valor de tipo `Equal a b`; en otro caso se debe retornar `Nothing`.

Durante la evaluación de `ERun`, luego de la verificación de que el código está bien formado, se realiza la unificación entre los tipos de la expresión citada `e` y la expresión de excepción `e2`. Si la unificación es exitosa, existe un testigo de que el tipo de `e` es el mismo que el de `e2`. Por lo tanto, se evalúa la expresión `e` en el ambiente vacío (durante el chequeo de tipos estático se asegura que `e` es cerrada). Si la unificación falla, se evalúa la expresión `e2` en el ambiente `env`. Los tipos de `e` y `e2` se obtienen por la función de inferencia de tipos `getType`, la cual se basa en las reglas de tipado de las expresiones:

```
getType:: Exp p n t -> Rep t
getType (ELInt i)       = Int
getType (ELBool b)      = Bool
getType (EPair e1 e2)   = (Prod (getType e1) (getType e2))
getType (EPFst e)       = case (getType e) of
                            (Prod r1 r2) -> r1
...
getType (EAbs t e)      = (Arr t (getType e))
getType (EApp e1 e2)    = case (getType e1) of
                            (Arr r1 r2) -> r2
getType (EVar v t)      = t
...
getType (EBr e renv)    = Cod renv
getType (ESubst e s)    = case (getType e) of
                            (Cod renv) -> Cod (evalSubR s renv)
getType (ERun e1 e2)    = (getType e2)
getType (EEsc e t)      = t
getType (EAlt p e1 e2)  = (getType e2)
```

Observar que las anotaciones de tipos y de contextos de tipos se utilizan en el algoritmo de inferencia.

[7] Las mónadas en Ωmega son tipos de datos algebraicos con constructor `Monad` y las funciones `return`, `bind` y `fail`. La definición de la mónada `maybeM` es la siguiente:

```
maybeM = (Monad Just bind fail) where
    return x = Just x
    fail s = Nothing
    bind Nothing g = Nothing
    bind (Just x) g = g x
```

Evaluar `EBr` involucra la evaluación de plantillas (*templates*) de código de manera de evaluar una expresión polimórfica en su "pasado". Esto se realiza con la función `bd`, que es la definida por Sheard [She05a] con el agregado de chequeo de tipos dinámico.

```
data CountBr :: *0 ~> *0 ~> *0 where
    CountBrZ :: a -> CountBr (b,a) c
    CountBrS :: CountBr a b -> CountBr (a,c) (b,c)


bd :: CountBr a z -> Exp a n t -> (Exp z n t,Bool)
bd env (ELInt i)                = (ELInt i,True)
...
bd env (EPair e1 e2)            = let   (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                  in (EPair x1 x2, b1 && b2)
...
bd env (EBr e renv)             = let (x,b) = (bd (CountBrS env) e)
                                  in (EBr x renv, b)
...
bd (CountBrZ env) (EEsc e t)    = case (eval e env) of
                                    (Q x renv) ->
                                        case eqType (getType x) t of
                                            Just Eq -> (x,True)
                                            Nothing -> (getAny t,False)
                                    _ -> (getAny t,False)
bd (CountBrS r) (EEsc e t)      = let (x,b) = (bd r e)
                                  in (EEsc x t, b)
```

Esencialmente, esta función atraviesa una expresión, generando una copia sin escapes embebidos en el primer nivel y un booleano que expresa si el código producido está bien tipado. El parámetro de tipo `CountBr` cuenta los paréntesis que rodean a la expresión. Cuando se encuentra un `EBr` se incrementa el contador. Para `(EEsc e t)`, si el contador de paréntesis es `(CountBrS env)`, el mismo se decrementa. Si el contador es cero, se evalúa la expresión `e`, si `e` está bien formada y el tipo es el esperado, se retorna el código resultante. En otro caso se genera una expresión *dummy*, con tipo `t`, utilizando la función `getAny :: Rep t -> Exp p n t`.

A continuación se mostrarán las versiones en Ωmega de algunas de las evaluaciones de ejemplo de la sección 4.2.2.

- La expresión `run [|9|]{}|0` evalúa a 9 en el nivel cero. Esta evaluación se escribe en Ωmega de la siguiente forma:

  `eval0 (ERun (EBr (ELInt 9) REnvNil) (ELInt 0))`
  lo cual implica evaluar la primera subexpresión del run:

  `eval (EBr (ELInt 9) REnvNil) EnvNil`
  cuyo resultado, dado que `bd (CountBrZ EnvNil) (ELInt 9)` no modifica a la expresión citada, es `Q (ELInt 9) REnvNil`. Como el tipo de la expresión `(ELInt 9)` y de la expresión de excepción es el mismo (`Int`) entonces la función:

  `eqType (getType (ELInt 9)) (getType (ELInt 0))`
  retorna el valor testigo `Just Eq`. Siendo entonces el resultado de la evaluación el de evaluar `eval (ELInt 9) EnvNil`, que es el valor entero 9.

- La expresión `run [|true|]{}|0` evalúa a 0, dado que la unificación de tipos falla, como se verá a continuación. La evaluación:

  `eval0 (ERun (EBr (ELBool True) REnvNil) (ELInt 0))`
  implica, siguiendo pasos similares al caso anterior, analizar el valor `Q (ELBool True) REnvNil`. En este caso la unificación de tipos falla, ya que el tipo de la expresión `(ELBool True)` es distinto al de la expresión de excepción. Entonces la función:

  `eqType (getType (ELBool True)) (getType (ELInt 0))`
  retorna `Nothing`, por lo que el resultado de la evaluación es el de `eval (ELInt 0) EnvNil`, que es el valor entero 0.

- La expresión `run [|$([|true|]{})^int|]{}|0` evalúa a 0. En este caso, porque el tipado de la expresión citada falla. Al momento de ser evaluada la primera expresión:

  `eval (EBr (EEsc (EBr (ELBool True) REnvNil) Int) REnvNil) EnvNil`
  se debe invocar a la función que construye el código y verifica el tipado:

  `bd (CountBrZ EnvNil) (EEsc (EBr (ELBool True) REnvNil) Int)`
  Dado que se encuentra con un caso de escape, y el contador de paréntesis está en cero, se debe evaluar la subexpresión del escape para obtener una expresión citada y verificar que el tipo sea el que se pide. Esta evaluación resulta en el valor `(Q (ELBool True) REnvNil)`, por lo que los tipos no unifican, y la función `bd` retorna `(getAny Int,False)`. Entonces, la expresión paréntesis evalúa al valor de código fallido `(F EnvNil)`. Como este valor no tiene la forma `(Q e REnvNil)`, concuerda con el patrón `_` en la evaluación del `ERun`, y el resultado es el de evaluar `(eval (ELInt 0) EnvNil)`, que es el valor entero 0.

- En el caso anterior, si la primera expresión del `run` estuviera rodeada por un par de paréntesis más, el tipado no fallaría. Al evaluar:

  `(EBr (EBr (EEsc (EBr (ELBool True) REnvNil) Int) REnvNil) REnvNil)`
  se invoca la función de construcción de código:

  `bd (CountBrZ EnvNil) (EBr (EEsc (EBr (ELBool True) REnvNil) Int) REnvNil)`
  que se invoca recursivamente con la sub-expresión y el contador de paréntesis incrementado en uno:

  `bd (CountBrS (CountBrZ EnvNil)) (EEsc (EBr (ELBool True) REnvNil) Int)`
  En este caso se encuentra un `EEsc`, pero con el contador de paréntesis mayor a cero, por lo tanto su resultado es `((EEsc (EBr (ELBool True) REnvNil) Int),True)`. Este resultado se coloca en la vuelta de la recursión:

  `((EBr (EEsc (EBr (ELBool True) REnvNil) Int) REnvNil),True)`
  entonces el resultado de la evaluación de la primera expresión es:

  `Q (EBr (EEsc (EBr (ELBool True) REnvNil) Int) REnvNil) REnvNil`
  que es el valor `[|[|$([|true|]{})^int|]{}|]{}`. Dado que este resultado es de tipo código, y el tipo esperado en el `run` es entero, el resultado es igualmente el valor entero 0.

### 5.3.2. Sustitución Explícita

La evaluación de la sustitución explícita

$$\frac{e \overset{0}{\hookrightarrow} \texttt{[|}v_1\texttt{|]}\{\Gamma_1\} \qquad \Gamma_1 \vdash \Theta : v_1 \Longrightarrow v_2 \qquad \Gamma_1 \vdash \Theta \Rightarrow \Gamma_2}{e[\Theta] \overset{0}{\hookrightarrow} \texttt{[|}v_2\texttt{|]}\{\Gamma_2\}}$$

se implementa en Ωmega de la siguiente manera:

```
eval (ESubst e s) env = case (eval e env) of
                            (Q eb rb) ->
                                case (evalSub s eb rb) of
                                    (en,rn) -> (Q en rn)
                            (F rb) -> F (evalSubR s rb)
```

La función auxiliar `evalSub` se divide en dos funciones: una que aplica la sustitución a la expresión y otra que la aplica a la representación del contexto.

```
evalSub :: Subst g gp -> Exp p g t -> RepEnv g
        -> (Exp p gp t,RepEnv gp)
evalSub s e renv    = (evalSubE s e renv, evalSubR s renv)
```

Por ejemplo, la sustitución `[|`$(\#0^{\texttt{int}}, \#1^{\texttt{bool}})$`|]{bool,int}`$[\Uparrow$ `true`$/]$ se evalúa en Ωmega de la siguiente manera:

```
eval  (ESubst  (EBr (EPair (EVar VZ Int) (EVar (VS VZ) Bool))
                       (REnvCons Int (REnvCons Bool REnvNil)))
               (SLft (SSlsh (ELBool True))))
       EnvNil
```

que implica la evaluación de la expresión `EBr`, cuyo resultado es:

```
(Q   (EPair (EVar VZ Int) (EVar (VS VZ) Bool))
     (REnvCons Int (REnvCons Bool REnvNil)))
```

Dado este código, se debe invocar a la función `evalSub`, la cual a su vez realiza las siguientes invocaciones de las funciones auxiliares `evalSubE` y `evalSubR`:

```
(evalSubE   (SLft (SSlsh (ELBool True)))
            (EPair (EVar VZ Int) (EVar (VS VZ) Bool))
            (REnvCons Int (REnvCons Bool REnvNil)))
(evalSubR   (SLft (SSlsh (ELBool True)))
            (REnvCons Int (REnvCons Bool REnvNil)))
```

En las siguientes subsecciones se detallará la implementación de las funciones auxiliares.

### Representaciones de Contextos

La función que aplica la sustitución a las representaciones de contextos toma la codificación del juicio $\Gamma_1 \vdash \Theta \Rightarrow \Gamma_2$ de una sustitución $\Theta$, la representación de un contexto $\Gamma_1$ y devuelve la representación del contexto $\Gamma_2$.

```
evalSubR :: Subst g gp -> RepEnv g -> RepEnv gp
evalSubR (SSlsh e) (REnvCons t r)   = r
evalSubR (SSft t) r                 = (REnvCons t r)
evalSubR (SLft s) (REnvCons t r)    = (REnvCons t (evalSubR s r))
```

Esta función se separa en tres casos. En el caso de la barra $(e/)$ se elimina el primer tipo, en el caso de desplazamiento $(\uparrow^{\tau})$ el nuevo tipo se agrega al principio, y, en el caso

de levantar (⇑), el primer tipo se deja invariado y la sustitución se aplica recursivamente al resto del contexto. Entonces, la invocación del ejemplo:

```
(evalSubR   (SLft (SSlsh (ELBool True)))
            (REnvCons Int (REnvCons Bool REnvNil)))
```

resulta en la representación de contexto `(REnvCons Int REnvNil)`.

### Expresiones

La base de la función de sustitución sobre las expresiones es la definida por Sheard y Pasalic [SP04], extendida por el pasaje de la representación del ambiente origen. Toma una sustitución $\Gamma_1 \vdash \Theta \Rightarrow \Gamma_2$, una expresión en el nivel 0, con juicio $P; \Gamma_1 \vdash v_1 : \tau$, la representación del contexto $\Gamma_1$, y retorna la expresión resultante de la sustitución, con juicio $P; \Gamma_2 \vdash v_2 : \tau$. De esta forma se implementa la evaluación $\Gamma_1 \vdash \Theta : v_1 \Longrightarrow v_2$.

```
evalSubE :: Subst g gp -> (forall p. Exp p g t) -> RepEnv g
            -> Exp p gp t
evalSubE s (ELInt i) r      = ELInt i
...
evalSubE s (EPair e1 e2) r  = EPair (evalSubE s e1 r) (evalSubE s e2 r)
...
evalSubE s (EAbs t e) r     = EAbs t (evalSubE (SLft s) e (REnvCons t r))
evalSubE s (ESubst e sp) r  = ESubst (evalSubE s e r) sp
evalSubE (SSlsh e) (EVar VZ t) (REnvCons _ r)       = e
evalSubE (SSlsh e) (EVar (VS v) t) (REnvCons tp r)  = (EVar v t)
evalSubE (SLft s) (EVar VZ t) (REnvCons _ r)        = (EVar VZ t)
evalSubE (SLft s) (EVar (VS v) t) (REnvCons tp r)   =
            evalSubE (SSft tp) (evalSubE s (EVar v t) r) (evalSubR s r)
evalSubE (SSft tp) (EVar v t) (REnvCons _ r)        = (EVar (VS v) t)
```

Entonces, para evaluar el efecto de la sustitución sobre la expresión de ejemplo:

```
(evalSubE   (SLft (SSlsh (ELBool True)))
            (EPair (EVar VZ Int) (EVar (VS VZ) Bool))
            (REnvCons Int (REnvCons Bool REnvNil)))
```

se debe realizar recursivamente la evaluación de las dos sub-expresiones del par. En el caso de la primera sub-expresión:

```
(evalSubE   (SLft (SSlsh (ELBool True)))
            (EVar VZ Int)
            (REnvCons Int (REnvCons Bool REnvNil)))
```

el resultado es `(EVar VZ Int)`. Para la segunda sub-expresión:

```
(evalSubE   (SLft (SSlsh (ELBool True)))
            (EVar (VS VZ) Bool)
            (REnvCons Int (REnvCons Bool REnvNil)))
```

como la sustitución es `SLft` y la expresión es una variable con índice mayor a cero, se debe realizar la evaluación:

```
(evalSubE   (SSft Int)
            (evalSubE (SSlsh (ELBool True)) (EVar VZ Bool)
                      (REnvCons Bool REnvNil))
            (evalSubR (SSlsh (ELBool True))
                      (REnvCons Int (REnvCons Bool REnvNil))))
```

donde la evaluación de la sustitución `SSlsh (ELBool True)` sobre una variable de índice cero resulta en `(ELBool True)` y sobre la representación del contexto `(REnvCons Int (REnvCons Bool REnvNil))` resulta en `(REnvCons Bool REnvNil))`. Teniendo entonces la sustitución:

```
(evalSubE (SSft Int) (ELBool True) (REnvCons Bool REnvNil))
```

cuyo resultado es `(ELBool True)`. Por lo tanto, la evaluación de la sustitución del ejemplo resulta en el código:

```
(Q (EPair (EVar VZ Int) (ELBool True)) (REnvCons Int REnvNil))
```

La evaluación de la sustitución sobre expresiones entre paréntesis implica evaluar una plantilla de código con una función `bds`, similar a `bd` de la sección 5.3.1, que atraviesa la expresión y aplica la sustitución cuando el contador de paréntesis es cero.

```
evalSubE s (EBr e renv) r   = EBr (bds (CountSBrZ s r) e) renv
```

Por ejemplo, la siguiente sustitución:

```
(evalSubE   (SSlsh (EBr (ELInt 1) REnvNil))
            (EBr (EEsc (EVar VZ (Cod REnvNil)) Int) REnvNil)
            (REnvCons (Cod REnvNil) REnvNil))
```

llama a la función `bds` con el contador de paréntesis en cero para la sub-expresión del `EBr`:

```
bds   (CountSBrZ
          (SSlsh (EBr (ELInt 1) REnvNil))
          (REnvCons (Cod REnvNil) R EnvNil)))
       (EEsc (EVar VZ (Cod REnvNil)) Int)
```

como se encuentra en el caso `EEsc`, y el contador es cero, se debe evaluar el efecto de la sustitución que almacena el contador sobre la sub-expresión `(EVar VZ (Cod REnvNil))`, cuyo resultado es `(EBr (ELInt 1) REnvNil`. Por lo tanto la función `bds` retorna la expresión `(EEsc (EBr (ELInt 1) REnvNil) Int)` y finalmente el resultado de la sustitución es `(EBr (EEsc (EBr (ELInt 1) REnvNil) Int) REnvNil)`.

Para aplicar una sustitución a una expresión de alternación se siguen los siguientes pasos. Primero, el patrón se deja sin cambios. Luego, de forma similar a como se hace en la expresión de abstracción pero en un caso más general, se evalúa el efecto del patrón sobre la sustitución y la representación del contexto de la expresión concordada. La nueva sustitución se aplica luego a la expresión concordada con la nueva representación de contexto. Finalmente, se evalúa la sustitución original sobre la expresión alternativa.

```
evalSubE s  (EAlt p e1 e2) r = case (getType e2) of
                                 (Arr t1 t2) -> EAlt p
```

```
                                        (evalSubE (evalPatS p s)
                                            e1 (evalPatR p r t1))
                                        (evalSubE s e2 r)
```

La función auxiliar `evalPatR` devuelve el efecto de un patrón sobre la representación del contexto de "entrada". Esto se realiza atravesando el patrón y agregando un tipo por cada sub-patrón que liga variables. El tipo de `evalPatR` contempla la decisión tomada en la sección 4.1.3 de incluir sólo la extensión de los contextos de tipos en los juicios de patrones. El efecto de un patrón, con juicio $\vdash p : \tau_1 \Rightarrow \Gamma'$, sobre un contexto $\Gamma_1$ es un contexto $\Gamma_1, \Gamma'$.

```
evalPatR    :: (Pat t eout) -> RepEnv ein -> Rep t
            -> RepEnv {eapp eout ein}
```

Observar que esa decisión simplifica la definición de la operación de evaluación de la sustitución. Por ejemplo, la signatura de las función auxiliar `evalPatS`, que evalúa el efecto de un patrón sobre la sustitución a realizar sobre la primera rama de una alternación, es simplemente:

```
evalPatS    :: (Pat t eout) -> Subst g gp
            -> Subst {eapp eout g} {eapp eout gp}
```

Dados un patrón $p$, con juicio $\vdash p : \tau_1 \Rightarrow \Gamma'$, y una sustitución $\Theta$, con juicio $\Gamma_1 \vdash \Theta \Rightarrow \Gamma_2$, evaluar el efecto de $p$ sobre $\Theta$ resulta en una sustitución $\Theta'$ con juicio $\Gamma_1, \Gamma' \vdash \Theta' \Rightarrow \Gamma_2, \Gamma'$, manteniéndose aislado el efecto de $p$ sobre $\Gamma_1$ y $\Gamma_2$.

### 5.3.3. Concordancia de Patrones

La evaluación de la alternación `(EAlt p e1 e2)` se realiza mediante la evaluación del patrón `p`, y `e1` o `e2` dependiendo del resultado de la concordancia de patrones.

```
eval (EAlt p e1 e2) env = \ v -> case (evalPat p v env) of
                                    Just env2 -> eval e1 env2
                                    Nothing -> (eval e2 env) v
```

La función de evaluación de la concordancia de patrones, `evalPat`, tiene tres parámetros: un juicio de patrón de tipo `(Pat t eout)`, un valor de tipo `t`, para concordar con el patrón, y un ambiente de entrada con tipo `Env ein`. Si e la concordancia de patrones se realiza con éxito, la función retorna el valor `(Just env)`, siendo `env` el ambiente extendido (con tipo `Env {eapp eout ein}`), y se evalúa `e1` en ese ambiente. Si la concordancia falla, se retorna un valor `Nothing`, y se evalúa `e2` en el ambiente actual.

Por ejemplo, si al evaluar el patrón `(PLInt i)` se le pasa el valor `i`, se retorna el mismo ambiente pasado como parámetro. Por otra parte, evaluar `(PVar t)` nunca falla, simplemente se retorna el ambiente actual extendido con el valor pasado.

```
evalPat :: (Pat t eout) -> t -> Env ein -> Maybe (Env {eapp eout ein})

evalPat (PLInt i) v env = if (i==v)
                          then Just env
                          else Nothing
```

```
...
evalPat (PPair p1 p2) (v1,v2) env = case (evalPat p1 v1 env) of
                                        Just env1 -> evalPat p2 v2 env1
                                        Nothing     -> Nothing
evalPat (PVar t) v env  = Just (EnvCons v env)
evalPat PAny v env  = Just env
evalPat (PCod p) v env  = evalCPat p v env
```

En el caso de `(PPair p1 p2)`, se evalúa el patrón `p1` con el ambiente actual y luego se evalúa `p2` con el ambiente retornado por `p1`. Los patrones de código se evalúan utilizando la función `evalCPat`.

```
evalCPat    :: (PatCod f eout) -> (Cod f) -> Env ein
            -> Maybe (Env {eapp eout ein})

evalCPat (PCPVar t) e env = case e of
                                (Q v renv) ->
                                    case (eqType (getType v) t) of
                                        Just Eq -> (Just (EnvCons e env))
                                        Nothing -> Nothing
                                _ -> Nothing
evalCPat PCPAny e env = (Just env)
evalCPat PCPFail e env = case e of
                            (F renv) -> Just env
                            (Q eq renv) -> Nothing
evalCPat (PCLit t)  e env = case e of
                                (Q (ELInt i) renv) ->
                                    case (eqType Int t) of
                                        Just Eq -> (Just (EnvCons e env))
                                        Nothing -> Nothing
                                (Q (ELBool b) renv) ->
                                    case (eqType Bool t) of
                                        Just Eq -> (Just (EnvCons e env))
                                        Nothing -> Nothing
                                _ -> Nothing
evalCPat (PCLInt i)  e env = case e of
                                (Q (ELInt v) renv) -> if (i==v)
                                                        then (Just env)
                                                        else Nothing
                                _ -> Nothing
...
evalCPat (PCPair p1 p2) e env =
                case e of
                    (Q (EPair v1 v2) renv) ->
                        case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                            (Just env1) ->
                                evalCPat p2(eval (EBr v2 renv) env) env1
                            Nothing -> Nothing
                    _ -> Nothing
```

```
...
evalCPat (PCAbs tx pb) e env =
                case e of
                    (Q (EAbs tvx vb) renv) ->
                        case (eqType tvx tx) of
                            (Just Eq) -> evalCPat pb
                                            (eval (EBr vb
                                            (REnvCons tvx renv)) env) env
                            Nothing -> Nothing
                    _ -> Nothing
...
evalCPat (PCVar v t)  e env = case e of
                                (Q (EVar vv vt) renv) ->
                                    case (eqType t vt) of
                                        Just Eq ->  if (eqVar v vv)
                                                        then Just env
                                                        else Nothing
                                        Nothing -> Nothing
                                _ -> Nothing
...
evalCPat (PCBr r) e env = case e of
                            (Q (EBr ve renv2) renv) ->
                                case (eqType (Cod r) (Cod renv2)) of
                                    Just Eq -> Just env
                                    Nothing -> Nothing
                            _ -> Nothing
...
evalCPat (PCSubst p s)  e env =
                case e of
                    (Q (ESubst ve vs) renv) ->
                        if (eqSubst s vs)
                            then (evalCPat p (eval (EBr ve renv) env) env)
                            else Nothing
                    _ -> Nothing


evalCPat (PCAlt t p p1 p2)  e env =
        case e of
            (Q (EAlt vp v1 v2) renv) ->
                case (getType v2) of (Arr t1 t2) ->
                    case (eqPat p vp t t1) of
                        Just Eq ->
                            case evalCPat p1
                                        (eval (EBr v1
                                                (evalPatR vp renv t)) env)
                                        env of
                                Just env1 -> evalCPat p2
                                                (eval (EBr v2 renv) env)
                                                env1
```

```
                                Nothing -> Nothing
                        Nothing -> Nothing
        _ -> Nothing
```

Consideremos el caso de `(PCAbs tx pb)`. Si el valor pasado es un código que tiene forma `(Q (EAbs tvx vb) renv)` y `tx` representa el mismo tipo que `tvx`, el patrón de código `pb` se evalúa para concordar con la expresión `vb` citada con el contexto `(REnvCons tvx renv)`, que es el contexto `renv` del código pasado, extendido con el tipo `tvx` ligado en la abstracción. Entonces, la evaluación en Ωmega de la concordancia del patrón `[|`$\lambda^{\texttt{int}}$`. $(•)`$^{\texttt{int}}$`|]` con el código `[|`$\lambda^{\texttt{int}}$`. #0`$^{\texttt{int}}$`|]{}` partiendo de un ambiente vacío es:

```
evalCPat   (PCAbs Int (PCPVar Int))
           (Q (EAbs Int (EVar VZ Int)) REnvNil)
           EnvNil
```

Como el código pasado tiene la forma `(Q (EAbs tvx vb) renv)` y se cumple que los tipos `tvx` y `tx` unifican (`eqType Int Int`), el resultado es el de evaluar la concordancia del sub-patrón con el cuerpo de la abstracción `vb`, con el cual se debe construir un valor de tipo código para que sea un parámetro correcto de la función `evalCPat`. Por eso se evalúa la concordancia del sub-patrón con el resultado de evaluar en el contexto actual a la sub-expresión `(EVar VZ Int)` encerrada entre paréntesis (anotando el contexto actual extendido con el tipo `Int`).

```
evalCPat   (PCPVar Int)
           (eval (EBr (EVar VZ Int) (REnvCons Int REnvNil)) EnvNil)
           EnvNil
```

Como la evaluación de los paréntesis resulta en el valor `(Q (EVar VZ Int) (REnvCons Int REnvNil))`, que tiene la forma `(Q v renv)`, y los tipos unifican, entonces el resultado de la concordancia es:

```
(Just (EnvCons (Q (EVar VZ Int) (REnvCons Int REnvNil)) EnvNil)).
```

En los casos de `PCVar` y `PCSubst` se utilizan las funciones `eqVar` y `eqSust` respectivamente, que devuelven un booleano que indica si dos variables, en el primer caso, o dos sustituciones, en el segundo, son sintácticamente iguales. Para la alternación (`PCAlt`) se utiliza también una función que verifica la igualdad, en este caso, entre dos patrones. Pero en esta función se debe realizar una unificación de tipos, de manera de poder utilizar el patrón para encontrar el contexto en el cual se debe citar la primera sub-rama de la alternación para poder analizarla. Por lo tanto, la función `eqPat` tiene el siguiente tipo:

```
eqPat   :: Pat t1 e1 -> Pat t2 e2 -> Rep t1 -> Rep t2
        -> Maybe(Equal (Pat t1 e1) (Pat t2 e2))
```

tomando dos patrones y la representación de sus tipos, y devolviendo, si es posible, un testigo de que los patrones son iguales, y por lo tanto, se cumple que `t1=t2` y `e1=e2`.

## 5.4. Corrección (*Soundness*)

Como se vió en la subsección 5.1.3, el sistema de tipos de Ωmega asegura automáticamente la propiedad de *subject reduction*, dado que, según el tipo de `eval0 (forall p. Exp`

`p n t) -> Env n -> t`, la evaluación de cualquier expresión en el nivel 0 que satisface el juicio de tipado $P; \Gamma \vdash e : \tau$ produce, si termina, un valor de tipo $\tau$.

También se puede demostrar fácilmente el progreso, mediante una inducción que pruebe que la función `eval0` está definida para todos los términos bien-tipados. Demostrándose entonces que si un término no es un valor, existe una regla de reducción que es aplicable al mismo.

Entonces, la implementación en Ωmega de las semánticas del lenguaje permite demostrar la corrección de su sistema de tipos. Por lo tanto, se puede argumentar que en este lenguaje un término bien tipado evalúa a un valor del mismo tipo u ocasiona una secuencia infinita de reducciones.

# Capítulo 6

# Conclusiones y Trabajo Futuro

*Y daré fin a mis coplas*
*con aire de relación;*
*nunca falta un preguntón*
*más curioso que mujer,*
*y tal vez quiera saber*
*cómo fue la conclusión.*

Fragmento de "El Gaucho Martín Fierro".

## 6.1. Conclusiones

En este trabajo se presentó un lenguaje funcional multi-etapas homogéneo con soporte para análisis intensional y se definieron sus semánticas estática y operacional. Con el objetivo de poder realizar la inspección de código con una interfaz de alto nivel el lenguaje incorpora un mecanismo de concordancia de patrones sobre las expresiones citadas, que se utiliza para ligar las variables de una primitiva de alternación. Se define una categoría de patrones de código, que en su mayoría coinciden con la estructura de las expresiones del lenguaje y consisten en descomponer la expresión y aplicar otros patrones de código a sus sub-expresiones. También existen otros patrones que concuerdan con cualquier código o ligan las variables de la alternación con el código concordado.

La inclusión de análisis intensional debilita la noción de equivalencia en niveles superiores, por lo que se debe limitar la $\beta$ reducción al nivel 0, lo que implica que muchas optimizaciones interesantes no se puedan realizar en niveles superiores. Por otro lado, como se vió en algunos ejemplos presentados, la capacidad de acceder a la estructura del código permite la implementación de éstas y otras optimizaciones. Otro problema que trae la limitación de la $\beta$ reducción (observado por Taha [Tah99]) es que la persistencia entre etapas lleva a una pérdida de confluencia. Por esto no se agrega esta última característica al lenguaje, perdiendo el programador la ventaja de poder reutilizar en futuras etapas todas las primitivas y ligaduras definidas en la etapa actual.

Al diseñarse el sistema de tipos de este lenguaje se buscó llegar a un acuerdo entre la seguridad de un sistema de tipos estático y la flexibilidad del tipado dinámico. El chequeo de tipos de las expresiones en nivel 0 se realiza estáticamente, mientras que el chequeo de las expresiones citadas (niveles superiores) se realiza en tiempo de ejecución. El tipo $\texttt{cod}^{\Gamma^f}$ de las expresiones citadas no refleja el tipo de la expresión, pero sí el de sus variables libres. Al no reflejarse el tipo de la expresión, el mecanismo de inspección de código se vé potenciado, permitiéndose la construcción de funciones que descomponen o

atraviesan dicho código. El hecho de que se refleje el tipo de las variables libres restringe un poco este tipo de funciones, pero, por otro lado, permite que se asegure estáticamente si una expresión citada puede ser evaluada utilizando `run`, dado que un código puede ser ejecutado sólo si es cerrado. Una contra de este tipo es que se puede volver muy grande a medida que aumenta la cantidad de variables libres del código.

Evaluar una expresión de código requiere un chequeo de tipos dinámico, debido a que durante la ejecución puede generarse código mal tipado. Debido a esto se incluye una expresión de excepción en el operador `run`, de manera que si un código no está bien tipado, o su tipo no coincide con el de la expresión de excepción, éste no se evalúa, evitándose así la presencia de errores de tipado en tiempo de ejecución. Además, es posible detectar mediante el mecanismo de concordancia de patrones si un código está mal tipado, y por lo tanto concuerda con el patrón `fail`.

Para contar con una forma sencilla de capturar las variables libres de un código se incluye un operador de sustitución explícita sobre las expresiones citadas. Una sustitución explícita $\Theta$, con juicio $\Gamma \vdash \Theta \Rightarrow \Gamma'$, relaciona un contexto de tipos $\Gamma$ con un contexto de tipos "resultante" $\Gamma'$. Entonces, aplicar $\Theta$ sobre una expresión citada de tipo $\texttt{cod}^{\Gamma}$ resulta en una nueva expresión citada de tipo $\texttt{cod}^{\Gamma'}$. De esta forma se puede compatibilizar el tipo de dos expresiones citadas o cerrar una expresión, de manera de hacerla ejecutable.

El lenguaje fue implementado siguiendo el procedimiento propuesto por Sheard [She05b] de utilizar el metalenguaje Ωmega como una herramienta de asistencia automatizada. En este sentido se definieron las semánticas estática y dinámica en términos de Ωmega. Se representó la semántica estática como un GADT de Ωmega, donde cada juicio de tipado se codifica con un tipo múltiplemente indexado. Por ejemplo, el juicio de las expresiones $P; \Gamma \vdash e : \tau$ se codifica con el tipo `Exp p n t`. Cada constructor de este tipo tiene la estructura de un juicio formal del tipado del lenguaje. Los tipos básicos del lenguaje (`int`, `bool`, productos y funciones) se representan en términos de los tipos respectivos de Ωmega. El uso de los índices de de Bruijn para codificar las ligaduras de variables se debe, además de para evitar problemas asociados a $\alpha$-equivalencia, a la necesidad de codificar los contextos como índices en los tipos que representan juicios. Utilizando índices de de Bruijn se vé a un contexto como una lista de tipos, donde cada posición indica a una variable, lo cual se puede representar con el kind `Row` definido en Ωmega. De esta forma, por ejemplo, el índice `n` en el tipo `Exp p n t` mantiene la restricción del contexto en el cual tipa una expresión.

El tipo de código $\texttt{cod}^{\Gamma^f}$ se codifica con el tipo `Cod n` (el índice `n` representa al contexto $\Gamma^f$), donde se tiene un constructor `Q` para código bien formado, y otro `F` para código fallido. El requerimiento del constructor `Q` de que la pila de contextos del "pasado" de la expresión citada sea universalmente cuantificable refleja que un código bien formado es una expresión en el nivel 0, dado que no depende de su pasado.

Como el sistema de tipos del meta-lenguaje chequea y mantiene las propiedades codificadas, el sistema de tipos del lenguaje objeto se pudo construir y verificar interactivamente.

Se definió la semántica dinámica como una función de evaluación `eval`, con tipo `(forall p. Exp p n t) -> Env n -> t`, que a partir de un valor con tipo `Exp p n t`, que representa a un término bien tipado, y un valor con tipo `Env n`, que representa a un ambiente con forma `n`, retorna un valor con tipo `t`. La semántica natural fue concebida a medida que se construía esta función. El chequeo dinámico de tipos se implementó con una función de unificación `eqType`, con tipo `Rep a -> Rep b -> Maybe (Equal a b)`, que toma dos representaciones de tipo, verifica que tengan una igualdad estructural, y si es posible retorna un valor `Equal a b`, que es un testigo dinámico de que se cumple esa propiedad estática.

En base al sistema de tipos de Ωmega, se puede argumentar la corrección del sistema de tipos del lenguaje con respecto a su semántica operacional. Por lo tanto, se puede argumentar que todo término bien-tipado evalúa a un valor del mismo tipo u ocasiona una secuencia infinita de reducciones. En otras palabras, nunca se tranca.

Como se puede ver en el Capítulo 3, muchas de las características del lenguaje propuesto ya se encuentran en otros lenguajes. Pero a nuestro entender uno de los principales aportes de este trabajo es el lograr su combinación de una forma compatible. Otro aporte interesante se encuentra en el uso de Ωmega para su codificación. Debido a que el lenguaje objeto definido tiene una cierta complejidad relativa, mayor a la de los lenguajes para los que se había aplicado anteriormente, se utilizaron muchas de las técnicas sugeridas en los artículos sobre Ωmega [Pas04, SP04, She05b, She05a], mostrando sus fortalezas y debilidades.

## 6.2. Trabajo Futuro

En su estado actual el lenguaje propuesto es bastante impráctico debido a todas sus anotaciones de tipo. Sin embargo, al igual que como hacen Grundy *et al.* [GMO03] y Shields *et al.* [SSP98], muchas de las mismas podrían ser generadas automáticamente mediante un algoritmo de anotación de tipos del estilo del algoritmo de inferencia de tipos de Hindley-Milner.

Una extensión interesante al lenguaje sería el agregado de polimorfismo. Inclusive se debería estudiar la forma de agregar un cierto polimorfismo en los ambientes de las expresiones citadas. Esto es, por ejemplo, tener la capacidad de definir abstracciones con la forma $\lambda^{\texttt{cod}^{\texttt{a,int}}}$, que toma un código con cualquier contexto cuya primera variable sea entera.

La codificación en Ωmega fue construída para representar las semánticas definidas en el Capítulo 4, pero no existe una demostración formal de que la codificación y las semánticas sean equivalentes. Se debería demostrar que se cumple $e \overset{0}{\hookrightarrow} v$ si y sólo si `(eval0 e)` evalúa a la representación en Ωmega del valor $v$.

También resultaría de interés poder formalizar la relación entre la sintaxis con variables nombradas y la que utiliza índices de de Bruijn, de manera de poder tener un lenguaje más fácil de utilizar, pero con sus semánticas formalizadas.

Este trabajo es un primer paso en la búsqueda de proveer reflexión completa en lenguajes funcionales tipados. Otro paso a seguir en esta búsqueda sería el de agregarle al lenguaje la capacidad de reificar, manipular y absorber ambientes y continuaciones.

# Bibliografía


[Ala04] Lauri E. Alanko. Types and reflection. Tesis de Maestría, University of Helsinki, 2004.

[AM91] Andrew W. Appel y David B. MacQueen. Standard ML of New Jersey. En J. Maluszynski y M. Wirsing, editores, *Proceedings of theThirdInternational Symposium on Programming Language Implementation and Logic Programming*, número 528, páginas 1–13. Springer Verlag, 1991.

[Bar92] Hendrik Barendregt. *Handbook of Logic in Computer Science*, volumen 2, capítulo Lambda Calculi with Types, páginas 117–309. Oxford University Press, 1992.

[Bar06] Eli Barzilay. *Implementing Reflection in Nuprl.* Tesis de Doctorado, Cornell University, 2006.

[Baw99] Alan Bawden. Quasiquotation in Lisp. En *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, páginas 4–12, 1999.

[BBLR96] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, y Jocelyne Rouyer-Degli. $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, 1996.

[BGW93] Daniel G. Bobrow, Richard P. Gabriel, y Jon L. White. Clos in context — the shape of the design. En A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, páginas 29–61. MIT Press, 1993.

[BS02] Arthur I. Baars y S. Doaitse Swierstra. Typing dynamic typing. En *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, páginas 157–166. ACM Press, 2002.

[Caz98] Walter Cazzola. Evaluation of object-oriented reflective models. En *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, páginas 386–387, 1998.

[CFW85] William D. Clinger, Daniel P. Friedman, y Mitchell Wand. A scheme for a higher-level semantic algebra. En *Algebraic Methods in Semantics*, páginas 237–250. Cambridge University Press, 1985.

[CH02] James Cheney y Ralf Hinze. A lightweight implementation of generics and dynamics. En *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, páginas 90–104. ACM Press, 2002.

[CH03] James Cheney y Ralf Hinze. First-class phantom types. Reporte técnico, Cornell University, 2003.

[CMT04] Cristiano Calcagno, Eugenio Moggi, y Walid Taha. ML-like inference for classifiers. En David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, volumen 2986 de *Lecture Notes in Computer Science*, páginas 79–93. Springer, 2004.

[CW85] Luca Cardelli y Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. En *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, páginas 184–195, Washington, DC, USA, 1996. IEEE Computer Society.

[dB72] Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[DHM91] Bruce F. Duba, Robert Harper, y David MacQueen. Typing first-class continuations in ML. En *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, páginas 163–173, Orlando, Florida, 1991.

[DM95] François-N. Demers y Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. En *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, páginas 29–38, agosto 1995.

[DP96] Rowan Davies y Frank Pfenning. A modal analysis of staged computation. En *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 258–270, New York, NY, USA, 1996. ACM Press.

[dRS84] Jim des Rivières y Brian Cantwell Smith. The implementation of procedurally reflective languages. En *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, páginas 331–347. ACM Press, 1984.

[Fee01] Marc Feeley. A better API for first-class continuations. En *2nd Workshop on Scheme and Functional Programming*, septiembre 2001.

[FW84] Daniel P. Friedman y Mitchell Wand. Reification: Reflection without metaphysics. En *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, páginas 348–355, agosto 1984.

[GKH+84] Richard D. Greenblatt, Thomas F. Knight, Jack Holloway, David A. Moon, y Daniel L. Weinreb. The LISP Machine. En D. R. Barstow, H. E. Shrobe, y E. Sandewall, editores, *Interactive Programming Environments*, páginas 326–352. McGraw-Hill, New York, 1984.

[GMO03] Jim Grundy, Tom Melham, y John O'Leary. A reflective functional language for hardware design and theorem proving. Reporte Técnico PRG-RR-03-16, Programming Research Group, Oxford University Computing Laboratory, octubre 2003.

[HC96] George E. Hughes y Max J. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996.

[Hin03] Ralf Hinze. Fun with phantom types. En Jeremy Gibbons y Oege de Moor, editores, *The Fun of Programming*, páginas 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

[HS93] James Hook y Tim Sheard. A semantics of compile-time reflection. Reporte Técnico 93-019, Dept. of Computer Science and Engineering, Oregon Graduate Institute, Portland, Oregon, 1993.

[Jag92] Suresh Jagannathan. Reflective building blocks for modular systems. En *International Workshop on Reflection and Meta-Level Architectures*, noviembre 1992.

[Jag94] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, mayo 1994.

[KdRB91] Gregor Kiczales, Jim des Rivières, y Daniel G. Bobrow. *The art of metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[KM04] Sava Krstic y John Matthews. Semantics of the $reFL^{ect}$ language. En *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, páginas 32–42, 2004.

[Lan65] Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I/II. *Communications of the ACM*, 8(2/3):89–101:158–165, 1965.

[LO93] Konstantin Läufer y Martin Odersky. Self-interpretation and reflection in a statically typed language. En *Proc. OOPSLA Workshop on Reflection and Metalevel Architectures*. ACM Press, octubre 1993.

[Mae87a] Pattie Maes. *Computational reflection*. Tesis de Doctorado, Artificial intelligence laboratory, Vrije Universiteit, 1987.

[Mae87b] Pattie Maes. Concepts and experiments in computational reflection. En *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, páginas 147–155. ACM Press, 1987.

[McC62] John McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[Meu98] Wolfgang De Meuter. Agora: The story of the simplest mop in the world - or - the scheme of object-orientation. En Noble, Moore, y Taivalsaari, editores, *Prototype-based Programming*. Springer-Verlag, 1998.

[MF96] Anurag Mendhekar y Daniel P. Friedman. An exploration of relationships between reflective theories. En Gregor Kiczales, editor, *Proceedings of the Reflection'96 Conference*, páginas 245–262, abril 1996.

[MJD96] Jacques Malenfant, Marco Jacques, y François-N. Demers. A tutorial on behavioral reflection and its implementation. En Gregor Kiczales, editor, *Proceedings of the Reflection'96 Conference*, páginas 1–20, abril 1996.

[MTBS99] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, y Tim Sheard. An idealized MetaML: Simpler, and more expressive. En *European Symposium on Programming*, páginas 193–207, 1999.

[Nan02a] Aleksandar Nanevski. Meta-programming with names and necessity. En *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional programming*, páginas 206–217, New York, NY, USA, 2002. ACM Press.

[Nan02b] Aleksandar Nanevski. Meta-programming with names and necessity. Reporte técnico, School of Computer Science, Carnegie Mellon University, 2002.

[Nan04] Aleksandar Nanevski. *Functional Programming with Names and Necessity.* Tesis de Doctorado, School of Computer Science, Carnegie Mellon University, junio 2004.

[NP05] Aleksandar Nanevski y Frank Pfenning. Staged computation with names and necessity. *Journal of Functional Programming*, 15(6):893–939, 2005.

[Pas04] Emir Pasalic. *The Role of Type Equality in Meta-Programming.* Tesis de Doctorado, Oregon Health and Sciences University, The OGI School of Science and Engineering, 2004.

[PL04] Emir Pasalic y Nathan Linger. Meta-programming with typed object-language representations. En *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, páginas 136–167. Springer, 2004.

[QD96] Christian Queinnec y David DeRoure. Sharing code through first-class environments. En *Proceedings of ICFP'96 International Conference on Functional Programming*, páginas 251–261, mayo 1996.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. En *ACM '72: Proceedings of the ACM annual conference*, páginas 717–740. ACM Press, 1972.

[Riv96] Fred Rivard. Smalltalk: a reflective language. En *Reflection '96*, páginas 21–38, S. Francisco, USA, abril 1996.

[Riv97] Fred Rivard. *Évolution du comportement dans les langages à classes réflexifs dynamiquement typés.* Tesis de Doctorado, École des Mines de Nantes, 1997.

[Rod06] Leonardo Rodríguez. The reflex sandbox: An experimentation environment for an aspect-oriented kernel. Tesis de Maestría, Instituto de Computación, Facultad de Ingeniería de la Universidad de la República, marzo 2006.

[SBM00] Tim Sheard, Zine-El-Abidine Benaissa, y Matthieu Martel. *Introduction to Multistage Programming Using MetaML.* Pacific Software Research Center, Oregon Graduate Institute, 2 edición, 2000.

[Sch86] David A. Schmidt. *Denotational Semantics. A Methodology for Language Development.* Allyn and Bacon, Inc., Boston Mass., 1986.

[SF96] Jonathan M. Sobel y Daniel P. Friedman. An introduction to reflection-oriented programming. En Gregor Kiczales, editor, *Proceedings of the Reflection'96 Conference*, abril 1996.

[SG93] Guy L. Steele, Jr. y Richard P. Gabriel. The evolution of Lisp. *ACM SIGPLAN Notices*, 28(3):231–270, 1993.

[SH94] Tim Sheard y James Hook. Type safe meta-programming. Oregon Graduate Institute, 1994.

[She91] Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, 1991.

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. En *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, páginas 2–44, London, UK, 2001. Springer-Verlag.

[She04] Tim Sheard. Languages of the future. En *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, páginas 116–119. ACM Press, 2004.

[She05a] Tim Sheard. Playing with type systems. Presented at GPCE 05 MetaOCaml Workshop, 2005.

[She05b] Tim Sheard. Putting Curry-Howard to work. En *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, páginas 74–85. ACM Press, 2005.

[Smi82] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. Tesis de Doctorado, Massachusetts Institute of Technology, 1982.

[Smi84] Brian Cantwell Smith. Reflection and semantics in Lisp. En *Principles of Programming Languages (POPL84)*, páginas 23–35. ACM Press, 1984.

[SMKC93] David Stemple, Ronald Morrison, Graham N. C. Kirby, y Richard C. H. Connor. Integrating reflection, strong typing and static checking. En *16th Australian Computer Science Conference (ACSC'93)*, páginas 83–92, 1993.

[SP02] Tim Sheard y Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[SP04] Tim Sheard y Emir Pasalic. Meta-programming with built-in type equality. En *Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, páginas 106–124, jul 2004.

[SS75] Gerald Jay Sussman y Guy Lewis Steele. Scheme: An interpreter for extended lambda calculus. Reporte Técnico AI Lab Memo AIM-349, MIT AI Lab, diciembre 1975.

[SSB01] Masahiko Sato, Takafumi Sakurai, y Rod Burstall. Explicit environments. *Fundamenta Informaticae*, 45(1,2):79–115, 2001.

[SSP98] Mark Shields, Tim Sheard, y Simon Peyton Jones. Dynamic typing as staged type inference. En *Principles of Programming Languages (POPL 98)*, páginas 289–302, enero 1998.

[Sta02] Richard M. Stallman. *GNU Emacs Manual*. GNU Press, Boston, MA, USA, 15 edición, 2002.

[Ste94] Patrick Steyaert. *Open Design of Object-Oriented Languages A Foundation for Specializable Reflective Language Frameworks*. Tesis de Doctorado, Vrije Universiteit Brussels, 1994.

[Tah99] Walid Taha. *Multi-stage Programming: Its Theory and Applications*. Tesis de Doctorado, Oregon Graduate Institute of Science and Technology, noviembre 1999.

[Tah03] Walid Taha. A gentle introduction to multi-stage programming. En *Domain-Specific Program Generation*, páginas 30–50, 2003.

[TBS98] Walid Taha, Zine-El-Abidine Benaissa, y T. Sheard. Multi-stage programming: Axiomatization and type safety. volumen 1443, páginas 918–929, 1998.

[TN03] Walid Taha y Michael Florentin Nielsen. Environment classifiers. En *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 26–37, New York, NY, USA, 2003. ACM Press.

[TS00] Walid Taha y Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theory of Computer Science*, 248(1-2):211–242, 2000.

[VP06] Marcos Viera y Alberto Pardo. A multi-stage language with intensional analysis. En *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, páginas 11–20, New York, NY, USA, 2006. ACM Press.

[WF86] Mitchell Wand y Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. En *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, páginas 298–307. ACM Press, 1986.

[Yan99] Zhe Yang. A survey on multi-stage programming languages. Trabajo para el Doctorado, enero 1999.

# Índice de figuras

# Índice de cuadros

# Apéndice A

# Juicios y Evaluaciones del Lenguaje

## A.1. Juicios de Tipado

| | |
|---|---|
| $P; \Gamma \vdash e : \tau$ | La expresión $e$ tiene tipo $\tau$ en un contexto local $\Gamma$ bajo la pila $P$. |
| $\Gamma \vdash x : \tau$ | La variable $x$ tiene tipo $\tau$ en un contexto local $\Gamma$. |
| $\Gamma \vdash \Theta \Rightarrow \Gamma'$ | La sustitución $\Theta$ sobre una expresión tipada en un contexto local $\Gamma$ resulta en una expresión tipada en un contexto local $\Gamma'$. |
| $\vdash p : \tau \Rightarrow \Gamma$ | El patrón $p$, que puede concordar con un valor de tipo $\tau$, tiene las variables libres contenidas en $\Gamma$. |
| $\Gamma^f \Vdash pc \Rightarrow \Gamma'$ | El patrón $pc$, que podría concordar con un código de tipo $\texttt{cod}^{\Gamma^f}$, tiene las variables libres contenidas en $\Gamma'$. |

## A.2. Notaciones de Evaluación

| | |
|---|---|
| $e \overset{n}{\hookrightarrow} v$ | La expresión $e$ evalúa al valor $v$ en el nivel $n$. |
| $\Gamma_1 \vdash \Theta : v_1 \Longrightarrow v_2$ | La sustitución $\Theta$ sobre el valor $v_1$ en el contexto $\Gamma_1$ resulta en el valor $v_2$. |
| $\Theta : p \triangleright v$ | El patrón $p$ concuerda con el valor $v$ vía la sustitución $\Theta$. |
| $p \not\triangleright v$ | El patrón $p$ no concuerda con el valor $v$. |
| $\Theta : pc \trianglerighteq v$ | El patrón de código $pc$ concuerda con el código $v$ vía la sustitución $\Theta$. |
| $pc \not\trianglerighteq v$ | El patrón de código $pc$ no concuerda con el código $v$. |

# Apéndice B

# Implementación

## B.1. Semántica Estática

```
import "LangPrelude.prg" (Monad,maybeM,fst,snd)

data Env :: Row *0 ~> *0 where
    EnvNil  :: Env RNil
    EnvCons :: t -> Env r
            -> Env (RCons t r)

data Rep:: *0 ~> *0 where
    Int :: Rep Int
    Bool    :: Rep Bool
    Arr :: Rep a -> Rep b -> Rep(a -> b)
    Prod    :: Rep a -> Rep b -> Rep (a,b)
    Cod :: RepEnv n -> (Rep (Cod n))

data RepEnv:: Row *0 ~> *0 where
    REnvNil     :: RepEnv RNil
    REnvCons    :: Rep t -> RepEnv r
                -> RepEnv (RCons t r)

data Cod :: Row *0 ~> *0 where
    Q   :: (forall p. Exp p n t) -> RepEnv n
        -> Cod n
    F   :: RepEnv n
        -> Cod n
```

### B.1.1. Expresiones

```
data Exp :: *0 ~> Row *0 ~> *0 ~> *0 where

    ELBool  :: Bool -> Exp p n Bool
    EBNot   :: Exp p n Bool
            -> Exp p n Bool
    EBOr    :: Exp p n Bool -> Exp p n Bool
```

```
          -> Exp p n Bool
    EBAnd   :: Exp p n Bool -> Exp p n Bool
          -> Exp p n Bool

    ELInt   :: Int -> Exp p n Int
    EINeg   :: Exp p n Int
          -> Exp p n Int
    EIPlus  :: Exp p n Int -> Exp p n Int
          -> Exp p n Int
    EIMult  :: Exp p n Int -> Exp p n Int
          -> Exp p n Int

    EILess  :: Exp p n Int -> Exp p n Int
          -> Exp p n Bool
    EIEq    :: Exp p n Int -> Exp p n Int
          -> Exp p n Bool

    EPair   :: Exp p n t -> Exp p n s
          -> Exp p n (t,s)
    EPFst   :: Exp p n (t,s)
          -> Exp p n t
    EPSnd   :: Exp p n (t,s)
          -> Exp p n s

    EAbs    :: Rep s -> Exp p (RCons s n) t
          -> Exp p n (s->t)
    EFix    :: Exp p (RCons t n) t
          -> Exp p n t
    EApp    :: Exp p n (s->t) -> Exp p n s
          -> Exp p n t
    EVar    :: Var n t -> Rep t
          -> Exp p n t

    EAlt    :: Pat s c -> Exp p {eapp c n} t -> Exp p n (s->t)
          -> Exp p n (s->t)
    ECond   :: Exp p n Bool -> Exp p n t -> Exp p n t
          -> Exp p n t

    EBr     :: Exp (p,Env n) c t -> RepEnv c
          -> Exp p n (Cod c)
    ERun    :: Exp p n (Cod RNil) -> Exp p n t
          -> Exp p n t
    EEsc    :: Exp p b (Cod n) -> Rep t
          -> Exp (p, Env b) n t
    ESubst  :: Exp p n (Cod f) -> Subst f fc
          -> Exp p n (Cod fc)


eapp :: Row *0 ~> Row *0 ~> Row *0
```

```
{eapp RNil ys} = ys
{eapp (RCons x xs) ys} = RCons x {eapp xs ys}
{eapp {eapp xs ys} zs} = {eapp xs {eapp ys zs}}


eapp3 :: Row *0 ~> Row *0 ~> Row *0 ~> Row *0
{eapp3 xs ys zs} = {eapp xs {eapp ys zs}}


data Var :: Row *0 ~> *0 ~> *0 where
    VZ      :: Var (RCons t env) t
    VS      :: Var env t
            -> Var (RCons s env) t
```

### B.1.2. Patrones

```
data Pat :: *0 ~> Row *0 ~> *0 where

    PLInt   :: Int
            -> Pat Int RNil
    PLBool  :: Bool
            -> Pat Bool RNil

    PPair   :: Pat t1 c1 -> Pat t2 c2
            -> Pat (t1,t2) {eapp c2 c1}

    PVar    :: Rep t
            -> Pat t (RCons t RNil)
    PAny    :: Pat t RNil

    PCod    :: PatCod f c
            -> Pat (Cod f) c


data PatCod :: Row *0 ~> Row *0 ~> *0 where

    PCPVar  :: Rep t
            -> PatCod f (RCons (Cod f) RNil)
    PCPAny  :: PatCod f RNil
    PCPFail :: PatCod f RNil

    PCLit   :: Rep t
            -> PatCod f (RCons (Cod f) RNil)
    PCVar   :: Var vn t -> Rep t
            -> PatCod f RNil

    PCLInt  :: Int
            -> PatCod f RNil
```

```
PCLBool :: Bool
        -> PatCod f RNil

PCINeg  :: PatCod f c
        -> PatCod f c
PCIPlus :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}
PCIMult :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}
PCILess :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}
PCIEq   :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}
PCBNot  :: PatCod f c
        -> PatCod f c
PCBOr   :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}
PCBAnd  :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}

PCPair  :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}
PCPFst  :: PatCod f c
        -> PatCod f c
PCPSnd  :: PatCod f c
        -> PatCod f c

PCAbs   :: Rep s -> PatCod (RCons s f) c
        -> PatCod f c
PCApp   :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}

PCFix   :: Rep s -> PatCod (RCons s f) c
        -> PatCod f c
PCCond  :: PatCod f c1 -> PatCod f c2 -> PatCod f c3
        -> PatCod f {eapp3 c3 c2 c1}

PCBr    :: RepEnv fp
        -> PatCod f (RCons (Cod f) RNil)
PCRun   :: PatCod f c1 -> PatCod f c2
        -> PatCod f {eapp c2 c1}
PCSubst :: PatCod fc c -> Subst f fc
        -> PatCod fc c

PCAlt   :: Rep s -> Pat s fp -> PatCod {eapp fp f} c1
        -> PatCod f c2
        -> PatCod f {eapp c2 c1}
```

### B.1.3. Sustituciones

```
data Subst :: Row *0 ~> Row *0 ~> *0 where

    SSft    :: Rep t
            -> Subst n (RCons t n)
    SLft    :: Subst n c
            -> Subst (RCons t n) (RCons t c)
    SSlsh   :: (forall p. Exp p n t)
            -> Subst (RCons t n) n
```

## B.2. Semántica Dinámica

### B.2.1. Evaluación de Expresiones

```
eval0 :: (forall p. Exp p RNil t) -> t
eval0 e = eval e EnvNil


eval :: Exp p n t -> Env n -> t

eval (ELInt i) env          = i
eval (EINeg e) env          = - (eval e env)
eval (EIPlus e1 e2) env     = (eval e1 env) + (eval e2 env)
eval (EIMult e1 e2) env     = (eval e1 env) * (eval e2 env)

eval (ELBool b) env         = b
eval (EBNot e) env          = not (eval e env)
eval (EBOr e1 e2) env       = (eval e1 env) || (eval e2 env)
eval (EBAnd e1 e2) env      = (eval e1 env) && (eval e2 env)

eval (EILess e1 e2) env     = (eval e1 env) < (eval e2 env)
eval (EIEq e1 e2) env       = (eval e1 env) == (eval e2 env)

eval (EPair e1 e2) env      = (eval e1 env,eval e2 env)
eval (EPFst e) env          = fst (eval e env)
eval (EPSnd e) env          = snd (eval e env)

eval (EAbs t e) env         = \ v -> eval e (EnvCons v env)
eval (EApp f x) env         = (eval f env) (eval x env)
eval (EVar v t) env         = evalVar v t env
eval (EAlt p e1 e2) env     = \ v -> case (evalPat p v env) of
                                        (Just env2) -> eval e1 env2
                                        Nothing -> (eval e2 env) v
eval (ECond ec e1 e2) env   = if (eval ec env)
                                then (eval e1 env)
                                else (eval e2 env)
```

```
eval (EFix e) env          = lazy ( (\v -> (eval e (EnvCons v env)))
                                    (eval (EFix e) env))


eval (EBr e renv) env       = case (bd (CountBrZ env) e) of
                                (x,True) -> Q x renv
                                (x,False) -> F renv

eval (ERun e1 e2) env       = case (eval e1 env) of
                                (Q e REnvNil) ->
                                    case eqType (getType e) (getType e2) of
                                        Just Eq -> eval e EnvNil
                                        Nothing -> eval e2 env
                                _ -> eval e2 env

eval (ESubst e s) env       = case (eval e env) of
                                (Q eb rb) ->
                                    case (evalSub s eb rb) of
                                        (en,rn) -> (Q en rn)
                                (F rb) -> F (evalSubR s rb)
```

### B.2.2. Evaluación de Variables

```
evalVar :: Var n t -> Rep t -> Env n -> t

evalVar VZ t (EnvCons v vs)     = v
evalVar (VS v) t (EnvCons x xs) = evalVar v t xs
```

### B.2.3. Construcción de Código

```
bd :: CountBr a z -> Exp a n t -> (Exp z n t,Bool)

bd env (ELInt i)        =       (ELInt i,True)
bd env (EINeg e)        =       let
                                        (x,b) = (bd env e)
                                in
                                        (EINeg x , b)
bd env (EIPlus e1 e2)   =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EIPlus x1 x2, b1 && b2)
bd env (EIMult e1 e2)   =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
```

```
                                        (EIMult x1 x2, b1 && b2)

bd env (ELBool b)       =       (ELBool b,True)
bd env (EBNot e)        =       let
                                        (x,b) = (bd env e)
                                in
                                        (EBNot x, b)
bd env (EBOr e1 e2)     =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EBOr x1 x2, b1 && b2)
bd env (EBAnd e1 e2)    =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EBAnd x1 x2, b1 && b2)
bd env (EILess e1 e2)   =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EILess x1 x2, b1 && b2)
bd env (EIEq e1 e2)     =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EIEq x1 x2, b1 && b2)
bd env (EPair e1 e2)    =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EPair x1 x2, b1 && b2)
bd env (EPFst e)        =       let
                                        (x,b) = (bd env e)
                                in
                                        (EPFst x, b)
bd env (EPSnd e)        =       let
                                        (x,b) = (bd env e)
                                in
                                        (EPSnd x, b)
bd env (EAbs t e)       =       let
                                        (x,b) = (bd env e)
                                in
                                        (EAbs t x, b)
bd env (EApp e1 e2)     =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EApp x1 x2, b1 && b2)
```

```
bd env (EVar v t)       =       (EVar v t,True)
bd env (EAlt p e1 e2)   =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EAlt p x1 x2, b1 && b2)
bd env (ECond c e1 e2)  =       let
                                        (xc,bc) = (bd env c)
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (ECond xc x1 x2, (bc && b1) && b2)
bd env (EFix e)         =       let
                                        (x,b) = (bd env e)
                                in
                                        (EFix x, b)


bd env (EBr e renv)     =       let
                                        (x,b) = (bd (CountBrS env) e)
                                in
                                        (EBr x renv, b)
bd env (ESubst e s)     =       let
                                        (x,b) = (bd env e)
                                in
                                        (ESubst x s, b)
bd env (ERun e1 e2)     =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (ERun x1 x2, b1 && b2)
bd (CountBrZ env) (EEsc e t)
                        = case (eval e env) of
                                (Q x renv) ->
                                        case eqType (getType x) t of
                                                Just Eq -> (x,True)
                                                Nothing -> (getAny t,False)
                                _ -> (getAny t,False)

bd (CountBrS r) (EEsc e t)
                        =       let
                                        (x,b) = (bd r e)
                                in
                                        (EEsc x t, b)


data CountBr :: *0 ~> *0 ~> *0 where
    CountBrZ :: a -> CountBr (b,a) c
    CountBrS :: CountBr a b -> CountBr (a,c) (b,c)
```

```
bd :: CountBr a z -> Exp a n t -> (Exp z n t,Bool)


bd env (ELInt i)        =       (ELInt i,True)
bd env (EINeg e)        =       let
                                        (x,b) = (bd env e)
                                in
                                        (EINeg x , b)
bd env (EIPlus e1 e2)   =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EIPlus x1 x2, b1 && b2)
bd env (EIMult e1 e2)   =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EIMult x1 x2, b1 && b2)

bd env (ELBool b)       =       (ELBool b,True)
bd env (EBNot e)        =       let
                                        (x,b) = (bd env e)
                                in
                                        (EBNot x, b)
bd env (EBOr e1 e2)     =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EBOr x1 x2, b1 && b2)
bd env (EBAnd e1 e2)    =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EBAnd x1 x2, b1 && b2)
bd env (EILess e1 e2)   =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EILess x1 x2, b1 && b2)
bd env (EIEq e1 e2)     =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EIEq x1 x2, b1 && b2)
bd env (EPair e1 e2)    =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
```

```
                                in
                                        (EPair x1 x2, b1 && b2)
bd env (EPFst e)        =       let
                                        (x,b) = (bd env e)
                                in
                                        (EPFst x, b)
bd env (EPSnd e)        =       let
                                        (x,b) = (bd env e)
                                in
                                        (EPSnd x, b)
bd env (EAbs t e)       =       let
                                        (x,b) = (bd env e)
                                in
                                        (EAbs t x, b)
bd env (EApp e1 e2)     =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EApp x1 x2, b1 && b2)
bd env (EVar v t)       =       (EVar v t,True)
bd env (EAlt p e1 e2)   =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (EAlt p x1 x2, b1 && b2)
bd env (ECond c e1 e2)  =       let
                                        (xc,bc) = (bd env c)
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
                                        (ECond xc x1 x2, (bc && b1) && b2)
bd env (EFix e)         =       let
                                        (x,b) = (bd env e)
                                in
                                        (EFix x, b)


bd env (EBr e renv)     =       let
                                        (x,b) = (bd (CountBrS env) e)
                                in
                                        (EBr x renv, b)
bd env (ESubst e s)     =       let
                                        (x,b) = (bd env e)
                                in
                                        (ESubst x s, b)
bd env (ERun e1 e2)     =       let
                                        (x1,b1) = (bd env e1)
                                        (x2,b2) = (bd env e2)
                                in
```

```
                                        (ERun x1 x2, b1 && b2)
bd (CountBrZ env) (EEsc e t)
                        = case (eval e env) of
                                (Q x renv) ->
                                        case eqType (getType x) t of
                                                Just Eq -> (x,True)
                                                Nothing -> (getAny t,False)
                                _ -> (getAny t,False)

bd (CountBrS r) (EEsc e t)
                        =       let
                                        (x,b) = (bd r e)
                                in
                                        (EEsc x t, b)

getAny :: Rep t -> Exp p n t
getAny Int              = ELInt 0
getAny Bool             = ELBool True
getAny (Cod renv)       = EBr (ELInt 0) renv
getAny (Arr t1 t2)      = EAbs t1 (getAny t2)
getAny (Prod t1 t2)     = EPair (getAny t1) (getAny t2)
```

### B.2.4. Evaluación de la Sustitución Explícita

```
evalSub :: Subst g gp -> (forall p. Exp p g t) -> RepEnv g
        -> (Exp p gp t, RepEnv gp)
evalSub s e renv    = (evalSubE s e renv, evalSubR s renv)

evalSubE :: Subst g gp -> Exp p g t -> RepEnv g -> Exp p gp t

evalSubE s (ELInt i) r      = ELInt i
evalSubE s (EINeg e) r      = EINeg (evalSubE s e r)
evalSubE s (EIPlus e1 e2) r = EIPlus (evalSubE s e1 r) (evalSubE s e2 r)
evalSubE s (EIMult e1 e2) r = EIMult (evalSubE s e1 r) (evalSubE s e2 r)

evalSubE s (ELBool b) r     = ELBool b
evalSubE s (EBNot e) r      = EBNot (evalSubE s e r)
evalSubE s (EBOr e1 e2) r   = EBOr (evalSubE s e1 r) (evalSubE s e2 r)
evalSubE s (EBAnd e1 e2) r  = EBAnd (evalSubE s e1 r) (evalSubE s e2 r)

evalSubE s (EILess e1 e2) r = EILess (evalSubE s e1 r) (evalSubE s e2 r)
evalSubE s (EIEq e1 e2) r   = EIEq (evalSubE s e1 r) (evalSubE s e2 r)

evalSubE s (EPair e1 e2) r  = EPair (evalSubE s e1 r) (evalSubE s e2 r)
evalSubE s (EPFst e) r      = EPFst (evalSubE s e r)
evalSubE s (EPSnd e) r      = EPSnd (evalSubE s e r)

evalSubE s (EApp e1 e2) r   = EApp (evalSubE s e1 r) (evalSubE s e2 r)
```

```
evalSubE s (EAbs t e) r     = EAbs t (evalSubE (SLft s) e (REnvCons t r))

evalSubE (SSlsh e) (EVar VZ t) (REnvCons _ r)
                = e
evalSubE (SSlsh e) (EVar (VS v) t) (REnvCons tp r)
                = (EVar v t)
evalSubE (SLft s) (EVar VZ t) (REnvCons _ r)
                = (EVar VZ t)
evalSubE (SLft s) (EVar (VS v) t) (REnvCons tp r)
                = evalSubE (SSft tp) (evalSubE s (EVar v t) r)
                    (evalSubR s r)
evalSubE (SSft tp) (EVar v t) (REnvCons _ r)
                = (EVar (VS v) t)

evalSubE s  (EAlt p e1 e2) r    = case (getType e2) of
                    (Arr t1 t2)
                        -> EAlt p (evalSubE (evalPatS p s)
                            e1 (evalPatR p r t1))
                            (evalSubE s e2 r)
evalSubE s (ECond ec e1 e2) r   = ECond (evalSubE s ec r)
                    (evalSubE s e1 r) (evalSubE s e2 r)
evalSubE s (EFix e) r       = EFix (evalSubE (SLft s) e
                    (REnvCons (getType e) r))

evalSubE s (EBr e renv) r   = EBr (bds (CountSBrZ s r) e) renv
evalSubE s (ERun e1 e2) r   = ERun (evalSubE s e1 r) (evalSubE s e2 r)

evalSubE s (ESubst e sp) r  = ESubst (evalSubE s e r) sp


evalSubR :: Subst g gp -> RepEnv g -> RepEnv gp

evalSubR (SSlsh e) (REnvCons t r)   = r
evalSubR (SSft t) r                 = (REnvCons t r)
evalSubR (SLft s) (REnvCons t r)    = (REnvCons t (evalSubR s r))


data CountSBr :: *0 ~> *0 ~> *0 where
    CountSBrZ   :: Subst g gp
                -> RepEnv g
                -> CountSBr (b,Env g) (b,Env gp)
    CountSBrS   :: CountSBr a b
                -> CountSBr (a,c) (b,c)

bds :: CountSBr p pc -> Exp p n t -> Exp pc n t

bds c (ELInt i)         = ELInt i
bds c (EINeg e)         = EINeg (bds c e)
bds c (EIPlus e1 e2)    = EIPlus (bds c e1) (bds c e2)
```

```
bds c (EIMult e1 e2)    = EIMult (bds c e1) (bds c e2)

bds c (ELBool b)        = ELBool b
bds c (EBNot e)         = EBNot (bds c e)
bds c (EBOr e1 e2)      = EBOr (bds c e1) (bds c e2)
bds c (EBAnd e1 e2)     = EBAnd (bds c e1) (bds c e2)

bds c (EILess e1 e2)    = EILess (bds c e1) (bds c e2)
bds c (EIEq e1 e2)      = EIEq (bds c e1) (bds c e2)

bds c (EPair e1 e2)     = EPair (bds c e1) (bds c e2)
bds c (EPFst e)         = EPFst (bds c e)
bds c (EPSnd e)         = EPSnd (bds c e)

bds c (EAbs t e)        = EAbs t (bds c e)
bds c (EApp e1 e2)      = EApp (bds c e1) (bds c e2)
bds c (EVar v t)        = EVar v t
bds c (EAlt p e1 e2)    = EAlt p (bds c e1) (bds c e2)
bds c (ECond ec e1 e2)  = ECond (bds c ec) (bds c e1) (bds c e2)
bds c (EFix e)          = EFix (bds c e)

bds c (EBr e renv)      = EBr (bds (CountSBrS c) e) renv
bds c (ERun e1 e2)      = ERun (bds c e1) (bds c e2)
bds c (ESubst e s)      = ESubst (bds c e) s
bds (CountSBrS c) (EEsc e t)
                        = EEsc (bds c e) t
bds (CountSBrZ s r) (EEsc e t)
                        = EEsc (evalSubE s e r) t


bds c (ELInt i)         = ELInt i
bds c (EINeg e)         = EINeg (bds c e)
bds c (EIPlus e1 e2)    = EIPlus (bds c e1) (bds c e2)
bds c (EIMult e1 e2)    = EIMult (bds c e1) (bds c e2)

bds c (ELBool b)        = ELBool b
bds c (EBNot e)         = EBNot (bds c e)
bds c (EBOr e1 e2)      = EBOr (bds c e1) (bds c e2)
bds c (EBAnd e1 e2)     = EBAnd (bds c e1) (bds c e2)

bds c (EILess e1 e2)    = EILess (bds c e1) (bds c e2)
bds c (EIEq e1 e2)      = EIEq (bds c e1) (bds c e2)

bds c (EPair e1 e2)     = EPair (bds c e1) (bds c e2)
bds c (EPFst e)         = EPFst (bds c e)
bds c (EPSnd e)         = EPSnd (bds c e)

bds c (EAbs t e)        = EAbs t (bds c e)
bds c (EApp e1 e2)      = EApp (bds c e1) (bds c e2)
```

```
bds c (EVar v t)        = EVar v t
bds c (EAlt p e1 e2)    = EAlt p (bds c e1) (bds c e2)
bds c (ECond ec e1 e2)  = ECond (bds c ec) (bds c e1) (bds c e2)
bds c (EFix e)          = EFix (bds c e)

bds c (EBr e renv)      = EBr (bds (CountSBrS c) e) renv
bds c (ERun e1 e2)      = ERun (bds c e1) (bds c e2)
bds c (ESubst e s)      = ESubst (bds c e) s
bds (CountSBrS c) (EEsc e t)
                        = EEsc (bds c e) t
bds (CountSBrZ s r) (EEsc e t)
                        = EEsc (evalSubE s e r) t



evalPatR :: (Pat t eout) -> RepEnv ein -> Rep t -> RepEnv {eapp eout ein}
evalPatR (PLInt i) r rt         = r
evalPatR (PLBool b) r rt        = r
evalPatR (PPair p1 p2) r rt     = case rt of (Prod t1 t2)
                                        -> evalPatR p2 (evalPatR p1 r t1) t2
evalPatR (PVar t) r rt          = REnvCons t r
evalPatR (PAny) r rt            = r
evalPatR (PCod p) r rt          = evalCPatR p r rt

evalCPatR       :: (PatCod f eout) -> RepEnv ein -> Rep (Cod f)
                        -> RepEnv {eapp eout ein}
evalCPatR (PCPVar t) r rt       = REnvCons rt r
evalCPatR PCPAny r rt           = r
evalCPatR PCPFail r rt          = r
evalCPatR (PCLit t) r (Cod f)   = REnvCons (Cod REnvNil) r
evalCPatR (PCVar v t) r rt      = r
evalCPatR (PCLInt i) r rt       = r
evalCPatR (PCLBool b) r rt      = r
evalCPatR (PCINeg p) r rt       = evalCPatR p r rt
evalCPatR (PCIPlus p1 p2) r rt  = evalCPatR p2 (evalCPatR p1 r rt) rt
evalCPatR (PCIMult p1 p2) r rt  = evalCPatR p2 (evalCPatR p1 r rt) rt
evalCPatR (PCILess p1 p2) r rt  = evalCPatR p2 (evalCPatR p1 r rt) rt
evalCPatR (PCIEq p1 p2) r rt    = evalCPatR p2 (evalCPatR p1 r rt) rt
evalCPatR (PCBNot p) r rt       = evalCPatR p r rt
evalCPatR (PCBOr p1 p2) r rt    = evalCPatR p2 (evalCPatR p1 r rt) rt
evalCPatR (PCBAnd p1 p2) r rt   = evalCPatR p2 (evalCPatR p1 r rt) rt
evalCPatR (PCPair p1 p2) r rt   = evalCPatR p2 (evalCPatR p1 r rt) rt
evalCPatR (PCPFst p) r rt       = evalCPatR p r rt
evalCPatR (PCPSnd p) r rt       = evalCPatR p r rt
evalCPatR (PCAbs tx pb) r (Cod f)
                                = evalCPatR pb r (Cod (REnvCons tx f))
evalCPatR (PCApp pf px) r rt    = evalCPatR px (evalCPatR pf r rt) rt
evalCPatR (PCFix tx pb) r (Cod f)
                                = evalCPatR pb r (Cod (REnvCons tx f))
```

```
evalCPatR (PCCond p1 p2 p3) r rt= evalCPatR p3 (evalCPatR p2 (evalCPatR p1 r rt) rt) rt
evalCPatR (PCBr c) r rt         = REnvCons rt r
evalCPatR (PCRun p1 p2) r rt    = evalCPatR p2 (evalCPatR p1 r rt) rt
evalCPatR (PCSubst p ps) r rt   = evalCPatR p r rt
evalCPatR (PCAlt t p p1 p2) r (Cod f)
                                = evalCPatR p2 (evalCPatR p1 r (Cod (evalPatR p f t)))
                                               (Cod f)


evalPatS :: (Pat t eout) -> Subst g gp
                         -> Subst {eapp eout g} {eapp eout gp}
evalPatS (PLInt i) s            = s
evalPatS (PLBool b) s           = s
evalPatS (PPair p1 p2) s        = evalPatS p2 (evalPatS p1 s)
evalPatS (PVar t) s             = (SLft s)
evalPatS (PAny) s               = s
evalPatS (PCod p) s             = evalCPatS p s


evalCPatS :: (PatCod f eout) -> Subst g gp
                         -> Subst {eapp eout g} {eapp eout gp}

evalCPatS (PCPVar t) s          = (SLft s)
evalCPatS PCPAny s              = s
evalCPatS PCPFail s             = s
evalCPatS (PCLit t) s           = (SLft s)
evalCPatS (PCVar v t) s         = s
evalCPatS (PCLInt i) s          = s
evalCPatS (PCLBool b) s         = s
evalCPatS (PCINeg p) s          = evalCPatS p s
evalCPatS (PCIPlus p1 p2) s     = evalCPatS p2 (evalCPatS p1 s)
evalCPatS (PCIMult p1 p2) s     = evalCPatS p2 (evalCPatS p1 s)
evalCPatS (PCILess p1 p2) s     = evalCPatS p2 (evalCPatS p1 s)
evalCPatS (PCIEq p1 p2) s       = evalCPatS p2 (evalCPatS p1 s)
evalCPatS (PCBNot p) s          = evalCPatS p s
evalCPatS (PCBOr p1 p2) s       = evalCPatS p2 (evalCPatS p1 s)
evalCPatS (PCBAnd p1 p2) s      = evalCPatS p2 (evalCPatS p1 s)
evalCPatS (PCPair p1 p2) s      = evalCPatS p2 (evalCPatS p1 s)
evalCPatS (PCPFst p) s          = evalCPatS p s
evalCPatS (PCPSnd p) s          = evalCPatS p s
evalCPatS (PCAbs tx pb) s       = evalCPatS pb s
evalCPatS (PCApp pf px) s       = evalCPatS px (evalCPatS pf s)
evalCPatS (PCFix tx p) s        = evalCPatS p s
evalCPatS (PCCond p1 p2 p3) s   = evalCPatS p3 (evalCPatS p2 (evalCPatS p1 s))
evalCPatS (PCBr r) s            = (SLft s)
evalCPatS (PCRun p1 p2) s       = evalCPatS p2 (evalCPatS p1 s)
evalCPatS (PCSubst p ps) s      = evalCPatS p s
evalCPatS (PCAlt t p p1 p2) s   = evalCPatS p2 (evalCPatS p1 s)
```

### B.2.5. Evaluación de los Patrones

```
evalPat :: (Pat t eout) -> t -> Env ein -> Maybe (Env {eapp eout ein})

evalPat (PLInt i) v env         = if (i==v)
                                        then (Just env)
                                        else Nothing
evalPat (PLBool b) v env        = if (not ((b && not v) || (v && not b)))
                                        then (Just env)
                                        else Nothing
evalPat (PPair p1 p2) (v1,v2) env
                                = case (evalPat p1 v1 env) of
                                        (Just env1)     ->  evalPat p2 v2 env1
                                        Nothing         -> Nothing
evalPat (PVar t) v env          = (Just (EnvCons v env))
evalPat PAny v env              = (Just env)
evalPat (PCod p) v env          = evalCPat p v env


evalCPat :: (PatCod f eout) -> (Cod f) -> Env ein -> Maybe (Env {eapp eout ein})

evalCPat (PCPVar t) e env
                = case e of
                        (Q v renv) ->
                                case (eqType (getType v) t) of
                                        (Just Eq) -> (Just (EnvCons e env))
                                        Nothing -> Nothing
                        _ -> Nothing

evalCPat PCPAny e env
                = (Just env)

evalCPat PCPFail e env
                = case e of
                        (F renv) -> Just env
                        (Q eq renv) -> Nothing

evalCPat (PCLit t)  e env
                = case e of
                        (Q (ELInt i) renv) ->
                                case (eqType Int t) of
                                        Just Eq -> (Just (EnvCons e env))
                                        Nothing -> Nothing
                        (Q (ELBool b) renv) ->
                                case (eqType Bool t) of
                                        Just Eq -> (Just (EnvCons e env))
                                        Nothing -> Nothing
                        _ -> Nothing
```

```
evalCPat (PCVar v t)  e env
                = case e of
                        (Q (EVar vv vt) renv) ->
                                case (eqType t vt) of
                                        Just Eq ->
                                                if (eqVar v vv)
                                                        then Just env
                                                        else Nothing
                                        Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCLInt i)  e env
                = case e of
                        (Q (ELInt v) renv) ->
                                if (i==v)
                                        then (Just env)
                                        else Nothing
                        _ -> Nothing

evalCPat (PCLBool b) e env
                = case e of
                        (Q (ELBool v) renv) ->
                                if (not ((b && not v) || (v && not b)))
                                        then (Just env)
                                        else Nothing
                        _ -> Nothing


evalCPat (PCINeg p) e env
                = case e of
                        (Q (EINeg v) renv) -> (evalCPat p (eval (EBr v renv) env) env)
                        _ -> Nothing

evalCPat (PCIPlus p1 p2) e env
                = case e of
                        (Q (EIPlus v1 v2) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) -> evalCPat p2 (eval (EBr v2 renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCIMult p1 p2) e env
                = case e of
                        (Q (EIMult v1 v2) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) -> evalCPat p2 (eval (EBr v2 renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing
```

```
evalCPat (PCILess p1 p2) e env
                = case e of
                        (Q (EILess v1 v2) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) -> evalCPat p2 (eval (EBr v2 renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCIEq p1 p2) e env
                = case e of
                        (Q (EIEq v1 v2) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) -> evalCPat p2 (eval (EBr v2 renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCBNot p) e env
                = case e of
                        (Q (EBNot v) renv) -> (evalCPat p (eval (EBr v renv) env) env)
                        _ -> Nothing

evalCPat (PCBOr p1 p2) e env
                = case e of
                        (Q (EBOr v1 v2) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) -> evalCPat p2 (eval (EBr v2 renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCBAnd p1 p2) e env
                = case e of
                        (Q (EBAnd v1 v2) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) -> evalCPat p2 (eval (EBr v2 renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing


evalCPat (PCPair p1 p2) e env
                = case e of
                        (Q (EPair v1 v2) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) -> evalCPat p2 (eval (EBr v2 renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCPFst p) e env
                = case e of
                        (Q (EPFst v) renv) -> (evalCPat p (eval (EBr v renv) env) env)
```

```
                        _ -> Nothing

evalCPat (PCPSnd p) e env
                = case e of
                        (Q (EPSnd v) renv) -> (evalCPat p (eval (EBr v renv) env) env)
                        _ -> Nothing

evalCPat (PCAbs tx pb) e env
                = case e of
                        (Q (EAbs tvx vb) renv) ->
                           case (eqType tvx tx) of
                              (Just Eq) -> evalCPat pb
                                           (eval (EBr vb (REnvCons tvx renv)) env) env
                              Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCApp pf px) e env
                = case e of
                        (Q (EApp vf vx) renv) ->
                           case (evalCPat pf (eval (EBr vf renv) env) env) of
                              (Just env1) -> evalCPat px (eval (EBr vx renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCFix tx p) e env
                = case e of
                        (Q (EFix v) renv) ->
                           case (eqType tx (getType v)) of
                              (Just Eq) -> (evalCPat p
                                            (eval (EBr v (REnvCons tx renv)) env) env)
                              Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCCond p1 p2 p3) e env
                = case e of
                        (Q (ECond v1 v2 v3) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) ->
                                   case (evalCPat p2 (eval (EBr v2 renv) env) env1) of
                                      (Just env2) ->
                                             evalCPat p3 (eval (EBr v3 renv) env) env2
                                      Nothing -> Nothing
                              Nothing -> Nothing
                        _ -> Nothing



evalCPat (PCBr r) e env
                = case e of
```

```
                        (Q (EBr ve renv2) renv)
                                -> case (eqType (Cod r) (Cod renv2)) of
                                        Just Eq -> (Just (EnvCons e env))
                                        Nothing -> Nothing
                        _ -> Nothing

evalCPat (PCRun p1 p2) e env
                = case e of
                        (Q (ERun v1 v2) renv) ->
                           case (evalCPat p1 (eval (EBr v1 renv) env) env) of
                              (Just env1) -> evalCPat p2 (eval (EBr v2 renv) env) env1
                              Nothing -> Nothing
                        _ -> Nothing


evalCPat (PCSubst p s)  e env
                = case e of
                        (Q (ESubst ve vs) renv) ->
                                if (eqSubst vs s)
                                        then (evalCPat p
                                                        (eval (EBr ve renv) env) env)
                                        else Nothing
                        _ -> Nothing

evalCPat (PCAlt t p p1 p2)  e env
                = case e of
                        (Q (EAlt vp v1 v2) renv) ->
                           case (getType v2) of
                              (Arr t1 t2) ->
                                   case (eqPat p vp t t1) of
                                      Just Eq ->
                                         case (evalCPat p1 (eval (EBr v1
                                                (evalPatR vp renv t)) env) env) of
                                            Just env1 -> evalCPat p2 (eval
                                                                (EBr v2 renv) env) env1
                                            Nothing -> Nothing
                                      Nothing -> Nothing
                        _ -> Nothing
```

### B.2.6. Verificación de Igualdad y Unificación de Tipos

```
eqVar :: (Var n1 t) -> (Var n2 t) -> Bool
eqVar VZ VZ             = True
eqVar (VS v1) (VS v2)   = eqVar v1 v2
eqVar _ _               = False


eqSubst :: Subst g1 gp1 -> Subst g2 gp2 -> Bool
```

```
eqSubst (SSlsh e) (SSlsh e)     = True
eqSubst (SSft t1) (SSft t2)     = case (eqType t1 t2) of
                                        Just Eq -> True
                                        Nothing -> False
eqSubst (SLft s1) (SLft s2)     = eqSubst s1 s2
eqSubst _ _                     = False


eqPat = eqPAux where
        eqPAux :: Pat t1 e1 -> Pat t2 e2 -> Rep t1 -> Rep t2
                -> Maybe(Equal (Pat t1 e1) (Pat t2 e2))
        eqPAux (PLInt i1) (PLInt i2) Int Int =
                if (i1==i2)
                        then Just Eq
                        else Nothing
        eqPAux (PLBool b1) (PLBool b2) Bool Bool =
                if (not ((b1&&not b2)||(b2&&not b1)))
                        then Just Eq
                        else Nothing
        eqPAux (PPair p11 p12) (PPair p21 p22) (Prod t11 t12) (Prod t21 t22)=
                do{ Eq <- eqPAux p11 p21 t11 t21
                ; Eq <- eqPAux p12 p22 t12 t22
                ; return Eq }
        eqPAux (PVar t1) (PVar t2) _ _ =
                case (eqType t1 t2) of
                        Just Eq -> Just Eq
                        Nothing -> Nothing
        eqPAux PAny PAny t1 t2 =
                case (eqType t1 t2) of
                        Just Eq -> Just Eq
                        Nothing -> Nothing

        eqPAux (PCod (PCPair p11 p12)) (PCod (PCPair p21 p22)) (Cod f1) (Cod f2) =
                do{ Eq <- eqPAux (PCod p11) (PCod p21) (Cod f1) (Cod f2)
                ; Eq <- eqPAux (PCod p12) (PCod p22) (Cod f1) (Cod f2)
                ; return Eq }
        eqPAux (PCod (PCAbs t1 p1)) (PCod (PCAbs t2 p2)) (Cod f1) (Cod f2) =
                do{ Eq <- eqType t1 t2
                ; Eq <- eqPAux (PCod p1) (PCod p2)
                                (Cod (REnvCons t1 f1)) (Cod (REnvCons t1 f2))
                ; return Eq }
        eqPAux (PCod (PCAlt t1 p1 p11 p12)) (PCod (PCAlt t2 p2 p21 p22))
               (Cod f1) (Cod f2) =
                do{ Eq <- eqPAux p1 p2 t1 t2
                ; Eq <- eqPAux (PCod p11) (PCod p21)
                                (Cod (evalPatR p1 f1 t1)) (Cod (evalPatR p2 f2 t2))
                ; Eq <- eqPAux (PCod p12) (PCod p22) (Cod f1) (Cod f2)
                ; return Eq }
        eqPAux (PCod (PCCond p11 p12 p13)) (PCod (PCCond p21 p22 p23))
```

```
               (Cod f1) (Cod f2) =
                do{ Eq <- eqPAux (PCod p11) (PCod p21) (Cod f1) (Cod f2)
                ; Eq <- eqPAux (PCod p12) (PCod p22) (Cod f1) (Cod f2)
                ; Eq <- eqPAux (PCod p13) (PCod p23) (Cod f1) (Cod f2)
                ; return Eq }
        eqPAux (PCod (PCSubst p1 s1)) (PCod (PCSubst p2 s2)) (Cod f1) (Cod f2) =
                if (eqSubst s1 s2)
                        then case (eqPAux (PCod p1) (PCod p2) (Cod f1) (Cod f2)) of
                                        Just Eq -> Just Eq
                                        Nothing -> Nothing
                        else Nothing

        eqPAux (PCod p1) (PCod p2) (Cod f1) (Cod f2) =
                do{ Eq <- eqType (Cod f1) (Cod f2)
                ; Eq <- eqPCod p1 p2 f1
                ; return Eq }
        eqPAux _ _ _ _ = Nothing
        monad maybeM


eqPCod = eqPCAux where
        eqPCAux :: PatCod f c1 -> PatCod f c2 -> RepEnv f
                -> Maybe(Equal (PatCod f c1) (PatCod f c2))
        eqPCAux (PCPVar t1) (PCPVar t2) _ =
                case (eqType t1 t2) of
                        Just Eq -> Just Eq
                        Nothing -> Nothing
        eqPCAux PCPAny PCPAny _ = Just Eq
        eqPCAux (PCLInt i1) (PCLInt i2) _ =
                if (i1==i2)
                        then Just Eq
                        else Nothing
        eqPCAux (PCBr fp1) (PCBr fp2) f =
                case (eqType (Cod fp1) (Cod fp2)) of
                        Just Eq -> Just Eq
                        Nothing -> Nothing

        eqPCAux _ _ _ = Nothing

        monad maybeM

eqType = eqTAux where
    eqTAux :: Rep a -> Rep b -> Maybe(Equal a b)
    eqTAux Int Int = Just Eq
    eqTAux Bool Bool = Just Eq
    eqTAux (Cod REnvNil) (Cod REnvNil) = Just Eq
    eqTAux (Cod (REnvCons e1 r1)) (Cod (REnvCons e2 r2)) =
        do{ Eq <- eqTAux e1 e2
        ; Eq <- eqTAux (Cod r1) (Cod r2)
```

```
        ; return Eq }
    eqTAux (Arr a b) (Arr m n) =
        do { Eq <- eqTAux a m
        ; Eq <- eqTAux b n
        ; return Eq }
    eqTAux (Prod a b) (Prod m n) =
        do { Eq <- eqTAux a m
        ; Eq <- eqTAux b n
        ; return Eq }
    eqTAux _ _ = Nothing
    monad maybeM
```

### B.2.7. Inferencia de Tipos

```
getType:: Exp p n t -> Rep t

getType (ELInt i)       = Int
getType (EINeg e)       = Int
getType (EIPlus e1 e2)  = Int
getType (EIMult e1 e2)  = Int

getType (ELBool b)      = Bool
getType (EBNot e)       = Bool
getType (EBOr e1 e2)    = Bool
getType (EBAnd e1 e2)   = Bool

getType (EILess e1 e2)  = Bool
getType (EIEq e1 e2)    = Bool

getType (EPair e1 e2)   = (Prod (getType e1) (getType e2))
getType (EPFst e)       = case (getType e) of
                                (Prod r1 r2) -> r1
getType (EPSnd e)       = case (getType e) of
                                (Prod r1 r2) -> r2

getType (EAbs t e)      = (Arr t (getType e))
getType (EApp e1 e2)    = case (getType e1) of
                                (Arr r1 r2) -> r2
getType (EVar v t)      = t
getType (EAlt p e1 e2)  = (getType e2)
getType (ECond c e1 e2) = (getType e1)
getType (EFix e)        = (getType e)

getType (EBr e renv)    = Cod renv
getType (ESubst e s)    = case (getType e) of
                                (Cod renv) -> Cod (evalSubR s renv)
getType (ERun e1 e2)    = (getType e2)
getType (EEsc e t)      = t
```