Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Tesis Doctoral

Teoría de Mejoras con Efectos

Martín Arnaldo Ceresa

Director: Dr. Mauro Javier Jaskelioff

Miembros del Jurado: Dr. Miguel Pagano
Dr. Alejandro Diaz Caro
Dr. Hernán Melgratti

Tesis presentada en la Facultad de Ciencias Exactas, Ingeniería y Agrimensura, en cumplimiento parcial de los requisitos para optar al título de

Doctor en Informática

Rosario, 5 de Abril de 2023

Resumen

Optimizar programas es difícil. Al aplicar una transformación a un programa uno debe mostrar que se preserva la semántica del programa, y además, se tiene que asegurar que la transformación es realmente una optimización. El estudio de transformaciones de programas que preserven la semántica de los programas es un área de mucha investigación. Seguimos una línea de trabajo que comienza con la equivalencia observacional de Morris, continúa con la relación de bisimulación aplicativa de Abramsky y el método de Howe, concluyendo finalmente con una línea más reciente donde se agregan efectos algebraicos a la relación de bisimulación aplicativa de Dal Lago, Gavazzo y Levy. Asegurar que una transformación es realmente una optimización, que realmente se mejora el programa, es un camino menos explorado con la teoría de mejoras de Sands como el ejemplo más prominente. En esta tesis, conectamos estos dos caminos obteniendo una *teoría de mejoras* abstracta basada en la relación de bisimulación aplicativa con efectos extendiendo la relación de mejoras a lenguajes con efectos.

Dedicatoria

Mi tesis se la dedico a mi mamá, Silvia Noemí Marcon, que siempre me brindó el soporte que necesité en cada momento. Los hombros más altos que la vida me pudo dar.

Agradecimientos

Quisiera agradecer al programa de becas del CONICET que han brindado parte del soporte económico necesario para completar mis estudios y al CIFASIS por darme una gran oficina. También quisiera agradecer a la Universidad Nacional de Rosario, y en particular a la dirección de la Licenciatura en Ciencias de la Computación, que me permitieron formar parte del plantel docente con lo que pude adicionar económicamente a la beca. Agradezco, además, a mi madre por el suplemento económico que me brindó durante los años que duró la beca. Finalmente, al instituto de Software IMDEA donde me permitieron terminar de escribir el presente documento.

Quiero darle las gracias a mi supervisor, Dr. Mauro Jaskelioff, por su paciencia y todos los consejos que me brindó durante estos años.

Agradezco a todas las personas que forman y formaron parte de la Licenciatura en Ciencias de la Computación. La carrera fue el lugar donde aprendí gran parte de lo que sé, y más importante, me ayudó a desarrollarme como persona, y por mi parte espero haber ayudado a que crezca. A Ana Casali, Raúl Kantor, y Federico Severino Guimpel que siempre estuvieron dispuestos a escuchar mis opiniones. En especial, quisiera agradecer a Guido Macchi por su paciencia, su infinito conocimiento, y las grandiosas discusiones que tuvimos durante los años que tuve el placer de compartir con él, que aunque fueron muchos saben a poco.

Agradezco a las personas con las que compartí oficina: Dante Zanarini, Exequiel Rivas Gada, Guido Martinez, Felipe Gorostiaga, y Antonio Locascio. A su manera, y a la mía, hicimos que todos estos años sean un poco mejores espero que para todos. En especial me gustaría también agradecer al director del CIFASIS, Pablo Granitto, que siempre estuvo dispuesto a escuchar y aconsejarme a su manera.

Agradezco a todos mis amigos que supieron motivarme y empujarme para que continúe en el camino que concluye con éste documento. Incluso a aquellas personas que lo hicieron sin saberlo.

Antes de terminar, agradezco a César Sánchez que hizo que me volviera a enamorar de la ciencia y por motivarme en estos últimos años.

Finalmente, agradezco a mi madre y mi hermana por estar siempre acompañándome en cualquier camino que quisiera tomar.

Gracias por tanto y perdón por tan poco.

Índice general

Capítulo 1

Introducción

Optimizar un programa consiste en modificar un programa para que sea más *eficiente* o utilice menos recursos. Es decir, dado un programa, este se transforma de manera tal que se obtiene otro programa que es *mejor* en aspectos *intensivos* pero que es *equivalente* al original de forma *extensional*. Esto hace que probar que una transformación es efectivamente una optimización sea **difícil**, ya que requiere realizar dos pruebas: por un lado hay que probar que el sentido semántico se mantiene, es decir que nuestro nuevo programa se comporta de forma *similar* al original; mientras que además es necesario probar que la ejecución es mejor que la ejecución del programa original de alguna manera observable y cuantificable.

Si bien el estudio de los costos y la complejidad de los programas es muy relevante para el desarrollo y estudio de algoritmos, durante una época la industria del software se benefició del avance en los componentes electrónicos de la computadora. Mientras la famosa *Ley de Moore* (Moore 2006) estaba en auge y los procesadores duplicaban la cantidad de transistores cada dos años, los *mismos programas* se volvían más y más rápidos *sin tener que modificarlos*, ya que el hardware subyacente mejoraba. No era necesario transformar el programa de manera alguna para obtener una mejora en el tiempo de ejecución, simplemente se cambiaba el procesador o la memoria y todos los procesos eran más rápidos. Esta historia tiene su fin, ya que la perdida de energía en forma de temperatura vuelve inadmisible continuar aumentando el poder de cómputo de los procesadores, obligando a los fabricantes de procesadores a optar por introducir múltiples unidades de procesamiento. Esto obligó entonces a los programadores y diseñadores de software a cambiar la forma de pensar el software y los obligó escribir software que fuese más rápido, es decir, presentar soluciones algorítmicas más rápidas. Incluso se introduce un nuevo requisito dentro del desarrollo de software: escribir software previendo la inclusión de futuras optimizaciones con el objetivo de obtener el mayor provecho del hardware subyacente.

No siempre se cuenta con unidades de procesamiento avanzadas o con la capacidad de renovar el hardware subyacente. Por ejemplo, en unidades de procesamiento embebidas utilizadas en satélites geoestacionarios. A su vez el poder computacional del hardware puede ser extremadamente limitado o incluso el acceso al hardware puede ser limitado, haciendo que los desarrolladores deban escribir programas pensando en el uso correcto y eficiente del hardware

subyacente.

Diferentes áreas de la computación se han dedicado a estudiar el concepto de optimización de programas, donde posiblemente la más prominente sea el área de *compiladores*. Los compiladores son programas que transforman un programa escrito en un lenguaje de alto nivel en un programa *equivalente* en un lenguaje objeto que puede ser ejecutado por una unidad de procesamiento (Muchnick 1998). De forma amplia también podemos pensar a los compiladores como traductores de un lenguaje a otro, y que para realizar dicha tarea de traducción, utilizan varias etapas de procesamiento en las cuales se pueden aplicar transformaciones que no sólo mantienen el sentido semántico del programa sino que además mejoran su rendimiento. Donde mejorar el rendimiento puede ser, por ejemplo, menos instrucciones de código ensamblador, o eliminando fragmentos de código que nunca serán accedidos en la ejecución del programa. Por ejemplo, podemos transformar programas recursivos de cola en bucles `for` (Muchnick 1998, Capítulo 15), a los cuales se les pueden aplicar técnicas conocidas de optimización de bucles (Muchnick 1998, Capítulo 14). Una optimización aún más agresiva es la eliminación de sub-expresiones compartidas (Muchnick 1998, Capítulo 13), donde se busca reemplazar la evaluación de una misma sub-expresión múltiples veces por una sola.

Ya que los compiladores son una secuencia de transformaciones de programas, basan su correctitud en probar que cada transformación aplicada es correcta. Pero para saber si una optimización es correcta además deberíamos probar que realmente el programa que se obtiene es mejor, lo cual introduce la necesidad de comparar propiedades intensivas de la ejecución de programas.

1.1. Análisis de Costos y Análisis Asintótico

El análisis de un algoritmo consiste en predecir el consumo de recursos necesarios por el algoritmo (Cormen et al. 2009, Capítulo 2.2). En general se suelen utilizar como medida el espacio, el tiempo, o la cantidad de instrucciones, que se necesitan para su evaluación. En el caso de tener varios algoritmos que resuelvan el mismo problema, se pueden realizar análisis para determinar cuál es el que mejor se adapta a los requerimientos del sistema.

El *costo* de un programa consiste entonces en realizar una medición precisa de la cantidad de unidades básicas que toma su ejecución. Las unidades básicas es una medición que se realiza contando alguna operación básica como ser acceso a disco, llamadas a función, instrucciones del CPU, entre otras. Podemos diferenciar entonces dos niveles de abstracción: el más abstracto, macro costos, donde se cuentan operaciones de alto nivel como llamadas a función; y un nivel más bajo, micro costos, donde se cuentan operaciones de bajo nivel como ser instrucciones del CPU.

La *complejidad* de un algoritmo se refiere entonces al análisis asintótico de la función de costos que describe el uso de recursos necesarios para la evaluación algoritmo en base al tamaño de la entrada. El tamaño suele considerarse un valor en el dominio de los enteros positivos que codifica el tamaño real de las diferentes entradas del algoritmo, por ejemplo, en bits. Por ejemplo, un algoritmo que toma una lista de elementos y los reordena de forma tal que quedan en orden inverso, en general se dice que tiene una complejidad de orden

$O(n)$, ya que la función de costos del algoritmo es lineal respecto a la longitud de la lista n que toma como entrada.

La complejidad es muy útil para obtener una *intuición* sobre propiedades intensivas de nuestros programas y es muy utilizado al momento de hacer análisis sobre programas y algoritmos. En particular, nos permite además encontrar límites teóricos a ciertos algoritmos, por ejemplo, que no es posible ordenar una lista de elementos en menos de $O(n \log(n))$ utilizando algoritmos de ordenamiento basados en la comparación de elementos. Pero, al momento de comparar la ejecución de programas, estos no se ejecutan de forma asintótica, sino que por tiempo finito con restricciones físicas reales, haciendo que el comportamiento asintótico no sea el mejor análisis. Esto se agrava por el hecho que el análisis asintótico oculta constantes y factores dentro de la complejidad de los programas que juegan un papel importante al momento de comparar la ejecución de dos programas. Otro punto débil de la notación O-grande es cuando se requiere comparar la ejecución de dos programas, por ejemplo donde uno es un *cuanto* más rápido que el otro. Siguiendo el análisis asintótico estaríamos presos a decir que ambos programas comparten su comportamiento, pero sabemos que no comparten el mismo costo. Por lo que el análisis de costos de los programas es el enfoque preferido para el estudio de optimizaciones de programas.

En esta tesis nos concentraremos en el análisis de costos de programas para establecer optimizaciones reales sobre los mismos. A su vez nos concentraremos en unidad de costos macro, relativas al lenguaje, ya que sino estaríamos estableciendo unidades de costos relativas al hardware y perderíamos generalidad. Trabajamos basándonos en la posibilidad de observar *una unidad de costo* abstracta que llamaremos *tick*, que nos permite observar cuando un programa utiliza más recursos que otro, simplemente comparando los ticks necesarios para la evaluación de un programa con los del otro.

1.2. Lenguajes Funcionales

En un principio los lenguajes de programación se desarrollaron partiendo de las instrucciones de las unidades de procesamiento. Esto facilitaba la construcción de compiladores, ya que la traducción era casi directa o bien muy sencilla, pero con la evolución del hardware también vino la necesidad de diseñar nuevos lenguajes que sean capaces de expresar mejor los algoritmos y programas con *mayor abstracción* que estén más cerca del humano que de la unidad de procesamiento.

Los lenguajes funcionales son una respuesta a la necesidad de los programadores de expresar más fácilmente los programas, alejándose de las unidades de procesamiento y del *cómo* se deben de ejecutar los programas, para que los programadores se concentren en *qué* se debe computar. Los lenguajes de programación funcional son descendientes del modelo computacional del *lambda calculus*. Son lenguajes de programación de un alto nivel de abstracción, es decir, se sitúan lejos de las instrucciones que se ejecutan en las unidades de procesamiento. La propiedad fundamental de los lenguajes de programación funcionales es que el mecanismo que acciona la computación es la aplicación de funciones (Hutton 2016), a diferencia de otros paradigmas que se basan en accesos a memoria como son los lenguajes imperativos. Esto permite que los programadores desarrollen nuevas técnicas para facilitar la construcción de

nuevos programas más complejos, y debido a que se encuentran más cerca del lenguaje de la matemáticas, se pueden expresar propiedades sobre los programas de forma más sencilla. A su vez los lenguajes funcionales pueden incorporar otros mecanismos como ser:

Funciones Recursivas. Permiten (y fomentan) la definición de funciones recursivas, funciones que se definen utilizando otra instancia de sí mismas.

Funciones de alto orden. Las funciones son ciudadanos de primer orden, pueden ser argumentos o resultados de otras funciones.

Tipado Estático. El código fuente es tipado previamente a la ejecución, eliminando errores de ejecución.

Evaluación Lazy. Los términos se evalúan a medida que van siendo necesarios para la computación.

Al igual que las matemáticas, los lenguajes funcionales son un excelente medio para explicitar la semántica de los programas relegando decisiones al compilador quien eventualmente deberá emitir código en un lenguaje de menor nivel de abstracción. Por lo que, por definición, los compiladores de lenguajes funcionales realizan transformaciones radicales al código fuente para obtener una secuencia de instrucciones de bajo nivel. Esto permite al compilador realizar varios tipos de transformaciones, de alto nivel, transformando fragmentos de código funcional por otro, transformando fragmentos de código funcional en secuencia de instrucciones, y finalmente, transformando secuencias de instrucciones en otras.

La separación que proponen estos lenguajes entre la definición del programa y la logística de su evaluación, impone una cierta complejidad al momento de realizar el análisis de programas funcionales. Una función es la descripción de un valor en base a varios valores de entrada. Es decir, no establece cómo computar esos valores sino que intenta describir el valor resultante en base a las diferentes observaciones de los valores de entrada. Estos tipos de programas no establecen una correlación directa entre el programa que uno escribe y, por ejemplo, cómo se guardan los datos en memoria, si es necesario el uso de memoria dinámica o memoria estática, entre otras decisiones. Estas tareas son delegadas al compilador, haciendo que el razonamiento sobre el uso de recursos en lenguajes de programación funcional sea complejo, en particular cuando el lenguaje es lazy (Okasaki 1998).

Sin embargo, debido a que, en general, los lenguajes funcionales tienen una semántica clara, el desarrollo de transformaciones resulta fácil e intuitivo. Esto se da principalmente en lenguajes de programación funcional denominados puros con *transparencia referencial*. La transparencia referencial permite realizar un razonamiento ecuacional sobre los programas, permitiendo reemplazar un fragmento de código por otro *equivalente*. En otras palabras, podemos probar transformaciones de programas de forma muy elegante, pero probar optimizaciones sobre programas en lenguajes funcionales es más complicado, ya que no existe una clara conexión entre un programa y las propiedades de su evaluación. Para esto, tenemos que concebir el programa en conjunto con su función de evaluación, es decir, con una semántica dada y no solo una mera transformación de símbolos. Afortunadamente, existe una teoría que se adapta muy bien a los

lenguajes funcionales, y que permite expresar optimizaciones sobre la ejecución de programas, la *teoría de mejoras*.

La Teoría de Mejoras (*Improvement Theory*), desarrollada en los 90s por el Profesor *David Sands* (1990, 1991), busca mezclar conceptos intensivos y extensionales sobre los programas ya que queremos que una transformación sea correcta (extensional) pero también buscamos mejorar nuestros programas (de forma intensiva). Para obtener una teoría de mejoras es necesario introducir conceptos cercanos a las unidades de procesamiento, como memoria o tiempo computacional, a los evaluadores o intérpretes que son quienes definen *cómo* se computan los programas. Por esta misma razón utilizaremos conceptos de costos macros antes mencionados.

Como veremos más adelante, nuestro objeto de estudio son los lenguajes de programación funcionales pero que además introducen *efectos computacionales*. Es decir, no son lenguajes *puros* sino que son lenguajes que introducen operaciones que generan un cierto *efecto computacional*, como ser, entrada y salida, estado global, manejo de errores, no determinismo, operadores probabilísticos, entre otros. El objetivo de esta tesis es el de obtener una teoría de mejoras que nos permitan establecer optimizaciones en lenguajes con efectos.

1.3. Efectos Computacionales y Algebraicos

Los efectos computaciones son aquellos efectos producidos por la evaluación de un término de un lenguaje de programación. Un ejemplo clásico es la entrada/salida en un programa, hay programas que pueden depender de la entrada, o bien, pueden mostrar por pantalla información de la ejecución o incluso el resultado del mismo. Esto rompe la ilusión de que los programas son funciones matemáticas, o al menos, no lo son directamente. Introducir este tipo de nociones a los lenguajes funcionales nos permiten, entre otras cosas, interactuar con el entorno, como además introducir operaciones que permiten describir procesos del área de aplicación. En otras palabras, los programas ya no serán funciones sino que tendrán un comportamiento adicional.

El estudio de efectos computacionales es el foco de atención desde hace varias décadas, comenzando con la visión unificadora de Moggi (Moggi 1989), donde los efectos son observables desde el punto de vista del sistema, e incluso estudios más recientes explorando diferentes estructuras matemáticas (Plotkin y Power 2004). En el lenguaje de programación Haskell, las computaciones que involucran un efecto son marcadas al nivel de tipo con lo que se denominan *mónadas* (Moggi 1989, 1991). Por lo que resulta muy común encontrarse con funciones puras, i.e. $f : A \to B$, y además funciones que tienen algún efecto computacional $mv : A \to M(B)$, donde el efecto se encapsula en M. Un ejemplo es la mónada IO en Haskell, que es la que se encarga de toda la interacción entre programas Haskell y el mundo exterior.

Los efectos algebraicos (Plotkin y Power 2004) presentan un enfoque diferente desde una perspectiva del estudio de la teoría de lenguajes. Los efectos computacionales dentro de la evaluación de un programa se presentan mediante el uso de ciertos *operadores* que introducen efectos impuros. Por ejemplo, *set*/*get* para el manejo de memoria mutable, *read*/*print* para obtener interacción con la entrada y salida, etc. De esta manera, los efectos computacionales quedan restringidos a aquellos introducidos por los operadores. Esto involucra enton-

ces una modificación al lenguaje, relegando la interpretación de los efectos al momento de la evaluación.

1.4. Semántica Operacional y Semántica Denotacional

La semántica operacional se dedica a estudiar los programas en base a cómo se evalúan los operadores del lenguaje, mientras que la denotacional se encarga de encontrarle un sentido al programa definiendo su comportamiento en un dominio matemático que modela su comportamiento. Debido a que el objeto de estudio de la tesis es una teoría de costos con efectos algebraicos, deberemos relacionar conceptos operacionales (costos) y denotacionales (interpretación de los efectos). Es decir, necesitamos mezclar propiedades intensivas con propiedades extensionales, propiedades sobre la ejecución de un programa funcional con el comportamiento del mismo.

1.5. Teoría de Mejoras sin y con Efectos

La teoría de mejoras busca introducir conceptos operacionales dentro de la denotación de programas. Se equipa la relación de evaluación con el costo necesario para evaluar un término en un valor, y luego se utiliza estos costos para establecer relaciones entre programas. Al tener una relación de evaluación exponiendo el costo de términos podemos definir una nueva relación más refinada de equivalencia observacional, obteniendo entonces la noción de mejora. Un programa mejora a otro si no hay contexto que observe un empeoramiento en la evaluación del programa completo. Por lo que la relación de mejora entre programas es un refinamiento de la aproximación observacional entre programas.

Las definiciones observacionales, si bien son muy intuitivas y expresivas, son en general complicadas para desarrollar teorías. Esto es debido a que utilizan cuantificadores universales "no hay contexto que observe...". Por lo que en general, estas definiciones vienen equipadas con un cuerpo teórico que facilita las demostraciones, acotando el espacio de búsqueda, como ser los "lemas de contexto", o derivando teoremas que permitan mejorar programas de forma modular como ser el "álgebra de ticks".

La definición de aproximación observacional deja a libre interpretación qué es observar. Esto permitió que, al exponer información de la evaluación, el costo, Sands pueda observar una mejora, y así, poder definir formalmente mejoras entre programas. En esta tesis utilizaremos la misma idea, pero con la presencia de efectos algebraicos. Partiendo de una noción de equivalencia observacional de programa con efectos, en esta tesis, definimos una noción de mejoras que observe además efectos algebraicos, siguiendo los pasos de la teoría de mejoras tradicional. Esto lo logramos en dos pasos: primero, agregamos a la relación de evaluación de programas a valores con efectos el costo necesario para computar dicho valor; segundo, introducimos un nuevo operador algebraico que representa el consumo de un cómputo de costo, llamado tick. En otras palabras, exploraremos como agregamos el efecto de computar el costo de la evaluación a los efectos presentes en el lenguaje, y además, modificaremos la observación para además tener en cuenta el costo de la evaluación de programas.

Por ejemplo, asumamos un cálculo lambda con una familia operadores binarios $\oplus_r$ que representa la elección probabilística, $p \oplus_r q$ evalúa a p con

probabilidad r o a q con probabilidad $1 - r$. Nos preguntamos entonces como es posible optimizar dicho término, más aún, si podríamos optimizar p obteniendo otro término p', ¿es $p' \oplus q$ una optimización de $p \oplus q$? Peor aún, todavía no contamos con una definición de lo que es mejorar un programa probabilístico.

Introducir una noción de mejoras con efectos no es tan claro como parece, más aún cuando se busca obtener un cuerpo teórico general que nos permita introducir diferentes efectos. Durante la tesis estudiamos además el efecto de no-determinismo y excepciones. Concentramos nuestros esfuerzos en encontrar una teoría *general* que nos permita caracterizar optimizaciones.

1.6. Trabajo Relacionado

Desde el comienzo de la teoría de mejoras en los '90, el estudio de análisis de costos ha avanzado aunque no se puede decir que mucho. La teoría de mejoras tuvo un gran desarrollo mientras fue el foco de estudio del Prof. Sands en los '90, pero se desarrolló muy poco hasta recientemente que ha cobrado relevancia nuevamente en el área (Hackett y Hutton 2019; Handley, Vazou y Hutton 2019). A diferencia del trabajo de Sands, estos trabajos han consistido en buscar definir teorías de mejoras para lenguajes específicos con el objetivo de obtener una relación más refinada que la equivalencia y probar optimizaciones sobre los mismos. Es decir, no se han desarrollado nuevas técnicas en el uso de teorías de mejoras, sino que simplemente se ha aplicado la teoría a diferentes lenguajes con el objetivo de mostrar optimizaciones.

Erratic Fudgets Se ha estudiado agregar operadores, *erratic fudgets*, a un lenguaje funcional reducido con el objetivo de introducir no-determinismo en los programas (Moran, Sands y Carlsson 1999) obteniendo además una teoría de mejoras. Sin ser el objetivo del artículo, los autores Moran, Sands y Carlsson presentan la primer teoría de mejoras en un lenguaje con efectos algebraicos: no determinismo introducido por operadores llamados erratic fudgets. En esta tesis estudiaremos la posibilidad de introducir el efecto de no-determinismo al lenguaje, pero a diferencia de los erratic fudgets, buscaremos hacer observaciones más refinadas sobre las posibles observaciones del sistema. Además, estudiamos lenguajes con efectos algebraicos de forma general, donde primero desarrollamos la teoría de forma paramétrica en los efectos algebraicos.

Un enfoque denotacional para medir complejidad computacional en Lenguajes Funcionales *Katheryn Van Stone* (Van Stone 2003) presenta en su tesis doctoral, como bien dice el título, un análisis de complejidad dentro de lenguajes funcionales con un punto de vista denotacional. Es decir, se introducen nociones de costos dentro de la interpretación de programas funcionales como funciones matemáticas, introduciendo además efectos siguiendo las ideas de Moggi (Moggi 1989). Nuestro enfoque es diferente ya que introducimos efectos mediante operadores, permitiéndonos hacer una análisis más específico a cada lenguaje manteniendo un framework general. Además, nuestro análisis gira entorno al análisis de costo de programas, mediante la adición de un nuevo operador tick.

Espacio Métricos Otra noción posible es interpretar programas con costos dentro de un espacio métrico, y construir una teoría de mejoras alrededor de dicha construcción (Hackett y Hutton 2019). De esta manera se puede formalizar de forma elegante la noción de que la evaluación de un programa es menor que la del otro, y además permite demostrar propiedades muy útiles mapeando propiedades de los espacio métricos. Nuestro enfoque en cambio es diferente, buscamos obtener teorías de mejoras para diferentes lenguajes, introduciendo de forma modular los diferentes operadores que introducen efectos.

Análisis de Costo Relacional Otro trabajo reciente es la definición de relaciones de costos, como ser RelCost (Cicek et al. 2017), donde los autores definen un sistema de tipos refinados y un sistema de efectos donde se agrega el análisis de costos. El objetivo, y resultado, de dicho análisis es obtener una cota estimada del costo de la ejecución para así compararla entre diferentes programas. Difiere a nuestro enfoque en varios aspectos: 1) en esta tesis se deriva una noción de mejora partiendo de la noción de equivalencia, y por lo tanto, las optimizaciones son transformaciones correctas entre programas; 2) en este trabajo es sencillo introducir nuevos efectos ya que 3) el manejo de efectos algebraicos es explícito; 4) no es necesario conocer los costos de ejecución de los programas, simplemente mostrar que uno es menor que el otro.

Un problema que tienen las teorías de mejoras, es que, para cada nuevo lenguaje uno debe desarrollar toda la teoría de nuevo. Si bien se han realizados avances en el área, y es posible obtener teorías de mejoras de algunos lenguajes, no se han encontrado trabajos en lenguajes con efectos algebraicos, por ejemplo, que sean capaces de incorporar efectos con simplicidad sin tener que computar toda una teoría de análisis de costos o complejidad. Aún peor, no siempre es posible establecer una métrica clara al momento de definir cómo se computan los costos de la ejecución de un programa, por ende, no siempre es posible definir una *mejora* directamente. En este trabajo obtendremos una forma sencilla de derivar una teoría de mejoras partiendo de la noción de aproximación observacional. Dicha teoría derivada no siempre es tan refinada, por lo que exploraremos otras métodos para obtener una teoría de mejoras con efectos. La comparación entre teorías de mejoras queda fuera de este trabajo.

1.7. Organización y Enfoque de la Tesis

La tesis se basa en el uso de diferentes técnicas y teorías de las ciencias de la computación:

Aproximación Observacional Nos permite definir de forma clara cuando un programa se comporta observacionalmente similar a otro.

Teorías de Mejoras Nos permite definir una relación capaz de probar optimizaciones sobre programas.

Semánticas de Álgebra Inicial y Álgebras Continuas Nos permiten introducir operadores y definir los lenguajes de forma sencilla.

Semántica Formal y Teoría de Dominios Utilizamos la teoría de dominios para darle sentido semántico a los programas, conectando con álgebras continuas, al momento de definir operadores.

La tesis está dividida principalmente en dos partes:

Teoría de Mejoras sin Efectos: En la primer parte introduciremos los conceptos fundamentales de teorías de mejoras dentro de un lenguaje sin efectos, donde esbozaremos la mayoría de las técnicas que utilizaremos en la segunda parte.

Teoría de Mejoras con Efectos: Se introducirán los efectos algebraicos al lenguaje, y veremos como arribar a una definición de mejoras partiendo de la aproximación observacional.

La segunda parte de la tesis buscará recorrer el mismo camino que la primera, pero agregando efectos algebraicos. De esta manera primero introducimos técnicas generales, y luego nos enfocamos de lleno en los conceptos relacionados a los efectos algebraicos.

El resto de la tesis está dividida en capítulos. En el Capítulo 2 vemos los conceptos básicos que se utilizan a lo largo de la tesis, pero sólo los necesarios para poder desarrollar toda la primer parte de la tesis, es decir, no se introducen los efectos hasta que sean necesarios. En el Capítulo 3 introducimos el lenguaje que utilizamos como vehículo para introducir las nociones de equivalencia en el Capítulo 4 y la teoría de mejoras en el Capítulo 5. Luego comenzamos con la segunda parte de la tesis, se introducen efectos en el Capítulo 6, se define la noción de aproximaciones en el Capítulo 7, y se deriva una teoría de mejoras con efectos en el Capítulo 8. En el Capítulo 9 derivamos la respectiva teoría de mejoras para cada uno de los efectos algebraicos utilizados como ejemplo a lo largo de la tesis y vemos además un límite del enfoque y cómo solucionarlo. En el Capítulo 10 mostramos dos optimizaciones utilizando teorías de mejoras definidas en la tesis. Finalmente concluimos el trabajo con una sección de conclusiones y trabajos futuros en el Capítulo 11.

1.8. Publicaciones

El trabajo de esta tesis está publicado en la revista internacional Science of Computer Programming bajo el título «Effectful improvement theory». A diferencia de la tesis el artículo hace un desarrollo resumido sobre el camino realizado durante la investigación, es decir, se muestra directamente los resultados obtenidos. En este documento nos centraremos en hacer una presentación desarrollando la intuición detrás de la investigación y posibles extensiones al trabajo.

Martin A. Ceresa y Mauro J. Jaskelioff (2022). «Effectful improvement theory». En: *Science of Computer Programming* 217, pág. 102792. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2022.102792. URL: https://www.sciencedirect.com/science/article/pii/S0167642322000259

Otros trabajos

Durante el doctorado, producto de la colaboración con otros investigadores, se obtuvieron las siguientes publicaciones. Estos trabajos no forman parte del presente documento, pero forman parte de mi desarrollo como investigador.

HLola Lola es un lenguaje que permite especificar monitores para la monitorización en tiempo de ejecución de propiedades temporales en sistema síncronos. Una de las ideas fundamentales en la que se basa es en una separación clara entre los datos (teorías) y las propiedades temporales de los mismos (D'Angelo et al. 2005). Dado que los datos utilizados en la mayoría de los monitores son observables, es posible utilizar el potente sistema de tipos de Haskell para finalmente dar una implementación que cumpla con dicha separación. A dicha implementación la denominamos HLola.

Martín Ceresa, Felipe Gorostiaga y César Sánchez (2020). «Declarative Stream Runtime Verification (HLola)». En: *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*. Ed. por Bruno C. d. S. Oliveira. Vol. 12470. Lecture Notes in Computer Science. Springer, págs. 25-43. DOI: 10.1007/978-3-030-64437-6_2. URL: https://doi.org/10.1007/978-3-030-64437-6%5C_2

QuickFuzz QuickFuzz es una herramienta escrita en Haskell que permite reutilizar las librerías para escritas por la comunidad para la generación de casos de prueba. Rápidamente, la idea principal del proyecto consiste sacarle provecho a las representaciones internas que de tipos de datos como ser imágenes, fragmentos de código, archivos comprimidos, documentos (pdfs, docx, odt, entre otros) y archivos multimedia como Ogg, Midi y TTF. En Haskell, en general, los programadores definen una estructura para representar los diferentes tipos de datos, en QuickFuzz definimos técnicas agresivas y automáticas para la generación de elementos arbitrarios de que cumplan con la estructura definida. A estos elementos generados de forma arbitraria, y sin sentido semántico a priori, luego son mutados utilizando herramientas clásicas del área para generar fallos en otras herramientas. Las publicaciones obtenidas fueron:

Gustavo Grieco, Martín Ceresa, Agustín Mista et al. (2017). «QuickFuzz testing for fun and profit». En: *J. Syst. Softw.* 134, págs. 340-354. DOI: 10.1016/j.jss.2017.09.018. URL: https://doi.org/10.1016/j.jss.2017.09.018

Gustavo Grieco, Martín Ceresa y Pablo Buiras (2016). «QuickFuzz: an automatic random fuzzer for common file formats». En: *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. Ed. por Geoffrey Mainland. ACM, págs. 13-20. DOI: 10.1145/2976002.2976017. URL: https://doi.org/10.1145/2976002.2976017

Capítulo 2

Preliminares

El objetivo de este capítulo es introducir al lector a las principales herramientas, ya establecidas en el área, que se utilizan en este trabajo para darle *sentido* a los programas. La tesis se basa principalmente en la *construcción de teorías de mejoras*. Para entender las teorías de mejoras se utilizan a su vez cuestiones básicas del análisis de propiedades de los lenguajes funcionales de programación. El enfoque de la tesis es realizar dos tipos de análisis: uno semántico donde se busca que los programas mantengan su sentido, mientras que a su vez, se busca mejorar propiedades intensivas de los programas.

La idea principal de la teoría de mejoras se basa en refinar la noción de *equivalencia* y restringirla de manera tal que se tome en cuenta el costo de la evaluación de los programas. Esta idea es esencial al desarrollo de esta tesis y es lo que nos guía en el desarrollo de eventuales *nuevas teorías* de mejoras.

En este capítulo se presentan entonces las ideas básicas que utilizaremos a lo largo de la tesis:

- Teoría de Orden y Dominios
- Semántica de Álgebra Inicial

2.1. Teoría de Orden

En esta sección se introducen los conceptos básicos de teoría de orden para luego introducir teoría de dominios y la semántica denotacional en ellos.

Al momento de analizar el comportamiento de lenguajes de programación funcionales nos encontramos con que éstos describen o declaran computaciones centradas en *qué* valor se espera. Éste enfoque contrasta directamente con lenguajes no funcionales donde el objetivo se centra en *cómo* se obtienen dichos valores, a través de accesos a memoria u otras formas. Un enfoque natural es entonces asignarle a cada programa una función que describa su comportamiento. Debemos entonces sortear un problema que es el de la no terminación: ¿qué función denota un programa que no termina? Nos dedicamos entonces a presentar primero una noción de aproximación al comportamiento de programas y definir así la denotación de un programa como su aproximación al infinito, en otras palabras, el límite de una cadena de aproximaciones.

Definición 2.1.1 (Orden Parcial)**.** *Un orden parcial (*poset*) sobre un conjunto D es una relación binaria $(\sqsubseteq_D) \subseteq D \times D$ tal que cumple con las siguientes propiedades:*

Reflexiva: *Para todo $d \in D, d \sqsubseteq_D d$*

Transitiva: *Dados $x, y, z \in D$, $x \sqsubseteq_D y \wedge y \sqsubseteq_D z \implies x \sqsubseteq_D z$*

Anti-simétrica: *Dados $x, y \in D$, $x \sqsubseteq_D y \wedge y \sqsubseteq_D x \implies x = y$*

En particular, en el caso que la relación $(\sqsubseteq_D)$ solo sea reflexiva y transitiva, el par $(D, \sqsubseteq_D)$ se llama *preorden*. En el caso que el orden esté totalmente definido en base al contexto, se omitirá el subíndice de la relación.

Definición 2.1.2 (Cadena y ω-Cadena)**.** *Sea $(P, \sqsubseteq_P)$ un orden parcial, un subconjunto $X \subseteq P$ es una cadena si para todo $x, y \in X$, $x \sqsubseteq_P y$ o $y \sqsubseteq_P x$. Una ω-cadena es una secuencia* infinita *$\{x_n\}_{n<\omega}$ de elementos en P, tales que, $x_i \sqsubseteq_P x_j$ con $i \leq j$.*

Definimos a una función $f : (P, \sqsubseteq_P) \to (Q, \sqsubseteq_Q)$ entre los órdenes parciales $(P, \sqsubseteq_P)$ y $(Q, \sqsubseteq_Q)$ como una función entre los conjuntos de cada uno de ellos.

Definición 2.1.3 (Monotonía)**.** *Sean $(P_1, \sqsubseteq_{P_1})$ y $(P_2, \sqsubseteq_{P_2})$ dos órdenes parciales. Una función $f : P_1 \to P_2$ es* monótona *si y sólo si para todo par de valores $x, y \in P_1$ tales que $x \sqsubseteq_{P_1} y$, entonces $f(x) \sqsubseteq_{P_2} f(y)$.*

Definición 2.1.4 (Conjunto Dirigido)**.** *Sea $(A, \sqsubseteq_A)$ un poset. Un subconjunto $X \subseteq A$ es llamado dirigido si y sólo si todo subconjunto* finito *$X_0 \subseteq X$ tiene una cota superior en X.*

De la definición se desprende que un conjunto dirigido no puede ser vacío. Dado un conjunto dirigido X, tenemos que el conjunto vacío está contenido en X, y por lo tanto la cota superior del conjunto vacío debe estar en X.

Un orden parcial es llamado *predominio* u *orden parcial completo* (CPO) si y sólo si todo conjunto dirigido tiene un supremo. El supremo de un conjunto dirigido X lo notamos como $(\bigsqcup X)$. Además, un predominio es llamado *dominio* u *orden parcial completo con punta* (PCPO) si y sólo si tiene un elemento mínimo, notado generalmente con $\bot$.

Definición 2.1.5 (Función Continua)**.** *Sean $(A_1, \sqsubseteq_{A_1}), (A_2, \sqsubseteq_{A_2})$ dos predominios. Una función $f : (A_1, \sqsubseteq_{A_1}) \to (A_2, \sqsubseteq_{A_2})$ es continua si y sólo si preserva los supremos de los conjuntos dirigidos:*

$$f(\bigsqcup X) = \bigsqcup \{f(x) \mid x \in X\}$$

Para todo conjunto dirigido $X \subseteq A_1$.

A medida que avancemos en la interpretación de lenguajes de programación funcionales la noción de continuidad será **fundamental**. Las funciones continuas son aquellas que mantienen la estructura y dan sentido a transformaciones entre programas manipulando directamente los supremos pero preservando la estructura. Recordar que denotaremos a los programas con el límite de su cadena de aproximaciones, por lo que, funciones continuas representan manipulaciones de programas.

Nos dedicamos entonces a introducir las herramientas necesarias para poder denotar términos del cálculo lambda mediante el uso de dominios.

Productos

El resultado de realizar el producto de predominios es un predominio y sus funciones de proyección son continuas.

Teorema 2.1.1. *Dada una familia de predominios* $(A_i \mid i \in I)$*. Su producto* $\prod_{i\in I} A_i$ *es un predominio bajo el orden componente a componente y las funciones de proyección* $\pi_i : \prod_{i\in I} A_i \to A_i$ *son continuas. Más aún, si todos los predominios* A_i *son dominios, también lo es* $\prod_{i\in I} A_i$*. Sea* $(f_i : B \to A_i \mid i \in I)$ *una familia de funciones continuas, entonces existe una única función continua* $f : B \to \prod_{i\in I} A_i$ *tal que:*

$$\pi_i \circ f = f_i$$

Lema 2.1.1. *Sean* A_1, A_2 *y* A_3 *predominios. Entonces, una función* $f : A_1 \times A_2 \to A_3$ *es continua si y sólo si lo es en ambos argumentos.*

Exponenciales

Finalmente, definimos los exponenciales para poder interpretar las abstracciones de cálculo lambda dentro de los dominios. En otras palabras, tenemos que garantizar que los espacios de funciones o *exponenciales* estén bien definidos.

Teorema 2.1.2. *Sean* A_1 *y* A_2 *dos predominios. El conjunto* $A_2^{A_1} = [A_1 \to A_2]$ *definido por todas las funciones* continuas *de* A_1 *a* A_2 *es un predominio ordenado de forma* extensiva*.*

Sean $f, g : A_1 \to A_2$ *funciones continuas:*

$$f \sqsubseteq g \iff \forall a \in A_1, f(a) \sqsubseteq_{A_2} g(a)$$

Definimos entonces la función de aplicación de una función en un argumento. Dada una función continua entre dos predominios $f : A_1 \to A_2$, y un elemento de $a \in A_1$, definimos la función $ev(f, a) = f(a)$. Podemos ver que ev es continua en sus argumentos, y por Lema 2.1.1, la función ev es continua.

Finalmente, vemos que podemos conectar el producto con funciones continuas de la siguiente forma.

Teorema 2.1.3. *Sean* A_1, A_2 *y* A_3 *predominios. Para toda función continua* $f : A_1 \times A_2 \to A_3$ *existe una única función continua* $g : A_1 \to [A_2 \to A_3]$ *tal que:*

$$f(x, y) = g(x)(y)$$

Puntos Fijos

En esta sección exploramos la noción de punto fijo, que es la que utilizamos para definir funciones recursivas. Durante el resto de la sección, asumimos que D es un dominio.

Teorema 2.1.4. *Sea* $f : [D \to D]$ *una función continua. Entonces, existe el supremo de la cadena* $\{f^n(\bot)\}_{n<\omega}$*:*

$$\mu(f) = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$$

y además satisface las siguiente dos condiciones:

- $\mu(f) = f(\mu(f))$
- $\mu(f) \sqsubseteq d$, *cuando* $f(d) \sqsubseteq d$

En particular, $\mu(f)$ *es el menor punto fijo de* f*.*

Necesitamos que D sea un dominio para comenzar la cadena desde $f(\bot)$. Además, es fácil ver que $\{f^n(\bot)\}_{n<\omega}$ es una cadena por continuidad de f.

Por el teorema anterior, podemos ver que existe una función μ de $[D \to D]$ en D que toma funciones continuas f y las envía a su punto fijo. Se podría probar que μ es continua. Pero vamos a hacer un prueba un poco más general.

Teorema 2.1.5. *Sea* $\phi : [[D \to D] \to D] \to [[D \to D] \to D]$ *el operador continuo definido como:*

$$\phi(F)(f) = f(F(f))$$

con $F : [[D \to D] \to D]$ *y* $f : [D \to D]$*.*

Los puntos fijos de ϕ *son entonces los operadores continuos de punto fijo en* D*, por lo que* μ *es el menor punto fijo de* ϕ*.*

Como resultado obtenemos además que μ es continuo.

Finalmente, cerramos el capítulo con un resultado general del álgebra.

Teorema 2.1.6 (Teorema Knaster-Tarski)**.** *Dado un retículo completo* L *y una función que preserve el orden* $f : L \to L$*, el conjunto de puntos fijos de* f *en* L *es a su vez un retículo completo.*

El teorema de Knaster-Tarski nos presenta con un ínfimo y un supremo en el conjunto de puntos fijos de una función. Esto nos presenta entonces un principio de inducción y un principio de coinducción. El ínfimo de los puntos fijo es el menor de los pre-puntos fijos:

$$\mathit{lfp}(f) = \bigcap \{x \in L \mid f(x) \leq x\}$$

Mientras que el supremo de los puntos fijo es el mayor de los post-puntos fijos:

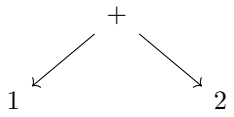
$$\mathit{gfp}(f) = \bigcup \{x \in L \mid x \leq f(x)\}$$

2.2. Semántica Inicial

La semántica inicial busca equipar con una noción de significado a estructuras mediante el uso del objeto inicial. La idea principal del enfoque es que mediante la propiedad de inicialidad podemos caracterizar fácilmente la sintaxis de los lenguajes. Más aún, podemos definir a las funciones de evaluación de términos en dominios semánticos que *respeten* los constructores del lenguaje.

El árbol de sintaxis abstracto de un término es simplemente una construcción detallada de los diferentes operadores del lenguaje y sus términos. Dicho árbol solo nos presenta con un forma de estructurar un término donde un constructor utiliza otros términos del lenguaje, pero no tiene ningún significado. Por ejemplo,

dada una expresión sintáctica como $1+2$, podemos representarla como un árbol con un nodo y dos hojas:



De esta manera, para obtener una función de evaluación del árbol sintáctico de un término en un dominio semántico D, basta con dar una función de interpretación o evaluación de cada operador del lenguaje en el dominio D. En otras palabras, podemos recorrer el árbol de sintaxis abstracto, a cada nodo representando un operador del lenguaje lo interpretamos con su función de evaluación, y hacemos lo mismo con sus argumentos, de forma recursiva.

En esta sección, repasamos los conceptos que utilizamos al momento de definir los operadores algebraicos y cómo utilizar los árboles de sintaxis abstractos para definir la noción de contexto. Finalmente, vemos como obtener funciones de costos analizando los árboles de sintaxis abstracta como una primer aproximación a un análisis de costos.

Dado un conjunto A notamos como A^* a el conjunto resultante de multiplicar A consigo mismo una cantidad finita de veces, $A \times \ldots \times A$, pudiendo incluso ser una cantidad nula $A^0 = \emptyset$. Además, utilizamos a λ para notar a la cadena vacía.

Álgebra Inicial Dentro de una categoría C, intuitivamente, un álgebra S es inicial en una clase $\mathbb{A}$ de álgebras si y solo si para toda $A \in \mathbb{A}$ existe un único homomorfismo $h_A : S \to A$. El árbol de sintaxis abstracto define *un* álgebra inicial, pero como veremos, eso en realidad no es importante ya que las álgebras iniciales son isomorfas, por lo que con caracterizar una nos es suficiente.

Dentro de los lenguajes de programación tiene sentido de hablar de *many-sorted* álgebras, ya que los lenguajes de programación suelen ser poblados de elementos básicos como *enteros, booleanos, cadenas de caracteres, etc*, un álgebra multi-sorted consiste entonces en una familia indexada de conjuntos, llamados *portadores* y una familia de operadores definidos en el producto cartesiano de esos conjuntos. Por ejemplo, un álgebra A con los tres tipos básicos antes mencionados tendría como conjuntos portadores a $\{A_{int}, A_{bool}, A_{string}\}$ con operaciones entre ellos como ser: $(\to_{string}) : A_{bool} \to A_{string} \to A_{string} \to A_{string}$, $(\times) : A_{int} \to A_{int} \to A_{int}$, $(=_{int}) : A_{int} \to A_{int} \to A_{bool}$, que se pueden interpretar como el condicional en *strings*, la multiplicación e igualdad de enteros respectivamente.

Definición 2.2.1 (Signatura Σ)**.** *Sea S un conjunto cuyos elementos llamaremos* sorts*. Una signatura Σ es una familia de conjuntos disjuntos $\langle \Sigma_{w,s} \rangle$ indexados por $S^* \times S$. El conjunto $\Sigma_{w,s}$ contiene los símbolos de los operadores cuyo tipo es $\langle w, s \rangle$, aridad w, sort s, y rango longitud de w.*

Definición 2.2.2 (Σ-Álgebra)**.** *Una Σ-álgebra indexada por sorts S, A, consiste entonces en una familia de conjuntos $\langle A_s \rangle_{s \in S}$ llamados portadores de A, con A_s siendo el portador del sort $s \in S$. Donde para cada $\langle w, s \rangle \in S^* \times S$ y*

para cada $\sigma \in \Sigma_{w,s}$, define una operación σ_A de tipo $\langle w, s\rangle$, es decir, $\sigma_A : A_{w_1} \times \cdots \times A_{w_n} \to A_s$ con $w = w_1 \cdots w_n$ y $w_i \in S$ con $i \in \{1, \ldots, n\}$.

Una operación σ_A de tipo $\langle \lambda, s\rangle$ es una constante de sort s, es decir, $\sigma_A \in A_s$.

Definimos entonces los homomorfismos entre álgebras como familia de homomorfismos entre los conjuntos portadores respetando la estructura de las álgebras.

Definición 2.2.3 (Σ-Álgebra Homomorfismo)**.** *Sean A, A' dos Σ-álgebras. Un homomorfismo $h : A \to A'$ es una familia de funciones $\langle h_s : A_s \to A'_s\rangle_{s\in S}$ tales que respetan los operadores:*

- *Para cada constante $\sigma \in \Sigma_{\lambda,s}$, tenemos que $h_s(\sigma_A) = \sigma_{A'}$*
- *Para cada operador $\sigma \in \Sigma_{w_1 \ldots w_n, s}$ y elementos $\langle a_1 \cdots a_n\rangle \in A_{s_1} \times \cdots A_{s_n}$, tenemos que $h_s(\sigma_A(a_1, \ldots, a_n)) = \sigma_{A'}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$*

Generalizar esto a álgebras con S sorts es sencillo, aunque para esto primero definimos notación para simplificar la escritura.

Familia Indexada Ahora A es una familia de conjuntos indexada por S, $A = \langle A_s\rangle_{s\in S}$, y con $w = s_1 \ldots s_n \in S^*$, podemos generalizar A^n como A^w de forma tal que $A^w = A_{s_1} \times \ldots A_{s_n}$.

Familia de Funciones Indexadas De forma similar podemos extrapolar una familia de funciones $h : A \to A', \langle h_s : A_s \to A'_s\rangle_{s\in S}$, y definir, una función $h^w : A^w \to A'^w$ con $h^w(a_1, \ldots, a_n) = \langle h_{s_1}(a_1), \ldots, h_{s_n}(a_{s_n})\rangle$

Para el caso especial de familias indexadas con $w = \lambda, (n = 0)$ se interpreta como $A^\lambda = A^0 = \{\lambda\}$, y una función σ_A con dominio A^λ la identificamos con una constante cuya valor es σ_{A^λ}.

Ahora entonces podemos definir homomorfismo entre álgebras multi-sorted.

Definición 2.2.4 (S-sorted Álgebra)**.** *Dado un sort S, una Σ-álgebra A con sort S consiste en:*

- *una familia indexada en S de conjuntos A, denominada portador*
- *un mapa que a cada símbolo $\sigma \in \Sigma_{w,s}$ le asigna una función $\sigma_A : A^w \to A_s$*

Definición 2.2.5 (S-sorted álgebra homomorfismo)**.** *Un homomorfismos h de A a B consiste en una familia de funciones indexadas en S, $\langle h_s : A_s \to B_s\rangle_{s\in S}$, de forma tal que para todo símbolo $\sigma \in \Sigma_{w,s}$ con aridad $w \in S^*, s \in S$ tenemos que: $h_s(\sigma_A(a)) = \sigma_B(h^w(a))$.*

En otras palabras que el siguiente diagrama conmute.

$$\begin{array}{ccc} A^w & \xrightarrow{\sigma_A} & A_s \\ {\scriptstyle h^w}\downarrow & & \downarrow{\scriptstyle h_s} \\ B^w & \xrightarrow[\sigma_B]{} & B_s \end{array}$$

Lo interesante de este enfoque, dado que utilizamos teoría de conjuntos, es que nos permite construir el árbol de sintaxis abstracta y una función para consumirlo [1].

Lema 2.2.1 (Existencia de Álgebra Inicial)**.** *La clase de* Σ*-álgebras con* Σ*-álgebra homomorfismos tiene una* Σ*-álgebra inicial denominada* T_Σ*.*

Obviaremos la demostración en esta tesis, ya que es un resultado conocido del área y se pueden encontrar en libros de álgebras universales (Bergman 2011).

El conjunto $T_{\Sigma,s}$, el elemento portador de sort S, lo podemos interpretar como el conjunto de expresiones bien formadas, o más concretamente como árboles de sintaxis abstractos (AST), utilizando los operadores $\sigma \in \Sigma$.

Podemos introducir la noción de álgebra libre de Σ-álgebras y así obtener una definición de operadores derivados.

Definición 2.2.6 (Σ-álgebra libre)**.** *Sea* X *una familia indexada en* S *disjunto respecto a* Σ*, cuyos valores llamaremos* variables*. Cada conjunto indexado* $X_s, s \in S$ *representa variables de sort* s*.*

Podemos entonces formar una jerarquía de conjuntos de la siguiente forma: para cada $s \in S$*,* $\Sigma_{\lambda,s}(X)_0 = \Sigma_s \cup X_s$*; para cada* $k > 0$*,* $\Sigma_{w,s}(X)_k = \Sigma_{w,s}$*.*

Podemos ver que $T_{\Sigma(X)}$ es el álgebra inicial de $\Sigma(X)$. La principal diferencia es que los árboles en el portador pueden tener variables en las hojas además de las constantes que teníamos originalmente. El álgebra $T_{\Sigma(X)}$ tiene como signatura $\Sigma(X)$ pero lo que queremos es obtener una Σ-álgebra, y los podemos lograr definiendo una nueva Σ-álgebra llamada $T_\Sigma(X)$ con portador $T_{\Sigma(X)}$ y símbolos de operaciones definidos en Σ.

Definición 2.2.7 (Σ-álgebra libre generada)**.** *Sea* $h : X \to A$ *una familia de funciones indexada en* S *tal que* $h_s : X_s \to A_s$*, entonces existe un único* Σ*-homomorfismo* $\overline{h} : T_\Sigma(X) \to A$

2.3. Semántica Inicial para Términos del Cálculo Lambda

Veamos como introducir semántica de álgebra inicial para un lenguaje mínimo con variables y expresiones "let–in". Dado un conjunto numerable de variables X, tenemos dos categorías sintácticas:

- Conjunto de Valores: $V, W ::= x \mid (\boldsymbol{\lambda}\ x\ .\ M)$
- Conjunto de Términos: $M, N ::= \mathbf{ret}(V) \mid (V\ W) \mid \mathbf{let}\ x = M\ \mathbf{in}\ N$

Entonces tenemos que para el conjunto de valores podemos definir la signatura

$$\Sigma^V_{w,s} = \begin{cases} var_x \mid x \in X & \text{si } w, s = \lambda, V \\ \{\lambda\} & \text{si } w, s = \langle X, T\rangle, V \\ \emptyset & \text{sino} \end{cases}$$

Donde T es el conjunto de términos definido por la siguiente signatura.

[1] Es posible obtener el mismo resultado utilizando otras teorías, pero para esta tesis basta con tener conjuntos.

En palabras, tenemos que Σ^V representa el conjunto de valores que contiene: las constantes, un elemento por cada elemento de X, y un constructor que toma dos argumentos, una variable y un término, retornando un valor que representa la abstracción.

Para el conjunto de términos tenemos la siguiente signatura:

$$\Sigma^T_{w,s} = \begin{cases} \{ret\} & \text{si } w, s = \langle V \rangle, T \\ \{app\} & \text{si } w, s = \langle V, V \rangle, T \\ \{let_x \mid x \in X\} & \text{si } w, s = \langle T, T \rangle, T \\ \emptyset & \text{sino} \end{cases}$$

Tenemos entonces que la signatura Σ^T representa tres constructores: uno para retornar valores, otro para construir la aplicación de un valor a un término, y finalmente la construcción del operador de substitución explícita.

El lector habrá notado una cierta circularidad en la definición de las dos categorías sintácticas. Los términos dependen de los valores y los valores de los términos. La circularidad es fácilmente evitable definiendo un tipo inductivo de los términos, utilizando álgebras iniciales.

Podemos entonces construir las álgebras iniciales de ambas signaturas, $T_{\Sigma^V}, T_{\Sigma^T}$, y de esta manera definir funciones que toman elementos sintácticos del lenguaje. Veamos entonces como podemos darle semántica a los términos utilizando el siguiente dominio: $D_V \cong I + [D_V \to D_V]$ (Goguen et al. 1977; Scott 1976), con I siendo el conjunto de los números enteros[2] Debido a la presencia de variables en el lenguaje llevaremos un entorno de variables definido simplemente como un mapa entre variables y valores: $E = D_V^X$. Definimos entonces el dominio semántico de los términos y valores como $D \equiv [E \to D_V]$, es decir, es el dominio de funciones continuas de entornos a D_V.

De la definición de D_V tenemos las siguientes funciones de inyección:

$$inj_I : I \to D_V$$
$$inj_{D_V \to D_V} : [D_V \to D_V] \to D_V$$

Veamos primero como construir un álgebra para los valores en el dominio semántico D presentando la interpretación de las variables y las abstracciones. En el caso de las variables, simplemente buscaremos su valor en el entorno, mientras que en el caso de las abstracciones utilizaremos una función que nos permite mapear funciones sintácticas en semánticas. Por ejemplo, mapear la función "$\lambda_x var_x$" a la función matemática identidad.

$$x_D = access_x$$
$$\lambda_D(x, m) = inj_{D \to D} \circ abstract(m \circ assign_x)$$

[2]En nuestro caso tendría más sentido diferenciar el dominio semántico de los valores y términos. Esto es debido a que la evaluación de términos puede divergir. Por simplicidad en esta sección hacemos caso omiso éste planteo, aunque lo retomamos en las siguientes secciones.

Donde definimos las siguientes funciones auxiliares:

$$
\begin{array}{lcl}
access_x & : & E \to D_V \\
access_x(e) & = & e(x) \\
assign_x & : & E \times D_V \to E \\
assign_x(e, v) & = & \lambda y \to \begin{cases} v & \text{si } y = x \\ e(y) & \text{sino} \end{cases} \\
abstract & : & (D_1 \times D_2 \to D_3) \to (D_1 \to (D_2 \to D_3)) \\
abstract(f)(x)(y) & = & f(x, y)
\end{array}
$$

Para interpretar los términos definimos funciones interpretando cada uno de los constructores del lenguaje:

$$
\begin{aligned}
ret_D(v) &= v \\
app_D(v, t) &= ap \circ [v, t] \\
let_D(x, m, t) &= t \circ assign_x \circ [m, 1_e]
\end{aligned}
$$

Utilizando la función auxiliar:

$$
\begin{aligned}
ap &: D_V \times D_V \to D_V \\
ap(f, x) &= \pi_{[D_V \to D_V]}(f)(x)
\end{aligned}
$$

Ahora que tenemos un álgebra que da semántica a la signatura (Σ^V, Σ^T), por inicialidad de $T_{\Sigma^V}, T_{\Sigma^T}$ sabemos que existe un único Σ-álgebra homomorfismo $h_V : T_{\Sigma^V} \to D, h_T : T_{\Sigma^T} \to D$ tal que los diagramas conmutan. Notar que en este caso, tal que los diagramas conmutan, nos da la garantía que las funciones de interpretación son mutuamente recursivas y que la interpretación de los términos utiliza la interpretación de sus subtérminos. De esta manera llevamos términos y valores a un mismo dominio semántico, D_V.

Lo que nos permite definir la equivalencia de programas como:

Definición 2.3.1 (Equivalencia (Semántica) de Programas)**.** *Sean $p_1, p_2 \in T_{\Sigma^T}$, definimos a $p_1 \equiv p_2$ si y solo si la interpretación de p_1 es igual a la interpretación de p_2 en el dominio semántico D_V.*

En palabras, dos términos son equivalentes si sus interpretaciones en el dominio semántico son iguales. Por ejemplo, construcciones sintácticas diferentes como $(\lambda_x.var_x), (\lambda_y.var_y) \in T_{\Sigma^V}$ son mapeadas a la misma función, en este caso la identidad. Estamos simplemente moviendo la discusión de un dominio a otro, lo interesante es que haciendo esto, podemos comparar programas no por su construcción sintáctica sino por su *interpretación semántica*. En este caso, los valores cerrados de nuestro lenguaje son interpretados en funciones matemáticas continuas, y estás pueden ser comparadas siguiendo una noción de equivalencia como ser la de *extensionalidad*, es decir, dos funciones son equivalentes si aplicadas al mismo argumento se obtiene la misma respuesta.

El objetivo de este trabajo es distinguir propiedades intensivas sobre la ejecución de programas, por lo que si quisiéramos utilizar este enfoque deberíamos equipar las interpretaciones semánticas con información acerca de su evaluación.

Es decir, queremos distinguir propiedades de cómo se realiza la evaluación de términos, y más aún, nuestro objetivo es construir una relación entre términos no una noción absoluta de costos. Veamos primero cómo obtener información a través del árbol sintáctico.

2.4. Costos como Álgebras

Como primer aproximación podemos introducir una noción de costo de evaluación dentro de la teoría del álgebra semántica inicial. En otras palabras, deducir el costo de un programa a partir del proceso de interpretación de un término sintáctico en su denotación. Para esto podemos pensar que el análisis de costo de la evaluación es totalmente independiente de los valores que son computados y sólo dependen de la construcción sintáctica de los términos, en otras palabras, dada una signatura Σ: $Cost \in Alg_{\Sigma}$.

Un álgebra de costos es simplemente una forma de asignarle costos a los términos del lenguaje y hay diferentes formas de asignar costos, por ejemplo, computando el tamaño del árbol sintáctico o contando las veces que aparece cierto constructor. Nuestro interés está en analizar propiedades intensivas de la evaluación de términos por lo que nos concentraremos en caracterizar álgebras que expongan propiedades de la evaluación, aunque lamentablemente, estas propiedades no son fáciles de estudiar por sí solas, y no vamos a caracterizar completamente dichas álgebras de costos en esta sección, sino que nos concentraremos en definirlas cuando tengamos la posibilidad de *observar* la evaluación de los términos.

Veamos de todas maneras como podemos trabajar una noción de costo sintácticas utilizando la maquinaria definida hasta el momento.

Definición 2.4.1 (Altura del árbol sintáctico)**.** *Sea Σ una signatura. Definimos el costo de un término como la altura sintáctica del mismo.*

- $\forall \sigma \in \Sigma_{\lambda,s}, \sigma_{\mathbb{N}} = 1$
- $\forall \sigma \in \Sigma_{w,s}, t_1 \dots t_n \in \mathbb{N}^{|w|}, \sigma_{\mathbb{N}} = \max_{i \in n} t_i + 1$

Definición 2.4.2 (Cantidad de constantes utilizadas en el árbol sintáctico)**.** *Sea Σ una signatura. Definimos el costo de un término como la cantidad de constantes que aparecen en su árbol sintáctico.*

- $\forall \sigma \in \Sigma_{\lambda,s}, \sigma_{\mathbb{N}} = 1$
- $\forall \sigma \in \Sigma_{w,s}, t_1 \dots t_n \in \mathbb{N}^{|w|}, \sigma_{\mathbb{N}} = \sum_{i \in n} t_i$

Ambas nociones de costos si bien válidas o *razonables*, no se ajustan a lo que consideraríamos una noción real del costo de la evaluación de términos, de hecho no tienen relación alguna con la evaluación de los términos. Todavía no hay una conexión real entre el cómputo de los costos de la evaluación y el álgebra de costos. Las nociones definidas hasta el momento son nociones estáticas, resultado de *recorrer* y analizar el árbol sintáctico de un término, mientras que el análisis de la evaluación de términos es de carácter dinámico.

Este enfoque tiene el problema que el costo no tiene conexión alguna con la evaluación de los términos, sino que simplemente extraen información del árbol

sintáctico. Por ejemplo, podríamos describir sintácticamente un valor con un árbol sintáctico muy grande, cuya evaluación no sería costosa, mientras que un término cuya evaluación podría ser costosa con un árbol sintáctico pequeño.

Peor aún si llevamos la idea más básica de costos al dominio semántico antes presentado estaremos en problemas, ya que no hay una noción clara para acumular los costos de los valores. En otras palabras, no es fácil definir el dominio semántico de nuestros programas, recordemos que asumíamos el dominio de Scott:

$$D \equiv I + [D \to D]$$

Pero por ejemplo, no está tan claro que exista un dominio tal que a cada valor se le pueda asignar el valor que se utilizó al computarlo, en principio porque depende del término de partida, es decir, depende del contexto. Por ejemplo:

$$D \times \mathbb{N} \equiv I \times \mathbb{N} + [D \to D] \times \mathbb{N}$$

Pero los elementos de $[D \to D] \times \mathbb{N}$ no terminan de caracterizar el costo requerido para computar un valor, en el sentido que se acumula el costo utilizado para computarlo pero dependiendo del contexto también un valor puede ser utilizado para continuar computando, y por ende, tendría más sentido en pensar que son funciones de valores con costos en valores con costos.

Finalmente, nuestro objetivo no es establecer el costo de evaluar un término, sino que queremos saber cómo es posible mejorarlo, y para esto, queremos obtener una relación entre términos. Para esto definiremos relaciones entre álgebras en la siguiente sección.

2.5. Congruencia y Relaciones entre Álgebras

Las preguntas de esta sección son: ¿qué podemos *observar* de relaciones sobre los términos sintácticos?, ¿es posible construir una relación entre términos que observe su evaluación y relacione aquellos términos que presentan una mejora?

La noción de congruencia está definida como *relación algebraica* entre términos construidos a partir de la signatura Σ.

Definición 2.5.1 (Σ-congruencia)**.** *Sea Σ una signatura y A, B dos Σ-álgebras. Definimos una Σ-congruencia entre A y B como una familia de relaciones $\langle R_s \subseteq A_s \times B_s \rangle_{s \in S}$ tal que se puede extender a una sub-álgebra del producto de $A \times B$. Es decir, la familia de relaciones $\langle R_s \rangle_{s \in S} \in Alg_\Sigma$ define una Σ-álgebra tal que las proyecciones $\pi_{s,A} : R_s \to A_s, \pi_{s,B} : R_s \to B_s$ son Σ-homomorfismos a A y B respectivamente.*

$$\begin{array}{ccccc} A^w & \xleftarrow{\pi_A^w} & R^w & \xrightarrow{\pi_B^w} & B^w \\ {\scriptstyle \sigma_A}\downarrow & & \downarrow{\scriptstyle \sigma_R} & & \downarrow{\scriptstyle \sigma_B} \\ A & \xleftarrow[\pi_A]{} & R & \xrightarrow[\pi_B]{} & B \end{array}$$

Sea $A \in Alg_\Sigma$ una Σ-álgebra inicial y R una Σ-congruencia en A con proyecciones $(\pi_1, \pi_2 : R \to A)$. Por inicialidad de A existe un único homomorfismo

$(i : A \to R)$ tal que:

$$\pi_1 \circ i = 1_A = \pi_2 \circ i$$

En otras palabras, esto implica que para todo $a \in A, i(a) = (a, a) \in R$, por lo que R es una relación reflexiva, $(=_A \subseteq R)$. Adicionalmente la relación $=_A$ es una Σ-congruencia en A, por lo que hemos demostrado el siguiente teorema.

Teorema 2.5.1. *Sea A una álgebra inicial de Σ, la relación de igualdad $=_A$ en A es la menor Σ-congruencia:*

$$=_A = \bigcap \{R \subseteq A \times A | R \text{ es una } \Sigma\text{-congruencia en } A\}$$

Sea una Σ-álgebra $A \in Alg_\Sigma$ y $O \subseteq A \times A$ una familia de preórdenes en S, $(\pi_1, \pi_2 : O \to A)$, de forma tal que O sea una Σ-álgebra:

- Para cada $\sigma \in \Sigma_{w,s}$, dados pares de valores relacionados $p_1, \cdots, p_n \in O_{s_1}, \cdots, O_{s_n}$ tenemos que $\sigma_O(p_1, \cdots, p_n) \in O_s \subseteq A_s \times A_s$.
- Para cada $\sigma \in \Sigma_{\lambda,s}$, $\sigma_O \in O_s \subseteq A_s \times A_s$

Álgebras Continuas

Interpretamos la evaluación de términos como una secuencia de aproximaciones. Dado que los términos están representados como un constructor y sus argumentos, la evaluación se basa en interpretar sus argumentos y luego el constructor en sí. El proceso de evaluación de un constructor y sus argumentos se repite una cantidad finita de veces, primero interpretando hasta un constructor, luego hasta dos, luego hasta tres, y así sucesivamente. Dicho proceso define entonces una secuencia de evaluaciones aproximadas. Por lo que definimos a la evaluación como el supremo de su cadena de aproximaciones y por lo tanto utilizamos álgebras que respeten el orden de dichas cadenas. Es decir, utilizamos conjuntos parcialmente ordenados y restringimos nuestro universo de álgebras a aquellas que respetan el orden de los conjuntos ordenados con los que trabajemos.

Dado que mezclamos la semántica de los programas con propiedades intensivas de la evaluación utilizamos órdenes parciales completos. Recordemos que un CPO es un orden parcial de forma tal que todo conjunto dirigido tiene supremo.

En esta sección nos concentramos en definir álgebras cuyos portadores son CPO.

Definición 2.5.2 (Σ-Álgebra Ordenada)**.** *Sea Σ una signatura indexada por sorts S. Una Σ-álgebra ordenada consiste entonces en una familia de CPOs $\langle A_S \rangle_{s \in S}$ donde A_s es el portador del sort $s \in S$. Adicionalmente vamos a requerir que la interpretación de los símbolos de los operadores sean funciones continuas, es decir, que mantengan el sentido de las cadenas.*

Para todo $\langle w, s \rangle \in S^* \times S$ y $\sigma \in \Sigma_{w,s}$, define una función continua *en todos sus argumentos $\sigma_A : A_{w_1} \times \cdots \times A_{w_n} \to A_s$ con $w = w_1 \ldots w_n$ y $w_i \in S$ con $i = 1 \ldots n$.*

En otras palabras, un álgebra es ordenada si interpreta cada uno de los operadores como funciones continuas, es decir, como funciones que respetan el *sentido* de sus operandos.

Capítulo 3

Lenguaje

En este capítulo presentamos el lenguaje que utilizaremos como vehículo a lo largo de la tesis. Es un lenguaje basado en el cálculo lambda no tipado. A diferencia de una introducción clásica del cálculo lambda, definimos una variante de cálculo lambda *computacional* (PaulBlain Levy, Power y Thielecke 2003; Moggi 1989) con dos categorías sintácticas: una para valores y otra para términos. Dentro de los términos se agrega una construcción de ligadura de términos a variables o **let**. El uso del operador **let** no es simplemente *azúcar sintáctico* sino que introduce el orden de evaluación explícitamente.

El lector con conocimientos básicos en el cálculo lambda puede saltar el capítulo una vez vista la definición del lenguaje con dos categorías sintácticas.

El capítulo está divido principalmente en dos, primero se introduce la sintaxis del lenguaje, y luego la evaluación de términos cerrados en valores.

3.1. Sintaxis del Lenguaje

En esta primer parte del capítulo presentamos los conceptos sintácticos básicos que utilizaremos a lo largo de la tesis.

Definición 3.1.1 (Lenguaje)**.** *Dado un conjunto numerable de variables* $\mathbb{V}$*.*

Definimos el conjunto de valores $\mathcal{V}$ *como sigue:*

$$V, W ::= x \mid (\boldsymbol{\lambda}\ x\ .\ M)$$

Y a su vez el conjunto de términos Λ*:*

$$M, N ::= \mathbf{ret}(V) \mid (V\ W) \mid \mathbf{let}\ x = M\ \mathbf{in}\ N$$

Donde los nombres de las variables son elementos del conjunto $\mathbb{V}$*, es decir,* $x \in \mathbb{V}$*.*

De esta manera tenemos dos categorías sintácticas una para valores, $\mathcal{V}$, y otra para términos, Λ. Las funciones de evaluación tomarán un término M en Λ y buscarán asignarle un valor V en $\mathcal{V}$.

A continuación definimos notaciones y convenciones para simplificar la escritura de lambda términos y que utilizaremos a lo largo de la tesis.

Notación 3.1.1. *Introducimos las siguientes notaciones:*

- *Usaremos las letras mayúsculas U, V y W para notar valores en $\mathcal{V}$*
- *Usaremos las letras mayúsculas $M, N, L, P, \ldots$ para notar términos en Λ*
- *Usaremos las letras minúsculas $x, y, z, \ldots$ para notar variables en $\mathbb{V}$*
- *Se omitirán los paréntesis más externos*
- *El símbolo $\equiv$ denota la equivalencia* sintáctica.

De esta manera quedan definidos los conjuntos de valores $\mathcal{V}$ como: variables o lambda abstracciones, mientras que el conjunto de términos Λ como: valores, aplicaciones entre valores o substitución explícita de un término en otro. En la mayoría de las presentaciones de el cálculo lambda (Barendregt 1985; Pierce 2002), éste se introduce sin substitución explícita, operador **let** en nuestro caso, y con la aplicación entre términos directamente. Sin embargo esta presentación es equivalente ya que a cada aplicación entre términos, en el lambda cálculos clásico, $(M\ N)$[1] es posible reescribirla como **let** $x = M$ **in** (**let** $y = N$ **in** $x\ y$) en el lenguaje dado por la Definición 3.1.1 con x, y variables frescas.

Notar que el operador de ligadura de variables (**let** $x = M$ **in** N) **no** es *recursivo*. Es decir, la variable x no va a ser reemplazada por M dentro de M.

Convención 3.1.1. *Introducimos las siguientes convenciones para simplificar la notación:*

- *Aplicación es asociativa a izquierda y se omitirán paréntesis innecesarios*
- *Se podrá agrupar lambda abstracciones de la siguiente manera, dadas variables $x_1, x_2, \ldots, x_n \in \mathbb{V}$, y término $M \in \Lambda$:*

$$\boldsymbol{\lambda}\ x_1\ .\ \mathbf{ret}(\boldsymbol{\lambda}\ x_2\ .\ \mathbf{ret}(\ldots(\boldsymbol{\lambda}\ x_n\ .\ M))) \equiv \boldsymbol{\lambda}\ x_1\, x_2\ \ldots\ x_n\ .\ M$$

Variables Libres y Ligadas

Definimos el conjunto de variables de un término T, $\mathbb{V}(T)$, simplemente como las variables que aparecen en él. A su vez, notamos de la misma manera al conjunto de variables de un valor V, $\mathbb{V}(V)$.

Definición 3.1.2 (Variables Ligadas)**.** *Definimos las variables* ligadas *como aquellas variables que se definen en una lambda abstracción o en una substitución explícita.*

Definición 3.1.3 (Variables Libres)**.** *Definimos las variables libres como aquellas variables que aparecen en un término pero no están* ligadas *bajo ninguna abstracción o substitución explícita.*

[1] El término $M\ N$ no forma parte de nuestro lenguaje.

Formalmente notamos las variables libres de un término T como $\mathsf{FV}_\Lambda(T)$, mientras que las variables libres de un valor W como $\mathsf{FV}_\mathcal{V}(W)$.

$$\begin{array}{rcl} \mathsf{FV}_\mathcal{V} & : & \mathcal{V} \to \mathbb{V} \\ \mathsf{FV}_\mathcal{V}(x) & \doteq & \{x\} \\ \mathsf{FV}_\mathcal{V}(\boldsymbol{\lambda}\ x\ .\ M) & \doteq & \mathsf{FV}_\Lambda(M) \setminus \{x\} \end{array}$$

$$\begin{array}{rcl} \mathsf{FV}_\Lambda & : & \Lambda \to \mathbb{V} \\ \mathsf{FV}_\Lambda(\mathbf{ret}(V)) & \doteq & \mathsf{FV}_\mathcal{V}(V) \\ \mathsf{FV}_\Lambda(V\ W) & \doteq & \mathsf{FV}_\Lambda(V) \cup \mathsf{FV}_\Lambda(W) \\ \mathsf{FV}_\Lambda(\mathbf{let}\ x = M\ \mathbf{in}\ N) & \doteq & \mathsf{FV}_\Lambda(N) \setminus \{x\} \cup \mathsf{FV}_\Lambda(M) \end{array}$$

Utilizando la definición de variables libres, definimos formalmente las variables ligadas como el conjunto de las variables que aparecen en un término menos el conjunto de variables libres: $\mathsf{BV}_\Lambda(T) \doteq \mathbb{V}(T) \setminus \mathsf{FV}_\Lambda(T)$ y $\mathsf{BV}_\mathcal{V}(V) \doteq \mathbb{V}(V) \setminus \mathsf{FV}_\mathcal{V}(V)$.

Hay variables que pueden aparecer libres y ligadas en el mismo término. Por ejemplo:

$$x\,(\mathbf{let}\ x = (\boldsymbol{\lambda}\ x\ .\ \mathbf{ret}(x))\ \mathbf{in}\ x)$$

En este caso se considera que la variable x aparece libre por más que luego aparezca ligada dentro de un operador **let** y una abstracción.

Notación 3.1.2. *Notamos $\mathcal{V}(X)$ al conjunto de valores con variables libres en X, esto es: $\mathcal{V}(X) \doteq \{W \mid \mathsf{FV}_\mathcal{V}(W) \subseteq X\}$.*

Mientras que para los términos lo notamos como $\Lambda(X)$, el conjunto de términos con variables libres en X: $\Lambda(X) \doteq \{M \mid \mathsf{FV}_\Lambda(M) \subseteq X\}$

Finalmente, notamos como $\mathcal{V}_0, \Lambda_0$ a los conjuntos de valores y términos sin variables libres, respectivamente. Es decir, $\mathcal{V}_0 \doteq \mathcal{V}(\emptyset)$ y $\Lambda_0 \doteq \Lambda(\emptyset)$

Substitución

La *substitución* es el mecanismo fundamental para la evaluación de términos del cálculo lambda. En nuestro caso, por la construcción sintáctica presentada en la definición del lenguaje (Definición 3.1.1), la substitución se define reemplazando variables por valores, esto se debe a que las variables solo ocurren en lugares donde pueden ir valores.

Definición 3.1.4. *Debido a las dos categorías sintácticas definimos entonces dos substituciones, una para términos y otra para valores, de una variable $y \in \mathbb{V}$ por un valor $W \in \mathcal{V}$ con captación de variables. La substitución en un término N, la notamos $N[y := W]$, mientras que la substitución sobre un valor V, la notamos $V[W/y]$. Definimos la substitución de forma recursiva sobre la estructura de términos y valores de la siguiente manera:*

$$\begin{array}{lcl} x[W/y] & \doteq & \begin{cases} x \text{ si } x \neq y \\ W \text{ si } x = y \end{cases} \\ (\boldsymbol{\lambda}\ x\ .\ M)[W/y] & \doteq & \boldsymbol{\lambda}\ x\ .\ (M[y := W]) \end{array}$$

$$\begin{array}{lcl} V[y := W] & \doteq & V[W/y] \\ (V\,V')[y := W] & \doteq & (V[W/y])\,(V'[W/y]) \\ (\mathbf{let}\ x = M\ \mathbf{in}\ N)[y := V] & \doteq & \mathbf{let}\ x = M[y := W]\ \mathbf{in}\ N[y := W] \end{array}$$

Notar que la substitución podría capturar variables, y modificar la definición del programa. Por ejemplo,

$$(\boldsymbol{\lambda}\ x\ .\ x)[(x\,(\boldsymbol{\lambda}\ y\ .\ y))/x] = \boldsymbol{\lambda}\ x\ .\ (x\,(\boldsymbol{\lambda}\ y\ .\ y))$$

Para evitar la captura de variables deberemos tener cuidado al momento de realizar substituciones, ya que estas introducen un comportamiento extraño y anómalo en el lenguaje. Capturar variables libres implica asignarles una definición a variables que en principio están definidas por el contexto.

Para aliviar la manipulación de términos y valores en esta tesis utilizaremos la siguiente convención sobre los términos con los que estaremos trabajando.

Convención 3.1.2. *Asumiremos que dados $M_1, M_2, \ldots, M_n$ términos del lenguaje, en cualquier contexto matemático, las variables ligadas serán distintas que las variables libres de todos los términos (Barendregt 1985, Sección 2.1.13)*[2].

De esta manera separamos el espacio de nombre de las variables por sus roles sin mezclarlas. Es decir, no se puede usar el mismo nombre para una variable libre que una ligada. Por lo que en el anterior ejemplo, ya no es posible realizar la substitución. Es posible dar un algoritmo para traducir términos y valores que no respeten dicha convención en términos y valores que sí la respeten. Por lo tanto asumimos que todos los términos la respetan.

Probamos un lema útil al momento de realizar substituciones.

Lema 3.1.1 (Lema de Substitución)**.** *Dado un términos $M \in \Lambda$, valores $V, U, W \in \mathcal{V}$, y dos variables $x, y \in \mathbb{V}$, tal que $x \notin \mathsf{FV}(W)$:*

- $M[x := V][y := W] \equiv M[y := W][x := V[y := W]]$
- $U[V/x][W/y] \equiv U[W/y][V[W/y]/x]$

Ya que el lenguaje se compone de dos categorías sintácticas la demostración consiste en inducción simultánea sobre el término M y el valor U. La demostración formal queda fuera de la tesis siendo éste un resultado conocido del área y la prueba no resulta relevante.

3.2. Evaluación

En esta sección, definimos la evaluación de términos en Λ a valores en $\mathcal{V}$. En concreto la evaluación de programas, es decir, de términos sin variables libres. Debido a que el lenguaje se define como dos categorías sintácticas (Definición 3.1.1), las relaciones de evaluación deben ser definidas de forma mutuamente recursiva entre ellas. Ambas relaciones son parciales, ya que habrá términos a los cuales no será posible asignarles valores.

Debido a la naturaleza del lenguaje, la sintaxis guía la evaluación de los términos. La aplicación de funciones se define en base a dos valores, es decir, que a diferencia del cálculo lambda tradicional, no es posible dar una función de evaluación que primero evalúe el cuerpo de la función y luego sus argumentos (*call-by-name*), y otra que evalúe los argumentos y luego el cuerpo de la

[2]La notación fue introducida en realidad por *Thomas Ottmann* en 1972. Para más información ver las correcciones en la sexta impresión del libro de Barendregt(1985).

función (*call-by-value*). La evaluación sigue una evaluación de *call-by-value*, ya que los argumentos de la aplicación de función se evalúan antes de aplicarles la función correspondiente.

La definición del lenguaje de esta manera tiene una propiedad que nos va a ser útil en la segunda parte de la tesis. Los términos del lenguaje describen sintácticamente cómo deben ser evaluados, que resulta fundamental al momento de computar el orden de los efectos.

El objetivo de esta sección es entonces definir una relación entre términos en Λ_0 con valores en $\mathcal{V}_0$. No caracterizamos la evaluación *parcial* de términos que tengan variables libres. Primero repasamos conceptos bien establecidos en la literatura para luego definir la evaluación de términos. Se recomienda a el lector que ya tenga conocimientos sobre evaluación de términos en el cálculo lambda pasar directamente a la Sección 3.3.

Semántica Operacional

Una forma de evaluación de términos es describir cómo son ejecutados los términos dentro de una máquina abstracta. Las máquinas abstractas van desde abstracciones concretas de unidades de procesamiento hasta de alto nivel cercanas a los términos.

En nuestro caso, a modo de ejemplo, veremos dos semánticas operacionales para el conjunto de términos de nuestro lenguaje.

Evaluación de Pasos Cortos

La evaluación de pasos cortos indica como evaluar localmente cada una de las construcciones del lenguaje. Se llaman de *pasos cortos* ya que se definen pequeñas modificaciones de forma tal que se pueda ir traduciendo un término en otro de forma local hasta llevarlo, de ser posible, a un término que represente un valor.

Definición 3.2.1 (Evaluación Pasos Cortos)**.** *Definimos la evaluación a pasos cortos como una relación entre términos,* $(\to) \subseteq \Lambda \times \Lambda$*, que determina la evaluación describiendo un paso local.*

$$\frac{}{(\boldsymbol{\lambda}\ x\ .\ M)\ V \to M[x := V]}$$

$$\frac{}{\mathbf{let}\ x = \mathbf{ret}(V)\ \mathbf{in}\ M \to M[x := V]}$$

$$\frac{M \to M'}{\mathbf{let}\ x = M\ \mathbf{in}\ N \to \mathbf{let}\ x = M'\ \mathbf{in}\ N}$$

Notar que $(\to)$ es una relación parcial ya que no todos los términos se relacionan con otro término. Por ejemplo un término con variable libre x, $x\,V$ con $V \in \mathcal{V}$ no evalúa a ningún otro término. Esto es ya que al ser x una variable libre no se conoce su definición y en principio podría tomar cualquier definición.

Para subsanar esta limitación podemos definir una relación de evaluación directamente sobre términos cerrados $\to_\emptyset \subseteq \Lambda_0 \times \Lambda_0$ siguiendo la misma definición que $(\to)$. Esta solución no es satisfactoria ya que deja afuera términos abiertos que aceptan un valor, por ejemplo, si tienen variables libres que no utilizan.

Definimos además la *clausura reflexiva y transitiva* de la relación $(\to)$ como $(\to^*) \subseteq \Lambda \times \Lambda$:

$$\frac{}{M \to^* M}$$

$$\frac{M \to M'' \qquad M'' \to^* M'}{M \to^* M'}$$

De las definiciones tenemos que $(\to) \subseteq (\to^*)$. Sean $M, N \in \Lambda$ tales que $M \to N$. Tenemos entonces por definición que $N \to^* N$ por reflexividad, y en conjunto con $M \to N$, podemos concluir que $M \to^* N$.

Evaluación a Grandes Pasos

La evaluación por pequeños pasos permite observar localmente el proceso de la reducción de un término, pero también es posible definir otra relación que directamente relacione términos con valores.

Definición 3.2.2 (Evaluación a Grandes Pasos)**.** *Definimos entonces la evaluación de un término en un valor,* $(\Downarrow) \subseteq \Lambda \times \mathcal{V}$ *como:*

$$\frac{}{\mathbf{ret}(V) \Downarrow V}$$

$$\frac{M[x := W] \Downarrow V}{(\boldsymbol{\lambda}\ x\ .\ M)\, W \Downarrow V}$$

$$\frac{M \Downarrow V' \qquad N[x := V'] \Downarrow V}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow V}$$

Ejemplo 3.2.1. *Definimos la identidad como la familia de términos* α*-equivalentes* $I_x \doteq \boldsymbol{\lambda}\ x\ .\ \mathbf{ret}(x)$*. En el lambda cálculo tradicional, la identidad es definida simplemente retornando el argumento, pero como en nuestro lenguaje tenemos una diferencia entre términos y valores, debemos utilizar el constructor* **ret** *para retornar un término.*

Podemos ver la evaluación a grandes pasos de la identidad aplicada a la identidad es la identidad.

$$= \frac{\dfrac{}{\mathbf{ret}(\boldsymbol{\lambda}\ y\ .\ \mathbf{ret}(y)) \Downarrow (\boldsymbol{\lambda}\ y\ .\ \mathbf{ret}(y))}}{\mathbf{ret}(x)[x := (\boldsymbol{\lambda}\ y\ .\ \mathbf{ret}(y))] \Downarrow (\boldsymbol{\lambda}\ y\ .\ \mathbf{ret}(y))}}{(\boldsymbol{\lambda}\ x\ .\ \mathbf{ret}(x))\,(\boldsymbol{\lambda}\ y\ .\ \mathbf{ret}(y)) \Downarrow (\boldsymbol{\lambda}\ y\ .\ \mathbf{ret}(y))}$$

Esto nos deja con dos formas de evaluar términos, a pasos cortos (Definición 3.2.1), y la recién definida evaluación a grandes pasos. Sin embargo, se puede ver que la relación a grandes pasos coincide con la clausura transitiva reflexiva de la relación de pasos cortos.

Lema 3.2.1. *Sean* $M \in T$*,* $V \in \mathcal{V}$*, entonces tenemos que:* $M \Downarrow V \iff M \to^* \mathbf{ret}(V) \not\to$

Esto significa que hay *dos formas de evaluar términos* que relacionan los mismo términos con los mismos valores.

Términos sin valores

Lamentablemente, estas relaciones son parciales, es decir, hay términos cerrados dentro de nuestro lenguaje que *no tienen un valor asociado*. Veamos por ejemplo el conocido caso del término Ω.

Definición 3.2.3 (Término Omega)**.** *Definimos primero la expresión delta como* $\delta \doteq \boldsymbol{\lambda}\, x\, .\, x\, x$*. Observando su construcción sintáctica podemos determinar que* $\delta \in \mathcal{V}_0$*.*

Sea entonces el término cerrado Omega: $\Omega \doteq \delta\, \delta$*.*

Comenzamos, de forma muy optimista, a buscar el valor asociado a Ω tal cual nos indican las relaciones anteriores:

$$\begin{array}{rcll} \Omega & = & \delta\, \delta & (1) \\ & = & (\boldsymbol{\lambda}\, x\, .\, x\, x)\, \delta & (2) \\ & \to & x\, x[x := \delta] & (3) \\ & = & \delta\, \delta & (4) \end{array}$$

Observamos que partiendo de la definición de Ω en la ecuación (1), y dando un paso de evaluación de la relación de pasos cortos, llegamos al mismo término en la ecuación (4). Por lo que concluimos que el término cerrado Ω no tiene valor asociado por estas relaciones, en símbolos, $\Omega \notin \mathsf{dom}(\to^*), \mathsf{dom}(\Downarrow)$. Por lo tanto, no todos los términos (sintácticamente correctos, y siguiendo las convenciones establecidas) poseen un valor siguiendo las relaciones antes definidas, en particular no están caracterizados aquellos cuya *evaluación* no termina o diverge.

La observación de que la evaluación de un término diverja es de vital importancia para una de las definiciones de *equivalencia* de programas que veremos en el Capítulo 4, y por ende, estamos interesados en poder caracterizar la evaluación de dichos términos.

Además el problema de presentar un algoritmo para determinar si un término diverge o no, es un problema indecidible (Turing 1937). Pero sí es posible probar que la evaluación de ciertos términos diverge, como hicimos con el término Ω, y caracterizarlos otorgándoles un valor, carente de significado (más allá de notar a aquellos términos cuya evaluación diverge), notado usualmente como $\bot$.

Semántica Denotacional

La semántica denotacional se encarga de definir la semántica de los lenguajes de programación basado en un dominio semántico concreto. Los términos son mapeados en un dominio semántico que puede no ser el mismo lenguaje de programación, y en general, es utilizado un dominio de entidades puramente matemáticas, como ser, el espacio de funciones matemáticas. Dependiendo del rol que tenga la persona que defina el dominio semántico, diseñador de lenguaje, diseñador de compiladores, programadores, etc, las definiciones pueden variar para poder expresar diferentes propiedades del lenguaje. Sin embargo, se busca que la semántica sea: concisa, sin ambigüedades, operable matemáticamente, mecánicamente verificable, ejecutable y leíble dependiendo el punto de vista de cada uno (Allison 1987).

Dentro de este capítulo hemos visto la definición sintáctica del lenguaje y evaluadores siguiendo la noción de semántica operacional. En particular,

vimos una noción de evaluación basada en pasos, donde dado un término, este se *reescribe* múltiples veces de manera tal que ya no se pueda reescribir más alcanzando un valor. Este enfoque es muy restrictivo, ya que términos que *significan lo mismo* pero que se *escriben diferente* son considerados *distintos*.

En lo que queda de esta sección nos dedicaremos a definir lo que es para nosotros el *significado* de un término escrito en el lenguaje vehículo. La semántica denotacional recibe su nombre pensando en que cada programa o término *describe* o *denota* una función matemática entre el dominio de las posibles entradas y el codominio de las posibles salidas que el programa puede dar. La semántica denotacional entonces consiste en asignarle a cada término su denotación o función matemática.

La intuición nos indica que podemos pensar a los valores cerrados, en $\mathcal{V}_0$, como funciones matemáticas de valores en valores, y por ende, podemos concluir que un dominio podría ser el de las funciones de valores en valores $Func(\mathcal{V}_0, \mathcal{V}_0)$. Definamos entonces un mapeo de términos directamente a funciones matemáticas de un argumento:

$$\begin{aligned}
[\![-]\!] &: \Lambda_0 \to (\mathcal{V}_0 \to \mathcal{V}_0) \\
[\![\mathbf{ret}((\boldsymbol{\lambda}\ x\ .\ T))]\!] &\doteq y \mapsto T[x := y] \\
[\![(\boldsymbol{\lambda}\ x\ .\ T)\ W]\!] &\doteq [\![T[x := W]]\!] \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!] &\doteq (v \mapsto [\![N[x := v]]\!]) \circ [\![M]\!]
\end{aligned}$$

Pero las ecuaciones recién descriptas, lamentablemente, no tienen solución, en otras palabras no definen una función. Volvamos a explorar el término Ω definido en la sección anterior.

$$\begin{aligned}
[\![\Omega]\!] &= [\![\delta\ \delta]\!] \\
&= [\![(\boldsymbol{\lambda}\ x\ .\ x\ x)\ \delta]\!] \\
&= [\![(x\ x)[x := \delta]]\!] \\
&= [\![\delta\ \delta]\!]
\end{aligned}$$

En otras palabras, el mapeo que antes se intentó definir, no es un mapeo válido para todos los términos del lenguaje, ya que hemos encontrado un término, Ω que no tiene un valor asociado, en este caso, una función en $Func(\mathcal{V}_0, \mathcal{V}_0)$. En éste caso Ω caracteriza simplemente la noción de un término cuya evaluación no termina, y en el caso que se quiera definir un mapeo de términos a valores tendremos que o bien aceptar un mapeo *parcial* o una noción más amplia de valores que nos permita identificar aquellos términos cuya evaluación no termina o no tengan un valor.

En la siguiente sección, nos dedicamos a encontrar una noción de evaluación de aproximaciones que nos permita mapear términos en valores, y vamos a dar los fundamentos necesarios para que un mapeo similar al anterior tenga sentido.

3.3. Evaluación Aproximada

En esta sección, definimos una nueva relación de evaluación de términos utilizando una noción recursiva de evaluación. Utilizamos un enfoque de aproximación de valores, y de dominios semánticos, de forma tal que debemos extrapolar la noción de continuidad del dominio a las funciones de evaluación

y aproximación de términos. Esto se asemeja a una forma de evaluación más cerca de las conocidas como de *reescritura* de términos, ya que reescriben los términos hasta alcanzar un valor, donde la computación se produce a través de la substitución de variables por valores.

Como vimos en las secciones anteriores, hay términos que no tienen ningún valor asignado a ellos, como ser Ω, por lo tanto no vamos a poder definir una función (total). Esto lo podemos remediar introduciendo un nuevo elemento al conjunto de valores. Llamamos a este elemento adicional $\bot$ (*bottom* en palabras), denotando que un término no posee un valor utilizando la evaluación aproximada. Definimos a continuación una familia de relaciones indexada por los naturales, donde cada relación es una relación de evaluación a grandes pasos.

Definición 3.3.1 (Evaluación Aproximada de Términos)**.** *Definimos una familia de relaciones* $\Downarrow_n \subseteq \Lambda_0 \times (\mathcal{V}_0 + \bot)$ *con* $n \in \mathbb{N}$ *de la siguiente forma:*

$$\textit{lBot}\ \frac{}{M \Downarrow_0 \iota_r(\bot)} \qquad \textit{lRet}\ \frac{}{\mathbf{ret}(V) \Downarrow_{n+1} \iota_l(V)}$$

$$\textit{lApp}\ \frac{M[x := W] \Downarrow_n X}{(\boldsymbol{\lambda}\ x\ .\ M)\, W \Downarrow_{n+1} X} \qquad \textit{lLetD}\ \frac{M \Downarrow_n \iota_r(\bot)}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow_{n+1} \iota_r(\bot)}$$

$$\textit{lLetV}\ \frac{M \Downarrow_n \iota_l(V) \qquad N[x := V] \Downarrow_n a}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow_{n+1} a}$$

Donde $a \in \mathcal{V}_0 + \bot$.

Para cada número natural, $n \in \mathbb{N}$, podemos definir una relación entre términos cerrados y valores con un símbolo distinguido, $\bot$, donde relaciona a un término $M \in \Lambda_0$ con un valor $V \in \mathcal{V}_0$, si su árbol de derivación tiene profundidad n, o con $\bot$ si n es insuficiente.

Podemos caracterizar la relación ($\Downarrow_0$) como la relación que mapea todo término cerrado a $\bot$ por su definición.

Teorema 3.3.1 (Relación 0)**.** *La relación* $\Downarrow_0 = \{(M, \iota_r(\bot)) \mid M \in \Lambda_0\}$.

Una vez encontrado un valor para un término, todas las relaciones siguientes coinciden en éste valor.

Teorema 3.3.2 (Convergencia)**.** *Sea* $n \in \mathbb{N}$ *y* $M \in \Lambda_0$ *tal que existe* $V \in \mathcal{V}_0$ *tal que* $M \Downarrow_n \iota_l(V)$*, entonces para todo* $m \in \mathbb{N}, m > n$*,* $M \Downarrow_m \iota_l(V)$.

Demostración. Utilizaremos inducción en $n \in \mathbb{N}$ para probar la convergencia en las relaciones aproximadas de evaluación de términos.

El caso base con $n = 0$ se cumple por vacuidad. Por Teorema 3.3.1, la relación $\Downarrow_0 = \{(M, \iota_r(\bot)) \mid M \in \Lambda_0\}$, y por ende, se cumple por vacuidad que para todo $M \in \Lambda_0$ tal que existe $V \in \mathcal{V}_0, M \Downarrow_0 \iota_l(V)$, se cumple que para todo $m \in \mathbb{N}, m > 0, M \Downarrow_m \iota_l(V)$.

Sea $n \in \mathbb{N}, M \in \Lambda_0$, tenemos dos casos posibles. El caso donde n no sea suficiente como para alcanzar el valor de M, se resuelve nuevamente por vacuidad, y por ende, válido. Por lo que veamos el caso donde existe $V \in \mathcal{V}_0, M \Downarrow_n \iota_l(V)$. Sabemos por hipótesis inductiva que para todo $M' \in \Lambda_0$ tal que existe $V' \in \mathcal{V}_0$ con $M' \Downarrow_{n'} \iota_l(V')$ con $n' < n$ entonces $M' \Downarrow_{n'+1} \iota_l(V')$. Queremos entonces probar que $M \Downarrow_{n+1} \iota_l(V)$, y procederemos entonces haciendo un análisis por casos en la estructura de M.

- Sea $M \doteq \mathbf{ret}(V)$. Sabemos por la ley `lRet` que $M \Downarrow_{n+1} \iota_l(V)$.
- Sea $M \doteq W\,W'$ con $W, W' \in \Lambda_0$. Por $M \Downarrow_n \iota_l(V)$ existen $x \in \mathbb{V}, N \in \Lambda_0$ tales que $W \doteq \boldsymbol{\lambda}\ x\ .\ N$, y más aún, que $N[x := W] \Downarrow_{n-1} \iota_l(V)$. Por hipótesis inductiva concluimos que $N[x := W] \Downarrow_n \iota_l(V)$, y finalmente que $M \Downarrow_{n+1} \iota_l(V)$.
- Sea $M \doteq \mathbf{let}\ x = N\ \mathbf{in}\ O$, con $N \in \Lambda_0$ y $O \in \Lambda_x$. Como $M \Downarrow_n \iota_l(V)$ sabemos que existe $W \in \mathcal{V}_0$ tal que: $N \Downarrow_{n-1} \iota_l(W), O[x := W] \Downarrow_{n-1} \iota_l(V)$. Nuevamente por hipótesis inductiva sabemos que $N \Downarrow_n \iota_l(W), O[x := W] \Downarrow_n \iota_l(V)$, y por lo tanto concluimos que $M \Downarrow_{n+1} \iota_l(V)$.

De esta manera vemos que la familia de relaciones concuerdan en el valor asignado a un término una vez encontrado dicho valor. □

Por el teorema recién mostrado, concluimos que la familia de relaciones $\langle \Downarrow_n \rangle_{n<\omega}$ definen una noción de aproximación a la evaluación de un término. Las sucesivas aproximaciones de un término $M \in \Lambda_0$ definen una ω-cadena $\langle M \Downarrow_n \rangle_{n<\omega}$ siguiendo el orden discreto. Más aún, utilizando el teorema de convergencia podemos ver que las ω-cadenas definidas son *estacionarias* contemplando dos casos dependiendo si la evaluación de un término converge o no:

- si la evaluación de M no converge, la cadena es constante $\iota_r(\bot)$,
- mientras que si la evaluación converge, existe un valor $V \in \mathcal{V}_0$ y un $n \in \mathbb{N}$, tal que para todo $n' \in \mathbb{N}, n' \geq n, M \Downarrow_{n'} \iota_l(V)$.

Y de esta manera entonces podemos definir la evaluación de un término simplemente como el supremo de la cadena de aproximaciones:

$$[\![M]\!] = \sup_{n \in \mathbb{N}} \{M \Downarrow_n\}$$

Definición 3.3.2. *Sea $M \in \Lambda_0$ un término. Definimos la n-ésima aproximación $M^{(n)} \in (\mathcal{V}_0 + \bot)$ de M de la siguiente forma:*

$$\begin{array}{rcll} M^{(0)} & \doteq & \iota_r(\bot) & \\ (\mathbf{ret}(V))^{(n+1)} & \doteq & \iota_l(V) & \\ ((\boldsymbol{\lambda}\ x\ .\ M)\,V)^{(n+1)} & \doteq & (M[x := V])^{(n)} & \\ (\mathbf{let}\ x = M\ \mathbf{in}\ N)^{(n+1)} & \doteq & \begin{cases} \iota_r(\bot) & si\ (M)^{(n)} = \iota_r(\bot) \\ (N[x := V])^{(n)} & si\ (M)^{(n)} = \iota_l(V) \end{cases} & \end{array}$$

Probamos que la evaluación aproximada sigue la relación de aproximaciones n-th.

Lema 3.3.1. *Para todo término cerrado M, $M \Downarrow_n M^{(n)}$.*

Demostración. La prueba procede por inducción en n. Para el caso que $n = 0$, tenemos trivialmente que para todo $M \in \Lambda_0, M \Downarrow_0 \iota_r(\bot)$. Para el caso inductivo $n = m + 1$ con $m \geq 0$. Procedemos entonces a hacer análisis por casos en las posibles reglas aplicadas en la derivación de $M \Downarrow_{m+1} M^{(m+1)}$.

- Si la regla aplicada fue (`lRet`) tenemos que $M = \mathbf{ret}(V)$ con $V \in \Lambda_0$, y por definición tenemos que $(\mathbf{ret}(V))^{(m+1)} = \iota_l(V)$, y por definición, $M \Downarrow_{m+1} \iota_l(V)$.

- Si la regla aplicada fue (`lApp`) tenemos que $M = (\boldsymbol{\lambda}\ x\ .\ N)\, W$ con $N \in \Lambda_x, W \in \mathcal{V}_0$, y que $N[x := W] \Downarrow_m X$ con $X \in \mathcal{V}_{0\perp}$ y $M \Downarrow_{m+1} X$. Por otro lado tenemos que $M^{(m+1)} = (\boldsymbol{\lambda}\ x\ .\ N)\, W^{(m+1)} = (N[x := W])^{(m)}$. Por hipótesis inductiva tenemos que $N[x := W] \Downarrow_m (M[x := W])^{(m)}$, por lo que concluimos $M \Downarrow_{m+1} (M[x := W])^{(m)} = M^{(m+1)}$.

- Si la regla aplicada fue (`lLetD`) tenemos que $M = \mathbf{let}\ x = P\ \mathbf{in}\ Q \Downarrow_{m+1} \iota_r(\perp)$ y que $P \Downarrow_m \iota_r(\perp)$. Por otra lado tenemos que $(\mathbf{let}\ x = P\ \mathbf{in}\ Q)^{(m+1)}$ depende de la evaluación de P, pero por hipótesis inductiva sabemos que $P^{(m)} = \iota_r(\perp)$, por lo que $M^{(m+1)} = \iota_r(\perp)$.

- Si la regla aplicada fue (`lLetV`) tenemos que $M = \mathbf{let}\ x = P\ \mathbf{in}\ Q \Downarrow_{m+1} \iota_l(W)$ con $W \in \Lambda_0$, $P \Downarrow_m V$ y $Q[x := V] \Downarrow_m \iota_l(W)$ con $W \in \mathcal{V}_0$. Por el otro lado tenemos que $M^{(m+1)} = (\mathbf{let}\ x = P\ \mathbf{in}\ Q)^{(m+1)}$ depende de $(P)^{(m)}$, pero por hipótesis inductiva sabemos que $P^{(m)}V$, y por lo tanto $M^{(m+1)} = (Q[x := V])^{(m)}$. Y finalmente, por hipótesis inductiva tenemos que $(Q[x := V])^{(m)}\iota_l(W)$, por lo que $(M)^{(m+1)} = \iota_l(W)$.

Por lo que concluimos que para todo término $M \in \Lambda_0, n < \omega, M \Downarrow_n M^{(n)}$ □

Por los teoremas recién mostrados, podemos definir la semántica operacional de forma ecuacional.

Lema 3.3.2. *La relación de evaluación respeta las siguientes ecuaciones:*

$$\begin{aligned}
[\![\mathbf{ret}(V)]\!] &= \iota_l(V) \\
[\![(\boldsymbol{\lambda}\ x\ .\ M)\, W]\!] &= [\![M[x := W]]\!] \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!] &= \begin{cases} \iota_r(\perp) & si\ [\![M]\!] = \iota_r(\perp) \\ [\![N[x := V]]\!] & si\ [\![M]\!] = \iota_l(V) \end{cases}
\end{aligned}$$

Demostración. La prueba sigue de hacer análisis por casos en la estructura del término M.

- Sea $M = \mathbf{ret}(V)$ con $V \in \mathcal{V}_0$. Por definición la evaluación del término es el supremo de su cadena de aproximaciones

$$[\![\mathbf{ret}(V)]\!] = \sup_{n\in\mathbb{N}}\{\mathbf{ret}(V) \Downarrow_n\}$$

Por Lema 3.3.1, y V ser un valor cerrado,

$$\sup_{n\in\mathbb{N}}\{\mathbf{ret}(V) \Downarrow_n\} = \sup_{n\in\mathbb{N}}\{(\mathbf{ret}(V))^{(n)}\}$$

Por convergencia y propiedades de cadenas,

$$\sup_{n\in\mathbb{N}}\{(\mathbf{ret}(V))^{(n)}\} = \sup_{n\in\mathbb{N}}\{(\mathbf{ret}(V))^{(n+1)}\}$$

Aplicando la definición de aproximación nos queda que

$$\sup_{n\in\mathbb{N}}\{(\mathbf{ret}(V))^{(n+1)}\} = \sup_{n\in\mathbb{N}}\{\iota_l(V)\} = \iota_l(V)$$

Concluimos que $[\![\mathbf{ret}(V)]\!] = \iota_l(V)$.

- Sea $M = (\boldsymbol{\lambda}\ x\ .\ N)\, W$ para $N \in \Lambda_{\{x\}}$ y $W \in \mathcal{V}_0$. Análogo al caso anterior, utilizamos la relación de n-th aproximación para caracterizar la evaluación de M haciendo un corrimiento en la cadena.

- Sea $M = \mathbf{let}\ x = N\ \mathbf{in}\ P$ con $N \in \Lambda_0$ y $P \in \Lambda_{\{x\}}$. Tenemos dos casos adicionales dependiendo en la terminación de la evaluación del término cerrado N.

  - Supongamos que la evaluación de N no converge a un valor, $[\![N]\!] = \iota_r(\bot)$. En símbolos tenemos que la cadena de aproximaciones es siempre $\iota_l(\bot)$, $\forall n \in \mathbb{N}, N \Downarrow_n \iota_r(\bot)$. Por definición de n-th aproximación, $(\mathbf{let}\ x = N\ \mathbf{in}\ P)^{(n+1)} = \iota_r(\bot)$ para todo $n \in \mathbb{N}$, y entonces, $[\![M]\!] = \iota_r(\bot)$.
  - Supongamos que la evaluación de N converge a un valor cerrado $V \in \mathcal{V}_0$. Por definición de la evaluación tenemos que existe $m \in \mathbb{N}$, para todo $n \in \mathbb{N}$ tal que $n > m$, $N \Downarrow_n \iota_l(V)$. Por definición de n-th aproximación, $(\mathbf{let}\ x = N\ \mathbf{in}\ P)^{(n+1)} = (N[x := V])^{(n)}$ para todo $n > m$, y entonces, $[\![M]\!] = [\![N[x := V]]\!]$.

□

Veamos entonces que si bien al término Ω no le podemos asignar un valor, en este caso podemos asignarle el elemento $\bot$. En las secciones anteriores vimos que la evaluación de Ω generaba una dependencia en sí misma, es decir, que la evaluación de Ω lleva a la evaluación de Ω, lo que produce que la evaluación del término no termine. En este caso, utilizando la evaluación aproximada, podemos mostrar que dado un $n \in \mathbb{N}$ nunca nos alcanzará para evaluar Ω, y por lo tanto, tendremos que $\Omega \Downarrow_n \iota_r(\bot)$.

Lema 3.3.3. *La evaluación de Ω diverge, es decir, $[\![\Omega]\!] = \iota_r(\bot)$.*

Demostración. Podemos verlo mostrando que la ω-cadena que genera es constante, en palabras, para todo $n \in \mathbb{N}, \Omega \Downarrow_n \iota_r(\bot)$. Anteriormente vimos que la evaluación del término Ω genera un ciclo en la evaluación, en este caso como tenemos una familia de evaluaciones, vemos que si bien no genera un ciclo, ya que técnicamente estamos utilizando otra relación dentro de la familia de relaciones, cualquier $n \in \mathbb{N}$ que utilicemos no nos alcanzará para asignarle un valor.

Sea $n \in \mathbb{N}$,

$$\texttt{lApp}\frac{\Omega = x\, x[x := \delta] \Downarrow_n V}{\Omega = (\boldsymbol{\lambda}\ x\ .\ (x\, x))\, \delta \Downarrow_{n+1} V}$$

Es decir, en un paso la evaluación de Ω vuelve a depender de la evaluación de Ω. En toda relación de evaluación $n \in \mathbb{N}$ es finito, por lo que la evaluación de Ω alcanzará inevitablemente la relación $(\Downarrow_0)$, y por lo tanto, $\Omega \Downarrow_n \iota_r(\bot)$. □

Sin importar la profundidad del árbol de derivación que utilicemos, siempre nos va a faltar expandir el árbol aún más.

3.4. Conclusiones del Capítulo

En este capítulo definimos el lenguaje básico que utilizamos a lo largo de la tesis, y repasamos diferentes formas de evaluar los términos del mismo. Definimos dos grandes categorías de otorgarle significado a los términos, la semántica operacional y denotacional, y a su vez, vimos una evaluación que nos permite observar cuando la evaluación de términos diverge.

No siempre está claro que tipo de semántica es la más adecuada. Dependiendo de las características del lenguaje y de las observaciones que quisiéramos realizar sobre la evaluación podemos elegir una otra forma. En el caso que se busque relacionar la evaluación del lenguaje con las instrucciones del computador generadas, una semántica operacional suele ser mejor. Mientras que si lo que se busca es un análisis composicional, las semánticas denotacionales son las adecuadas.

Desde el punto de análisis de costos de programas, en general se utilizan semánticas operacionales ya que estas están más cercas a las instrucciones generadas y cómo la evaluación es realizada, aunque es posible aumentar las semánticas denotacionales para que, además de denotar el valor de un término, sea posible discernir propiedades de la evaluación (Cicek et al. 2017; Van Stone 2003). En nuestro caso, utilizamos la evaluación aproximada donde nos permite definir una semántica operacional derivada de las ω-cadenas. A su vez, introducimos propiedades inspirándonos en propiedades de evaluaciones propias de semánticas denotacionales.

Finalmente, podríamos haber definido el lenguaje siguiendo una sintaxis estándar sin una separación sintáctica entre términos y valores. De todas maneras, además de atenernos a la bibliografía utilizada, veremos que dicha separación será muy útil al momento de introducir efectos en la evaluación de términos.

Capítulo 4

Equivalencia entre Programas

La equivalencia entre programas es uno de los problemas más estudiados dentro del área de ciencias de la computación. Se trata de resolver cuando dos programas son *iguales* bajo alguna definición de *equivalencia* (Lahiri et al. 2018). Por ejemplo, una noción de equivalencia es la de extensionalidad, o equivalencia *input-output* de programas, donde se espera que a entradas iguales los programas computen resultados iguales. Una noción así ignora cuestiones como la ejecución de los programas. Bajo esta visión dos programas que ante las mismas entradas retornen los mismo resultados serían equivalentes, independientemente que uno lo haga instantáneamente y otro tenga que estar ejecutando durante años.

Existen diferentes formas de comparar programas. Una forma diferente a la antes mencionada, consiste en otorgarles una interpretación o definición en otro dominio y comparar los programas en base a su interpretación. Este enfoque es útil ya que, al momento de interpretar programas, podemos abstraer propiedades que no sean necesarias para identificar programas equivalentes. Un ejemplo es la alfa-conversión (α-conversión) de términos en el cálculo lambda, donde identificamos términos que simplemente utilizan diferentes nombres de variables para ligar argumentos en abstracciones pero que son iguales en todo lo demás. Otra forma de comparar programas puede incluso tener en cuenta información sobre ejecución o interpretación de programas. En otras palabras, el concepto de programa adquiere entonces la propiedad de ejecución al momento de compararlo con otro. Estudiar el significado de los programas es el estudio de la semántica de los programas.

Dentro de los enfoques se encuentran a grandes rasgos, tres ramas de estudio, la *semántica denotacional* (Sección 3.2), la *semántica axiomática* (Floyd 1993; Hoare 1969), y la *semántica operacional* (Sección 3.2). La semántica denotacional presentada originalmente por Dana Scott (Scott 1982) conlleva un gran desarrollo matemático dentro de la teoría de dominios que nos otorga herramientas para trabajar particularmente con programas *recursivos*, y consiste en definir los programas como elementos de cierto dominio matemático. La semántica axiomática se basa en darle sentido a los programas por las propiedades que ellos cumplen. Mientras que la semántica operacional se basa en definir el comportamiento de los programas mediante una relación de ejecución (Pitts 2002).

Resumimos estos enfoques de la siguiente manera:

Semántica Axiomática define a los programas como las propiedades que

estos cumplen.

Semántica Denotacional define a los programas como su interpretación dentro de un dominio semántico, es decir, como su denotación.

Semántica Operacional define a los programas como su ejecución en una máquina abstracta o una relación de evaluación.

Semántica Axiomática Dentro de la semántica axiomática, la equivalencia de programas se define como la equivalencia entre las propiedades que estos cumplen. El mayor ejemplo de semántica axiomática es la lógica de Hoare (Hoare 1969), donde a un programa $C \in Prog$ se lo nota de la siguiente manera con predicados P, Q:

$$\{P\}\, C\, \{Q\}$$

Dicha tripleta de elementos le otorga una definición semántica a C como el programa que dada la pre-condición P, luego de la ejecución de C, se alcanza una post-condición Q.

Al definir la semántica de los programas como las propiedades que estos cumplen, en particular, la relación entre el pre-estado y el post-estado, podemos entonces definir que dos programas son equivalentes si cumplen las mismas propiedades.

Definición 4.0.1 (Equivalencia de Hoare)**.** *Sean A, B dos programas. Decimos que A y B son equivalentes si para toda pre-condición P y post-condición Q se cumple que:*

$$\{P\}\, A\, \{Q\} \iff \{P\}\, B\, \{Q\}$$

Este enfoque se basa en probar propiedades de los programas, pero no de su ejecución. Sin embargo, es posible aumentar el poder expresivo de las tripletas de Hoare e introducir relaciones entre programas, presentando un modelo Relacional de la lógica de Hoare (Benton 2004). Esto permite expresar relaciones de equivalencia dentro del mismo marco axiomático, lo que permite el desarrollo de técnicas más avanzadas para establecer relaciones entre los costos de programas (Cicek et al. 2017). De todas maneras, y a diferencia de la semántica operacional, el costo de los programas y operaciones se asignan mediante una codificación o interpretación de los mismos en el modelo axiomático. En otras palabras, no es un modelo emergente de la misma evaluación, sino una interpretación del observador sobre la evaluación del lenguaje.

Semántica Denotacional La semántica denotacional de los programas se encarga de identificar programas con elementos dentro de alguna estructura o teoría matemática. Este enfoque permite, una vez mapeados los elementos, utilizar *todo el poder de la matemática* u otras posibles técnicas dependiendo de la teoría matemática utilizada. En este caso la equivalencia entre programas se traduce en equivalencia entre las denotaciones que se les asigna a cada uno de los programas.

De nuevo tenemos el mismo problema que con la semántica axiomática, la propiedades de la evaluación de términos o ejecución de programas son introducidas un tanto artificialmente ya que no son propiedades emergentes de la misma, no hay noción de ejecución o evaluación, sino que se pueden llegar a introducir dentro del modelo de forma axiomática (Van Stone 2003).

Semántica Operacional La semántica operacional le otorga significado a los programas mediante la observación de su evaluación o ejecución. Al poder observar la evaluación de los programas resulta intuitivo relacionar aspectos de la misma, como el costo necesario para alcanzar un valor. Aunque, lamentablemente, también hace que el enfoque pueda llegar a ser demasiado concreto, si lo pensamos desde el punto de vista de los programas, ahora estos dependen de su evaluación, y esta, de una máquina *concreta*.

Las nociones semánticas antes nombradas son simplemente las más clásicas del área pero no son las únicas. Para dar un ejemplo, también se encuentra la semántica basada en juegos (*game semantics*), donde los programas se interpretan como las estrategias de un juego entre los diferentes agentes del sistema (Abramsky y McCusker 1999).

Cabe destacar que todos los enfoques utilizan técnicas de los demás e incluso intentan describir propiedades que parecen ser exclusivas de algún modelo particular. Un ejemplo claro es la noción de costo de evaluación de un programa, siendo esta una propiedad intensiva y emergente de la evaluación o ejecución de un programa.

En esta tesis nos concentramos en utilizar conceptos de semántica operacional incorporando además prácticas y herramientas de semántica denotacional. Nuestro objetivo es el estudio de propiedades intensivas resultantes de la ejecución de los programas, por lo que utilizaremos nociones de semánticas operacionales, pero a su vez también vamos a identificar programas con elementos dentro de un dominio semántico que sea capaz de interpretar los efectos del lenguaje. Por ejemplo, al momento de estudiar programas probabilísticos identificaremos a los programas como una sub-distribución de probabilidades que computan valores con cierta probabilidad.

La noción de equivalencia entre programas es también relevante no solo para áreas teóricas, como las nombradas en los párrafos anteriores, sino también para áreas relacionadas con la práctica de programación o con el área de compiladores. Estas áreas utilizan la noción de reescritura de programas, ya sea con el objetivo de simplificar la depuración de programas, la generación de casos de pruebas, o incluso con el énfasis de *optimizar programas*.

En lo que resta del capítulo exploramos diferentes definiciones de equivalencias entre programas, clásicas dentro del área de semántica operacional, que utilizamos a lo largo de la tesis. Para profundizar en este tema recomendamos al lector consultar el artículo presentado por Andrew Pitts (2002).

4.1. Equivalencia Contextual

Intuitivamente, dos términos M y M' son *contextualmente* equivalentes si cualquier ocurrencia de M puede intercambiarse por la de M' sin afectar el resultado final. Para poder dar una definición formal de dicho concepto intuitivo, deberemos primero formalizar dos nociones:

- cómo los programas son *evaluados* y así poder hablar del *resultado final*,
- cuales son las observaciones que se realizan sobre el resultado de las evaluaciones, es decir, definir que significa que el resultado final no ha sido afectado.

El primero podemos resolverlo definiendo la semántica operacional del lenguaje, o en concreto mediante una relación de evaluación, mientras que el segundo,

incluye definir qué propiedades queremos observar de la evaluación de los mismos. Esta tesis se basa en explorar cómo observar propiedades de la evaluación de programas, sus efectos, y dentro de ellos, cómo poder incorporar la noción de mejora mediante la adición del costo de la evaluación.

Para fundamentar la noción de equivalencia contextual es natural introducir primero la noción intuitiva, aplicable a diferentes ámbitos de la ciencia, de *comportamiento observacional* o el concepto de *identidad de los indiscernibles* (Leibniz 1989).

Definición 4.1.1 (Identidad de los Indiscernibles)**.** *La identidad de los indiscernibles define que dos elementos son equivalentes si y solo si ante todo juicio lo son.*

En símbolos tenemos que dados x, y, $x \equiv y$, si y solo si, para todo predicado P, $P(x) \iff P(y)$.

Este mismo concepto se esparce por el mundo de la ciencia tomando diferentes formas aunque todavía, en la ciencias experimentales, sigue siendo un foco de discusión (Forrest 2020).

Volviendo a nuestro tópico, vamos a juzgar y a realizar inferencias al *observar* el *comportamiento* de la evaluación de términos. A diferencia de los conceptos puramente matemáticos, la computación incluye el mecanismo por el cual los términos son evaluados, por lo que vamos a querer *observar* cómo se computa el resultado de la evaluación y no simplemente la igualdad en la veracidad de un juicio. En otras palabras debemos ser precisos al momento de definir los predicados que utilizaremos al momento de instanciar la definición de la identidad de los indiscernibles.

Contextos

Definimos la noción de contexto como un término del lenguaje con la posible presencia de **un** *agujero*, aunque éste puede aparecer *múltiples veces*. Conectando con la Definición 4.1.1, los contextos reemplazan la construcción de los predicados, aunque como resultado tendremos programas sin evaluar, y no fórmulas que son verdaderas o falsas, lo que indica que nuestro trabajo todavía no ha terminado.

Definición 4.1.2 (Contexto)**.** *Formalmente definimos a los contextos del lenguaje como dos categorías sintácticas, contextos valores y contextos términos, de la siguiente forma:*

$$\begin{array}{ll} \mathbb{V}, \mathbb{W} & ::= x \mid (\boldsymbol{\lambda}\ x\ .\ \mathbb{C}) \\ \mathbb{C}, \mathbb{D} & ::= [-] \mid \mathbf{ret}(\mathbb{V}) \mid (\mathbb{V}\,\mathbb{W}) \mid \mathbf{let}\ x = \mathbb{C}\ \mathbf{in}\ \mathbb{D} \end{array}$$

Lo que define dos categorías de contextos:

Contextos Valores : contextos en los que el agujero se sitúa en la posición de un término pero que al ser rellenado define un valor

Contextos Términos : contextos en el que el agujero se sitúa en la posición de un término pero que al ser rellenado define un término.

En lo que sigue de la tesis, a menos que sea necesaria la aclaración llamaremos a ambos tipos de contextos simplemente contextos.

Representamos al agujero con el símbolo $[-]$, a su vez todos los términos son, simplemente, contextos término sin la ocurrencia del agujero $[-]$. Esta definición nos permite tener contextos que tengan un solo tipo de agujero, aunque éste pueda tener múltiples apariciones.

Definición 4.1.3 (Llenado de Agujeros)**.** *Sea $\mathbb{C}$ un contexto término, y $M \in \Lambda$ un término. Definimos la operación de llenado de contexto $\mathbb{C}[M] \in \Lambda$ como el término resultante de reemplazar todas las apariciones del agujero $[-]$ por el término M.*

Sea $\mathbb{V}$ un contexto valor, y $M \in \Lambda$ un término. Definimos la operación de llenado de contexto $\mathbb{V}[M] \in \mathcal{V}$ como el valor resultante de reemplazar todas las apariciones del agujero $[-]$ por el término M.

En otras palabras, el llenado de agujeros es una substitución donde se permite la captura de variables libres. Más aún, formalizamos la idea de que las variables libres de los términos son establecidas por el entorno de un término (el contexto) y no por el término en sí. Esto establece una gran diferencia con la substitución estándar del cálculo lambda utilizada en la β-reducción donde se utiliza una substitución que no capture variables libres manteniendo la α equivalencia entre términos. Adicionalmente utilizamos el mismo operador para la composición de contextos, sean $\mathbb{C}, \mathbb{D}$ dos contextos, y t un término: $(\mathbb{C}[\mathbb{D}])[t] = \mathbb{C}[\mathbb{D}[t]]$

Definimos la substitución del símbolo $[-]$ por un término de la siguiente forma:

$$\begin{array}{rcl} x[M] & \doteq & x \\ (\boldsymbol{\lambda}\ x\ .\ \mathbb{C})[M] & \doteq & \boldsymbol{\lambda}\ x\ .\ (\mathbb{C}[M]) \end{array}$$

$$\begin{array}{rcl} [-][M] & \doteq & M \\ \mathbf{ret}(\mathbb{V})[M] & \doteq & \mathbf{ret}((\mathbb{V}[M])) \\ (\mathbb{V}\,\mathbb{W})[M] & \doteq & (\mathbb{V}[M])\,(\mathbb{W}[M]) \\ (\mathbf{let}\ x = \mathbb{C}\ \mathbf{in}\ \mathbb{D})[M] & \doteq & \mathbf{let}\ x = (\mathbb{C}[M])\ \mathbf{in}\ (\mathbb{D}[M]) \end{array}$$

Podemos ver con un ejemplo que el llenado de agujeros no preserva la relación de α-equivalencia entre términos. Es decir, que la noción de α-equivalencia entre contextos no se traduce directamente de la α-conversión de términos y valores dentro del lambda cálculo.

Ejemplo 4.1.1. *Sean $x, y \in \mathbb{V}$ dos variables distintas, podemos entonces pensar en definir dos contextos valor α-equivalentes como $\boldsymbol{\lambda}\ x\ .\ [-] \equiv_\alpha^{\mathcal{V}} \boldsymbol{\lambda}\ y\ .\ [-]$, pero si llenamos el agujero con el mismo término obtenemos dos valores que no son α-equivalentes:*

$$(\boldsymbol{\lambda}\ x\ .\ [-])[\mathbf{ret}(x)] = \boldsymbol{\lambda}\ x\ .\ \mathbf{ret}(x) \not\equiv_\alpha^{\mathcal{V}} \boldsymbol{\lambda}\ x\ .\ \mathbf{ret}(y) = (\boldsymbol{\lambda}\ y\ .\ [-][])\mathbf{ret}(x)$$

En otras palabras, la operación de llenado de contextos puede romper la relación de α-equivalencia, y por ende, hay que ser cuidadosos al momento de utilizarla. Por suerte no todo está perdido, y podemos ver que la composición de contextos sí respeta α-equivalencia, ya que es fácil probar que llenar contextos con contextos α-equivalentes resultan en contextos α-equivalentes, sean $\mathbb{C}_1, \mathbb{C}_2, \mathbb{C}$ contextos:

$$\mathbb{C}_1 \equiv_\alpha \mathbb{C}_2 \implies \mathbb{C}[\mathbb{C}_1] \equiv_\alpha \mathbb{C}[\mathbb{C}_2]$$

De esta manera podemos mantener la relación de α-equivalencia en estos contextos. Adicionalmente, es posible formalizar estos conceptos utilizando el concepto de *function variables* y *substitution of meta-abstractions for function variables* (Pitts 1994), donde se definen nuevas variables para los agujeros y una substitución de estas variables por términos anotando las variables que están ligadas.

Podemos incluso utilizar la noción de álgebra libre (Definición 2.2.6) para definir los contextos y el operador de llenado de agujeros. En otras palabras, los contextos son simplemente árboles de sintaxis abstracta que se pueden extender mediante la operación de llenado de agujeros, donde podemos utilizar el conjunto unitario para definir un único agujero.

Aproximación Observacional

Tomando la relación de evaluación de términos definida en la Sección 3.3, tenemos las herramientas necesarias para definir la aproximación observacional. Continuando con la definición de la identidad de los indiscernibles, lo que nos queda entonces por establecer es qué esperamos observar de la evaluación de los términos. Debido a que los programas son definidos como términos cerrados, los valores resultantes de la evaluación (en caso de que lo haya) serán cerrados; en otras palabras, serán abstracciones (Definición 3.1.1). Lo que observamos es que la evaluación de un término alcance un valor, o en otras palabras, que converja a un valor.

Para simplificar la escritura definimos la siguiente notación.

Notación 4.1.1. *Notamos utilizando directamente el operador de la relación de evaluación cuando un programa converge. Sea $M \in \Lambda_0$ decimos que M converge, $M \Downarrow$, si existe $W \in \mathcal{V}_0$ y $n \in \mathbb{N}$ tal que: $M \Downarrow_n \iota_l(W)$. En símbolos:*

$$M \Downarrow \doteq \exists W \in \mathcal{V}_0, n \in \mathbb{N}, M \Downarrow_n \iota_l(W)$$

De esta manera, utilizando contextos y la convergencia de la evaluación, podemos definir completamente la aproximación observacional de términos. Volviendo a la definición intuitiva dada al comienzo de la sección: dos términos son equivalentes, si y solo si, estos pueden ser intercambiados en todo contexto sin *observar ninguna diferencia*. En otras palabras, la noción de comportamiento que se observa es la de terminación conocida como la equivalencia contextual de *Morris* (Plotkin 1977).

Definición 4.1.4 (Aproximación Observacional)**.** *Sean M, N dos términos. Decimos que M aproxima observacionalmente a N si y solo si para todo contexto $\mathbb{C}$ que cierra a los términos M y N, $\mathbb{C}[M], \mathbb{C}[N] \in \Lambda_0$, si $\mathbb{C}[M] \Downarrow$, entonces $\mathbb{C}[N] \Downarrow$.*

Cuando un término M aproxima observacionalmente a otro N lo notamos como $M \sqsubseteq^T N$. De la misma forma podemos definir cuando un valor aproxima a otro: mientras que cuando un valor V aproxima observacionalmente a otro W lo notamos como $V \sqsubseteq^V W$. Utilizaremos el símbolo ($\sqsubseteq$) para ambas relaciones aclarando a cuál nos referimos en caso que sea necesario.

En los ejemplos que se muestran a continuación asumimos que contamos con una implementación de números naturales y sus operadores dentro del lenguaje

más un operador ternario $\mathbf{ifz}(M, N, S)$ cuya evaluación equivale a la evaluación de N si M evalúa a 0, o a la evaluación de S si M evalúa a un número distinto de 0.

La relación de aproximación observacional no es simétrica, ya que podría ser el caso que M esté menos definido que N. Por ejemplo, tomemos el siguiente término:

$$\texttt{Fact} = \boldsymbol{\lambda}\ f\ .\ \mathbf{ret}(\boldsymbol{\lambda}\ n\ .\ \mathbf{ifz}(n, 1, (\mathbf{let}\ x = \mathbf{pred}\, n\ \mathbf{in}\ (\mathbf{let}\ r = f\, x\ \mathbf{in}\ n * r))))$$

Intuitivamente, tenemos que:

$$\texttt{Fact}(\boldsymbol{\lambda}\ x\ .\ \Omega) \sqsubseteq \texttt{Fact}^2(\boldsymbol{\lambda}\ x\ .\ \Omega) \sqsubseteq \ldots$$
$$\ldots \sqsubseteq \texttt{Fact}^n(\boldsymbol{\lambda}\ x\ .\ \Omega) \sqsubseteq \texttt{Fact}^{n+1}(\boldsymbol{\lambda}\ x\ .\ \Omega) \sqsubseteq \ldots \sqsubseteq \mathbf{Z}\,\texttt{Fact}$$

Donde $\mathbf{Z}$ es el operador de punto fijo del cálculo lambda, definido a continuación:

$$\mathbf{Z} = \boldsymbol{\lambda}\ f\ .\ (\boldsymbol{\lambda}\ x\ .\ f\,(\boldsymbol{\lambda}\ y\ .\ \mathbf{let}\ z = x\,x\ \mathbf{in}\ z\,y))(\boldsymbol{\lambda}\ x\ .\ f\,(\boldsymbol{\lambda}\ y\ .\ \mathbf{let}\ z = x\,x\ \mathbf{in}\ z\,y))$$

Utilizamos el operador $\mathbf{Z}$ en vez del más utilizado operador de recursión $\mathbf{Y}$ ya que la evaluación de términos es eager.

Finalmente definimos la equivalencia observacional entre términos simplemente como la conjunción de que un término aproxime al otro, y vice versa.

Definición 4.1.5 (Equivalencia Observacional)**.** *Dados dos términos $M, N \in \Lambda$ decimos que son observacionalmente equivalentes:*

$$M \approx N \iff M \sqsubseteq N \wedge N \sqsubseteq M$$

Lamentablemente, una definición tan intuitiva deja de lado muchas sutilezas. La mayor complicación es la cuantificación universal sobre los contextos, donde por ejemplo, se tienen en cuenta contextos que no poseen agujeros, y por ende, la observación es la misma. Adicionalmente, las relaciones observacionales, al cuantificar universalmente sobre los contextos que observan las expresiones, hacen que probar que un término aproxima a otro sea difícil, mientras que mostrar que **no es así** es más sencillo. Necesitamos simplemente presentar un contexto que los diferencie. Esto podemos verlo con un ejemplo.

Ejemplo 4.1.2. *Con el fin de proveer un ejemplo intuitivo, primero definimos algunas construcciones útiles. Suponemos que tenemos disponible una codificación de booleanos. Es decir, los valores* **true**, **false** *representando verdadero y falso respectivamente, más un operador ternario $(V\ ?\ M : N)$ de forma tal que la evaluación del operador se comporta como M si V es* **true** *o como N si V es* **false**.

$$\begin{aligned}
\underline{0} &\doteq \boldsymbol{\lambda}\ f\ .\ \mathbf{ret}(\boldsymbol{\lambda}\ z\ .\ \mathbf{ret}(z)) \\
\mathit{succ} &\doteq \boldsymbol{\lambda}\ n\ .\ \mathbf{ret} \\
&\quad\ \boldsymbol{\lambda}\ f\ .\ \mathbf{ret} \\
&\quad\ \boldsymbol{\lambda}\ z\ .\ \mathbf{ret}(\mathbf{let}\ nF = n\,f\ \mathbf{in}\ \mathbf{let}\ nFZ = nF\,z\ \mathbf{in}\ f\,nFZ) \\
\mathit{isZero} &\doteq \boldsymbol{\lambda}\ n\ .\ \mathbf{let}\ nF = n\,(\boldsymbol{\lambda}\ q\ .\ \mathbf{false})\ \mathbf{in}\ nF\,\mathbf{true}
\end{aligned}$$

Podríamos suponer felizmente que el sucesor de cualquier número no puede ser cero, y escribirlo de la siguiente forma:

$$\mathbf{let}\ sx = (\mathit{succ}\,x)\ \mathbf{in}\ (\mathit{isZero}\,sx) \approx \mathbf{false}$$

Pero no es el caso, ya que tenemos un contexto que observa un comportamiento diferente entre los términos:

$$\mathbb{C} \doteq \textbf{let } x = \textbf{ret}(\boldsymbol{\lambda}\ a\ .\ \textbf{ret}(\boldsymbol{\lambda}\ b\ .\ \textbf{let } az = a\,\underline{0} \textbf{ in } (az\,?\,\textbf{ret}(b) : \Omega))) \textbf{ in } [-]$$

El contexto $\mathbb{C}$ fue fabricado a propósito para mostrar que la evaluación de un término converge mientras que la del otro diverge, y así mostrar que no son observacionalmente equivalentes. Esto lo podemos hacer dado que la evaluación de términos fuerza la evaluación del término `succ` x*. El ejemplo puede resultar complicado y artificioso, ya que depende directamente de cómo es la evaluación de términos, en nuestro caso, la semántica operacional. La derivación la dejaremos a cargo del lector, pero vamos a observar rápidamente que sucede. Si llenamos el agujero con el valor* **false** *el resultado de la evaluación es directamente el valor* **false**.

$$[\![\mathbb{C}[\textbf{false}]]\!] = \iota_l(\textbf{false})$$

Mientras que si llenamos el agujero con el término que valida que el sucesor de la variable x no es cero, llegaremos a que es equivalente a la convergencia de Ω.

$$[\![\mathbb{C}[\textbf{let } sx = \texttt{succ}\,x \textbf{ in } \texttt{isZero}\,sx]]\!] \equiv [\![\Omega]\!]$$

Esto se debe a que la variable x está libre en el término, y por ende, esta puede ser manipulada por el contexto. Este tipo de problemas se pueden evitar introduciendo un sistema de tipos que fuerce a que el contexto solo pueda instanciar x con valores naturales definidos dentro del ejemplo.

En el caso de que podamos tener computaciones siguiendo la estrategia de evaluación de CbN, el contexto que utilizaremos es distinto, y en este caso, más sencillo. Ya que el operador `succ` es estricto, dentro del modelo de CbN, podemos hacer que la evaluación diverja con un contexto que directamente defina a la variable libre x como Ω, como por ejemplo el siguiente contexto.

$$\mathbb{C} \doteq \textbf{let } x = \Omega \textbf{ in } [-]$$

En realidad lo que queremos escribir en la equivalencia del Ejemplo 4.1.2 es que si x es un término que converge a la representación un número natural, entonces sí deberíamos poder mostrar la equivalencia.

Este razonamiento podemos aplicarlo al lenguaje de programación *Haskell*. El lenguaje Haskell presenta un término que representa el valor *indefinido*, aunque no diverge su evaluación sino que termina abruptamente la ejecución del programa. Haskell es un lenguaje de programación perezoso por lo que sigue una estrategia de evaluación CbN, y define el término $undefined :: \forall a.a$, que se comporta como el término Ω. En otras palabras, es un símbolo especial que al ser evaluado se aborta la ejecución, podemos detectar la divergencia.

De todas maneras, esto establece que la evaluación de los programas impacta levemente en las estrategias a utilizar para distinguir términos. En una evaluación CbN, utilizamos variables libres y operadores estrictos para discernir entre valores o términos cuya evaluación no converge. Mientras que en el caso de evaluadores CbV los contextos solo pueden ligar las variables libres a valores, y por ende, no se puede utilizar la misma técnica que en CbN, pero, todos los operadores son estrictos por definición y podemos inducir la divergencia de la evaluación. En este último caso la construcción de contextos que se presenten

como testigos para mostrar que dos términos *no son* equivalentes se vuelve una tarea más complicada, pero en general es posible obtener una resultado similar. Esto da lugar a la siguiente pregunta: ¿cómo podemos estar seguros que los contextos, en conjunto con la evaluación, son suficientes para observar las equivalencias que buscamos? No es ninguna sorpresa que la equivalencia dependa entonces de lo que observamos sobre la evaluación. La relación entre las observaciones que hagamos sobre la evaluación de términos es lo que nos permitirá en esta tesis introducir la noción de costos. Además de observar el fenómeno de la convergencia en la evaluación de términos, observaremos otros efectos, entre ellos el costo necesario para evaluar un término.

Propiedades de la Equivalencia Observacional

Ya con la definición de equivalencia observacional, podemos desarrollar su teoría, i.e. propiedades que nos ayudarán a establecer fácilmente equivalencia entre términos. Por la naturaleza de la definición de equivalencia observacional (Definición 4.1.5), es relativamente fácil saber cuándo dos términos *no son observacionalmente equivalentes*, simplemente basta con dar un contexto testigo que discierna entre ellos.

Lamentablemente, es complicado tener que mostrar que dos términos son equivalentes, ya que hay que hacerlo *para todo contexto*. Para solucionar este problema se presentan una serie de propiedades que facilitarán las pruebas de equivalencia o aproximación observacional de términos.

Sean $M, P, Q \in \Lambda_0$ términos cerrados y $U, V, W \in \mathcal{V}_0$ valores cerrados:

$$M \sqsubseteq M \tag{4.1}$$
$$(M \sqsubseteq P \wedge P \sqsubseteq Q) \implies M \sqsubseteq Q \tag{4.2}$$
$$(M \sqsubseteq P \wedge P \sqsubseteq M) \implies M \approx P \tag{4.3}$$

$$M \sqsubseteq P \implies \boldsymbol{\lambda}\, x\, .\, M \sqsubseteq \boldsymbol{\lambda}\, x\, .\, P \tag{4.4}$$
$$P \sqsubseteq Q \implies \mathbf{let}\ x = P\ \mathbf{in}\ M \sqsubseteq \mathbf{let}\ x = Q\ \mathbf{in}\ M \tag{4.5}$$
$$P \sqsubseteq Q \implies \mathbf{let}\ x = M\ \mathbf{in}\ P \sqsubseteq \mathbf{let}\ x = M\ \mathbf{in}\ Q \tag{4.6}$$

$$V \sqsubseteq W \implies V\, U \sqsubseteq W\, U \tag{4.7}$$
$$V \sqsubseteq W \implies U\, V \sqsubseteq U\, W \tag{4.8}$$
$$V \sqsubseteq W \implies \mathbf{ret}(V) \sqsubseteq \mathbf{ret}(W) \tag{4.9}$$

Las propiedades 4.1-4.3 se desprenden directamente de las definiciones de aproximación y equivalencia observacional. Mientras que el resto se pueden probar fácilmente ya que cada propiedad se puede escribir como $P \sqsubseteq Q \implies \mathbb{C}[P] \sqsubseteq \mathbb{C}[Q]$.

Otras propiedades interesantes surgen de la relación entre la substitución, mecanismo básico de la evaluación, y la aproximación observacional:

$$V \sqsubseteq W \implies M[x := V] \sqsubseteq M[x := W] \tag{4.10}$$
$$M \sqsubseteq N \implies M[x := V] \sqsubseteq N[x := V] \tag{4.11}$$

En el caso de la propiedad 4.11 no sigue el patrón de las anteriores, ya que no se puede reescribir simplemente como un contexto, pero representa la

conexión entre la aproximación observacional y la evaluación de términos dentro del lenguaje.

Más aún, la β-reducción define términos equivalentes, al igual que el operador **let**, como podemos ver a continuación:

$$(\boldsymbol{\lambda}\ x\ .\ M)\, V \approx M[x := V] \tag{4.12}$$

$$\mathbf{let}\ x = \mathbf{ret}(V)\ \mathbf{in}\ M \approx M[x := V] \tag{4.13}$$

El hecho que la β-reducción y la evaluación de la construcción **let** sean válidas se sigue de la caracterización de $\sqsubseteq$ en términos de bisimilaridad del lenguaje que daremos en la siguiente sección. La relación de bisimilaridad se define utilizando la evaluación de los términos, y nos permite probar fácilmente propiedades sobre diferentes términos sintácticos (cerrados) pero que su evaluación se comporte de forma similar.

Finalmente, presentamos dos propiedades extensionales de la aproximación observacional.

$$\begin{array}{rcl} \boldsymbol{\lambda}\ x\ .\ N \sqsubseteq \boldsymbol{\lambda}\ x\ .\ M & \iff & \forall V \in \mathcal{V}, N[x := V] \sqsubseteq M[x := V] \\ M, N \in \Lambda(\overline{x}), M \sqsubseteq N & \iff & \forall V_1, V_2, \ldots, V_n \in \mathcal{V}, M[\overline{x} := \overline{V}] \sqsubseteq N[\overline{x} := \overline{V}] \end{array}$$

La primera relaciona las aproximaciones observacionales, sobre términos y sobre valores, mientras que la segunda permite explícitamente intercambiar apariciones de variables libres por valores. Lo mismo se podría lograr introduciendo las substituciones mediante construcciones **let**, aunque esto generaría una cadena de substituciones sucesivas que podrían llegar a reemplazar variables que originalmente estaban libres en los valores V_i con $1 \leq i \leq n$.

El término Ω es uno de los menos definidos de todos los programas. En símbolos tenemos que para todo término cerrado M:

$$\Omega \sqsubseteq M$$

Esto se desprende de la definición de aproximación observacional donde lo que se observa es la terminación de la evaluación. Intuitivamente para cualquier contexto que evalúe el agujero, la evaluación diverge, y en el caso que no lo evalúe los términos serán equivalentes. En la siguiente sección definimos una relación que nos permitirá probar formalmente que Ω es uno de los términos menos definidos.

Podemos introducir operadores de punto fijo, y por ende, recursión dentro del lenguaje. En este caso trabajaremos la recursión de la misma manera que se hace dentro de la semántica denotacional, proveyendo un punto fijo sobre la iteración del cuerpo de la función. Donde podemos definir el operador **fix** como el iterador **Z**

$$\mathbf{fix}\ x\ .\ F \equiv \mathbf{Z}\,(\boldsymbol{\lambda}\ x\ .\ F)$$

Es posible continuar probando propiedades sobre las funciones recursivas introduciendo la noción de continuidad sintáctica (*sintactic continuity*) (Pitts 2002), pero ya escapa el alcance de la tesis.

4.2. Aproximación Aplicativa

Probar que dos términos son observacionalmente equivalentes es *complejo*. Involucra probar *para todo contexto* que los términos se comportan igual,

incluso en contextos que *no tienen agujeros* o que *no los evalúan*. Una forma de simplificar la prueba es analizando cuáles son los contextos que realmente importan, definiendo un nuevo *preorden* entre términos que solo tenga en cuenta los contextos que utilizan sus agujeros. De todas maneras, queremos mantener el nivel de expresividad, por lo que primero reducimos la definición a los contextos que utilizan los agujeros y luego probaremos que es suficiente. Para esto utilizaremos una técnica creada por Abramsky (1990), y en esta sección presentaremos un resumen de su trabajo adaptado al lenguaje vehículo.

A su vez, debido a que nuestro lenguaje no alcanza formas normales sino más bien *formas normales débiles*, es decir, que no evalúa debajo de las expresiones lambda, tenemos que ser precavidos al momento de definir nociones de equivalencia entre programas. Podemos ver el problema concretamente en un ejemplo.

Ejemplo 4.2.1. *En las teorías estándares del cálculo lambda, donde se evalúan los términos debajo de las abstracciones, nos encontramos con que:*

$$\Omega \equiv \mathbf{ret}((\boldsymbol{\lambda}\ x\ .\ \Omega))$$

ya que ambos términos divergen en dichas evaluaciones. Pero dentro de nuestras definiciones la expresión $\mathbf{ret}(\boldsymbol{\lambda}\ x\ .\ \Omega)$ *es de hecho un término (cerrado) que evalúa a un valor* $(\boldsymbol{\lambda}\ x\ .\ \Omega)$*, que solo una vez que sea aplicado a un argumento su evaluación no convergerá.*

Para resolver esta divergencia entre las teorías o presentaciones estándares del cálculo lambda, Abramsky, presenta una noción llamada *Applicative Transition Systems*, como puente para unir las nociones estándares con nociones de concurrencia y otras nociones computacionales. En particular, presenta lo que se conoce como simulación aplicativa.

Nos concentramos en la evaluación de términos, en particular, utilizamos los términos que pueden ser evaluados, en otras palabras, en términos cerrados. A su vez, los programas cuya evaluación no diverge, alcanzan valores cerrados, es decir, lambdas abstracciones. Las lambda abstracciones pueden ser interpretadas como funciones anónimas, que guiándonos por la semántica toman un argumento y nos retornan un nuevo término. Lo que nos lleva entonces a preguntarnos cómo podríamos comparar funciones. Las matemáticas ya nos dan una solución, de forma *extensional*:

Definición 4.2.1 (Principio de Extensionalidad)**.** *Dadas dos funciones matemáticas* $f, g\colon A \to B$*, decimos que son extensionalmente equivalentes si y solo si para todo* $a \in A$ *tenemos que* $f(a) = g(a)$*.*

En nuestro caso con las lambdas abstracciones podemos hacer lo mismo, aunque tenemos que tener cuidado en no caer en un *razonamiento circular*. Una vez que apliquemos las lambdas abstracciones a un mismo argumento, obtenemos como resultado dos nuevos términos: ¿Cómo comparamos éstos nuevos términos?¿Podemos volver a usar la misma noción de aproximación aplicativa?. Para evitar caer en un razonamiento circular sin sentido, definimos la intuición de comparación extensional de valores recién mencionados en una operación monótona y utilizamos el teorema de Knaster-Tarski (Teorema 2.1.6) para encontrar el punto fijo de dicho razonamiento.

Podemos pensarlo como un experimento de la evaluación de términos. Dado un término cerrado M, el único *experimento* que podemos hacer es evaluarlo y ver si converge a una abstracción $\boldsymbol{\lambda}\ x\ .\ M_1$. Si lo hace, podemos llevar el experimento a un segundo nivel, tomar un término N_1, y realizar el mismo experimento con $(\boldsymbol{\lambda}\ x\ .\ M_1)\ N_1$, y así sucesivamente. Lo único que se observa en cada paso es el hecho de si la evaluación converge, no se exploran los términos o valores alcanzados. De esta manera podemos ir realizando experimentos a diferentes niveles de profundidad, observando en cada experimento que la evaluación de los términos converge.

Definición 4.2.2 (Extensión Aplicativa)**.** *Sea R una relación entre términos cerrados. Notamos como $\overrightarrow{R}$ a la extensión aplicativa definida como:*

$$V\ \overrightarrow{R}\ W \iff \forall U \in \mathcal{V}_0, V\,U\ R\,W\,U$$

Definición 4.2.3 (Aproximación Aplicativa)**.** *Sean M y N dos términos cerrados. Decimos que M aproxima aplicativamente a N, $M \sqsubseteq^A N$ si y solo si siempre que exista un valor cerrado V tal que $M \Downarrow_n \iota_l(V)$ para algún $n \in \mathbb{N}$ entonces existe un valor cerrado W tal que $N \Downarrow_m \iota_l(W)$ para algún $m \in \mathbb{N}$ y $V \overrightarrow{\sqsubseteq^A} W$.*

Podemos extender la relación a términos abiertos de la siguiente forma, sean $M, N \in \Lambda$:

$$M \sqsubseteq_\omega N \equiv \forall \sigma : Var \to \mathcal{V}_0, M\sigma \sqsubseteq N\sigma$$

Donde $M\sigma$ es la operación de substitución de toda variable libre $x \in \mathsf{FV}(M)$ por el valor $\sigma(x)$. Es decir, es posible extender la relación a términos abiertos simplemente cuantificando para toda substitución posible.

Finalmente definiendo la noción de *bisimulación* como la aproximación aplicativa mutua:

$$M \sim N \doteq M \sqsubseteq_\omega N \wedge N \sqsubseteq_\omega M$$

Para ver efectivamente que la definición de simulación aplicativa está bien fundada utilizamos un operador monótono y el teorema de Knaster-Tarski (Teorema 2.1.6). A modo de repaso, en el Capítulo 3, además de definir la sintaxis de nuestro lenguaje, definimos una noción de evaluación mediante una familia de relaciones de evaluación que van aproximando el valor asignado a un término. En particular caracterizamos la relación de evaluación como $[\![-]\!] : \Lambda_0 \to \mathcal{V}_0 + \bot$.

Definición 4.2.4. *Definimos el siguiente operador entre relaciones:*

$$\mathcal{O} : (\Lambda_0 \times \Lambda_0) \times (\mathcal{V}_0 \times \mathcal{V}_0) \to (\Lambda_0 \times \Lambda_0) \times (\mathcal{V}_0 \times \mathcal{V}_0)$$

Donde, sean $\mathcal{R}_\mathcal{V}, \mathcal{R}_\Lambda$ dos relaciones entre valores y términos cerrados respectivamente.

$$\mathcal{O}_\mathcal{V}(\mathcal{R}_\Lambda, \mathcal{R}_\mathcal{V}) = \{(V,U) : \forall W \in \mathcal{V}_0, V\,W\ \mathcal{R}_\Lambda\ U\,W\}$$

$$\mathcal{O}_\Lambda(\mathcal{R}_\Lambda, \mathcal{R}_\mathcal{V}) = \{(M,N) : \forall U \in \mathcal{V}_0, M \Downarrow U \implies \exists W \in \mathcal{V}_0, N \Downarrow W \wedge U\ \mathcal{R}_V\ W\}$$

Definimos el operador de la siguiente manera $\mathcal{O} = (\mathcal{O}_\mathcal{V}, \mathcal{O}_\Lambda)$.

Veamos que el operador recién definido es en efecto monótono respecto a la inclusión en el reticulado completo de pares de relaciones entre dos conjuntos.

Lema 4.2.1. *El operador $\mathcal{O}$ es monótono.*

Demostración. Sean $A, B \subseteq Rel(\mathcal{V}_0), P, Q \subseteq Rel(\Lambda_0)$ tales que $A \subseteq B \wedge P \subseteq Q$. Veamos entonces que

$$\mathcal{O}(A, P) \subseteq \mathcal{O}(B, Q)$$

Podemos separar la prueba en dos casos: por un lado que la contención entre las relaciones resultantes en los conjuntos de valores se mantiene, $\mathcal{O}_{\mathcal{V}}(A, P) \subseteq \mathcal{O}_{\mathcal{V}}(B, Q)$, y por otro, que la contención entre las relaciones resultantes en los conjuntos de términos también se mantiene, $\mathcal{O}_{\Lambda}(A, P) \subseteq \mathcal{O}_{\Lambda}(B, Q)$.

- Primer caso: $\mathcal{O}_{\mathcal{V}}(A, P) \subseteq \mathcal{O}_{\mathcal{V}}(B, Q)$. Sean W, U valores cerrados tales que $W\ \mathcal{O}_{\mathcal{V}}(A, P)\ U$. Por definición de $\mathcal{O}_{\mathcal{V}}(A, P)$, para cualquier valor cerrado V, $(W\,V)\ P\ (U\,V)$. Por $P \subseteq Q$, para todo valor cerrado V, $(W\,V)\ Q\ (U\,V)$. Finalmente concluimos, por su definición, que $W\ \mathcal{O}_{\mathcal{V}}(B, Q)\ U$.

- Segundo caso: $\mathcal{O}_{\Lambda}(A, P) \subseteq \mathcal{O}_{\Lambda}(B, Q)$. La prueba es similar al caso anterior, partimos de dos elementos relacionados, aplicamos la definición del operador y utilizamos la hipótesis que $A \subseteq B$. Sean $M, N \in \Lambda_0$, tales que $M\ \mathcal{O}_{\Lambda}(A, P)\ N$. Por definición de la relación $\mathcal{O}_{\Lambda}(A, P)$, se tiene que para todo valor $W \in \mathcal{V}_0, M \Downarrow W$, existe un valor $U \in \mathcal{V}_0, N \Downarrow U$ y además $W\ A\ U$. Sabemos por hipótesis que $A \subseteq B$, por lo que $W\ B\ U$. Por lo tanto $M\ \mathcal{O}_{\Lambda}(B, Q)\ N$.

Por lo que el operador $\mathcal{O}$ es monótono. □

Al ser el operador monótono y los conjuntos ordenados por contención un *reticulado completo*, tenemos por el teorema de Knaster-Tarski (Teorema 2.1.6) la existencia del menor punto fijo y el mayor punto fijo. Notar que los puntos fijos son aquellos que dan una solución a las ecuaciones que se plantearon en la definición de Aproximación Aplicativa (Definición 4.2.3). ¿Ahora cuál de los dos es el que queremos?

Por un lado tenemos que el menor punto fijo es el par de relaciones donde se caracterizan los términos cuya evaluación diverge o convergen a valores que al ser aplicados divergen. Esto es debido a que el operador $\mathcal{O}$ relaciona términos observando la convergencia de su evaluación y utilizando el operador lógico de implicación. En caso de que el antecedente de falso, significando que la evaluación diverge, el consecuente no necesario. Mecánicamente, el menor punto fijo está caracterizado por la aplicación sucesiva del operador $\mathcal{O}$ al par de relaciones vacías.

$$(\emptyset, \emptyset) \subseteq \mathcal{O}(\emptyset, \emptyset) \subseteq \mathcal{O}^2(\emptyset, \emptyset) \subseteq \ldots \subseteq \bigsqcup_{n \in \mathbb{N}} \mathcal{O}^n(\emptyset, \emptyset)$$

Comenzamos por la primer aplicación del operador.

$$\begin{array}{lcl} \mathcal{O}_{\mathcal{V}}(\emptyset, \emptyset) & = & \emptyset \\ \mathcal{O}_{\Lambda}(\emptyset, \emptyset) & = & \{(M, N) \in \Lambda_0 \times \Lambda_0 \mid M \Downarrow \iota_r(\bot)\} \end{array}$$

Definimos a al conjunto de términos cerrados cuya evaluación diverge como $D_{\Lambda} \doteq \{M \in \Lambda_0 \mid M \Downarrow \iota_r(\bot)\}$. Reescribir el resultado de la aplicación del

operador $\mathcal{O}$ al par de relaciones vacías como: $\mathcal{O}(\emptyset,\emptyset) = (D_\Lambda \times \Lambda_0, \emptyset)$. Si volvemos a aplicar el operador al resultado anterior, $\mathcal{O}(\mathcal{O}(\emptyset,\emptyset))$, obtenemos las siguientes relaciones:

$$\begin{array}{lcl} \mathcal{O}_\mathcal{V}(D_\Lambda \times \Lambda_0, \emptyset) & = & \{V, W \in \mathcal{V}_0 \mid \forall W \in \mathcal{V}_0, V\, W\ D_\Lambda \times \Lambda_0\ U\, W\} \\ \mathcal{O}_\Lambda(D_\Lambda \times \Lambda_0, \emptyset) & = & D_\Lambda \times \Lambda_0 \end{array}$$

Definimos el conjunto de *valores divergentes* como aquellos valores que ante cualquier argumento su evaluación diverge: $D_\mathcal{V} \doteq \{V \in \mathcal{V}_0 \mid \forall W \in \mathcal{V}_0, V\, W \in D_\Lambda\}$. Utilizando el conjunto de valores divergentes, la segunda iteración de la aplicación del operador de relaciones aplicado al par de relaciones vacías resulta en dos relaciones que caracterizan a los términos que cuya evaluación diverge y a valores que al ser utilizados su evaluación diverge, $\mathcal{O}^2(\emptyset,\emptyset) = (D_\Lambda \times \Lambda_0, D_\mathcal{V} \times \mathcal{V}_0)$.

Si seguimos aplicando el operador $\mathcal{O}$ a las relaciones resultantes, lo que estamos es continuar observando la divergencia de términos tal como lo define la definición aplicativa de Abramsky. Observamos los términos divergentes en la evaluación, luego los valores cuyas evaluaciones divergen al ser aplicados, luego adicionamos los términos que convergen a un valor que al ser aplicado diverge, y así sucesivamente. Cada vez que aplicamos el operador agregamos un nuevo nivel de observación de "términos cuya evaluación converge a un valor que al ser aplicado cae en la observación anterior". Los conjuntos de términos y valores divergentes los podemos caracterizar como una familia de conjuntos expresando el nivel de observaciones que realizamos.

$$\begin{array}{lcl} D^n_\Lambda & \doteq & \{M, N \in \Lambda_0 \mid M \Downarrow V \implies N \Downarrow U \wedge (V, U) \in D^{n-1}_\mathcal{V}\} \\ D^n_\mathcal{V} & \doteq & \{V, U \in \mathcal{V}_0 \mid \forall W \in \mathcal{V}_0, (V\, W) D^{n-1}_\Lambda (U\, W)\} \end{array}$$

Con los casos bases como las relaciones vacías: $D^0_\mathcal{V} = D^0_\Lambda \doteq \emptyset$.

Dado que el menor punto fijo del operador define relaciones ligadas a términos cuya evaluación diverge, no presenta un caso de estudio útil.

El mayor de los puntos fijos nos otorga además una definición *coinductiva* de la relación y con ella un *principio de coinducción*. Este resultado se desprende del Teorema 2.1.6, y es la caracterización del mayor punto fijo como la menor cota superior de los post-puntos fijos.

Decimos que el par de relaciones $R = (\mathcal{R}_V, \mathcal{R}_T)$ es una simulación aplicativa si $R \subseteq \mathcal{O}(R)$; definimos la relación (doble) de aproximación aplicativa como:

$$\sqsubseteq^A \doteq \bigcup \{R \mid R \text{ es una simulación aplicativa } \}$$

Donde tenemos en realidad dos relaciones de aproximación, una para cada categoría sintáctica. Si reemplazamos los símbolos por sus definiciones tenemos una definición de aproximación aplicativa como pares de relaciones, una para términos y otra para valores del lenguaje. Como resultado tenemos la relación de aproximación aplicativa definida coinductivamente dada a continuación.

Definición 4.2.5 (Aproximación Aplicativa como pares de relaciones)**.** *Definimos las relaciones* $(\leq_\mathcal{V}) \subseteq \mathcal{V}_0 \times \mathcal{V}_0$ *y* $(\leq_\Lambda) \subseteq \Lambda_0 \times \Lambda_0$ *como:*

$$\begin{array}{l} V \leq_\mathcal{V} W \iff \forall U \in \mathcal{V}_0, V\, U \leq_\Lambda W\, U \\ M \leq_\Lambda N \iff \forall V \in \mathcal{V}_0, M \Downarrow V \implies \exists W \in \mathcal{V}_0, N \Downarrow W \wedge V \leq_\mathcal{V} W \end{array}$$

Donde $\sqsubseteq^A \equiv (\leq_\mathcal{V}, \leq_\Lambda)$

Inspeccionando la definición de aproximación aplicativa dada al principio de la sección (Definición 4.2.3) se puede ver que es equivalente a la definición de aproximación aplicativa como pares de relaciones. Al definir la aproximación aplicativa como la unión de todas las simulaciones aplicativas, i.e. coinductivamente, para mostrar que dos términos están relacionados basta entonces con dar una simulación aplicativa que haga de testigo.

4.3. El Método de Howe

Un propiedad que toda noción de equivalencia entre programas debe tener es la propiedad de ser una *congruencia* y, bajo ciertas condiciones, que la simulación aplicativa está contenida en la equivalencia observacional. Una relación congruente entre términos del lenguaje nos indica que es compatible con las construcciones sintácticas del lenguaje. En particular, nos facilita el razonamiento ecuacional permitiéndonos utilizar transitividad para probar equivalencia entre varios términos, y por sobre todo, la capacidad de reemplazar iguales por iguales.

El enfoque utilizado por esta tesis es el de definir una noción de equivalencia de programas de forma coinductiva basada en la noción de bisimulación, por lo que, no es tan fácil ver que sea una congruencia.

En esta sección presentamos el método de Howe para mostrar que la relación de equivalencia definida de forma coinductiva es efectivamente una congruencia (Howe 1989).

El método de Howe fue muy utilizado luego de la definición de simulación de Abramsky (1990), fue extendido a lenguajes con alto orden (Pitts 2012), y más recientemente, a lenguajes con una cantidad de constructores arbitraria (P. B. Levy 2006). En particular, esta última extensión da lugar a que en este trabajo podamos saltarnos la limitación original de tener una cantidad finita de constructores del lenguaje y poder trabajar directamente con signaturas con una cantidad numerable de constructores.

En esta sección explicaremos brevemente el método utilizando el lenguaje presentado en el Capítulo 3. A diferencia de la presentación original nuestro lenguaje no es de tipo lazy, pero la definición de aproximación aplicativa (Definición 4.2.5) es la misma.

Introducimos una forma de extender una relación entre términos cerrados a términos abiertos.

Definición 4.3.1. *Dada un par de relaciones $(R_\Lambda, R_\mathcal{V})$ entre términos y valores, y un conjunto de variables $\overline{x} \subseteq \mathbb{V}$. Definimos la extensión de $(R_\Lambda, R_\mathcal{V})$ a términos abiertos con variables libres en $\overline{x}$ donde dos términos $M, N \in \Lambda(\overline{x})$ están relacionados*

$$\overline{x} \vdash M \; R \; N$$

si y solo si, para todo valores cerrados $V_1, \ldots, V_n \in \mathcal{V}_0$, tenemos que:

$$M[\overline{x} := \overline{V}] \; R \; N[\overline{x} := \overline{V}]$$

Definiciones sobre relaciones abiertas Podemos entonces definir las siguiente propiedades sobre relaciones abiertas. Sea $R = (R_\Lambda, R_\mathcal{V})$ una relación entre términos y valores, y $\overline{x}$ un conjunto de variables en $\mathbb{V}$.

- La relación R es *simétrica* si y solo si para todo par de términos $M, N \in \Lambda(\overline{x})$ tenemos que:

$$\overline{x} \vdash M \; R \; N \implies \overline{x} \vdash N \; R \; M$$

- La relación R es *transitiva* si y solo si para todo terminos $M, N, S \in \Lambda(\overline{x})$ tenemos que

$$\overline{x} \vdash M \; R \; N \wedge \overline{x} \vdash N \; R \; S \implies \overline{x} \vdash M \; R \; S$$

- La relación R es *compatible* si y solo si los siguientes predicados son válidos con $\overline{x} \subseteq \mathbb{V}$:
  - $\forall x \in \overline{x}.\overline{x} \vdash x \; R \; x$
  - $\forall x \notin \overline{x}.\overline{x} \cup \{x\} \vdash M \; R \; N \implies \overline{x} \vdash (\boldsymbol{\lambda} \; x \; . \; M) \; R \; (\boldsymbol{\lambda} \; x \; . \; N)$
  - $\overline{x} \vdash V \; R \; W \implies \overline{x} \vdash \mathbf{ret}(V) \; R \; \mathbf{ret}(W)$
  - $\overline{x} \vdash V \; R \; V' \wedge W \; R \; W' \implies \overline{x} \vdash (V \; W) \; R \; (V' \; W')$
  - $\forall x \notin \overline{x}, \overline{x} \vdash M \; R \; M' \wedge \overline{x} \cup \{x\} \vdash N \; R \; N'$
    $\implies \overline{x} \vdash (\mathbf{let} \; x = M \; \mathbf{in} \; N) \; R \; (\mathbf{let} \; x = M' \; \mathbf{in} \; N')$

Por lo que ya podemos definir que una relación entre términos y valores es una *precongruencia* si y solo si cumple con las propiedades de transitividad y es compatible. Mientras que una congruencia es además una relación que es simétrica.

El método de Howe consiste entonces en mostrar que la extensión a términos abiertos de la simulación aplicativa es una precongruencia. El método se basa en construir una relación que es precongruente por definición y probar que describe a todos los términos relacionados en la relación original, es decir, que son equivalentes. Esta nueva relación se la conoce como "candidato precongruente". En nuestro caso lo que haremos es tomar la relación de aproximación aplicativa y ver que es una precongruencia.

Definición 4.3.2 (Candidato Precongruente)**.** *Definimos un par de relaciones, entre valores* $(\leq_{\mathcal{V}}^*) \subseteq \mathcal{V} \times \mathcal{V}$*, y entre términos* $(\leq_{\Lambda}^*) \subseteq \Lambda \times \Lambda$*, mutuamente recursivas por inducción en la construcción del elemento a izquierda.*

Sean $V, W \in \mathcal{V}$*, por inducción en* V*:*

- *Para toda variable* $x \in \mathbb{V}$*,* $x \leq_{\mathcal{V}}^* W$ *si* $x \leq_{\mathcal{V}} W$
- $\boldsymbol{\lambda} \; x \; . \; M \leq_{\mathcal{V}}^* W$ *si existe* M' *tal que* $M \leq_{\Lambda}^* M'$ *y* $\boldsymbol{\lambda} \; x \; . \; M' \leq_{\mathcal{V}} W$

Sean $M, N \in \Lambda$*, por inducción en* M*:*

- *Para todo valor* $V \in \mathcal{V}$*,* $\mathbf{ret}(V) \leq_{\Lambda}^* N$ *si si existe* V' *tal que* $V \leq_{\mathcal{V}}^* V'$ *y* $\mathbf{ret}(V') \leq_{\Lambda} P$
- *Para todo valor* $V, W \in \mathcal{V}$*,* $V \; W \leq_{\Lambda}^* P$ *si existe* V', W' *tales que* $V \leq_{\mathcal{V}}^* V'$ *y* $W \leq_{\mathcal{V}}^* W'$*, y además,* $V' \; W' \leq_{\Lambda} N$
- $\mathbf{let} \; x = M \; \mathbf{in} \; N \leq_{\Lambda}^* P$ *si existe* M', N' *tales que* $M \leq_{\Lambda}^* M'$ *y* $N \leq_{\Lambda}^* N'$*, y además,* $\mathbf{let} \; x = M' \; \mathbf{in} \; N' \leq_{\Lambda} P$

Informalmente, podemos ver que $M \leq_{\Lambda}^* N$ si N se puede obtener a partir de M reemplazando subtérminos en M por términos que son más *grandes* en relación a $\leq_{\Lambda}$ y $\leq_{\mathcal{V}}$. Como consecuencia inmediata de la definición del candidato, tenemos que el par de relaciones $({\leq_{\Lambda}}^*, {\leq_{\mathcal{V}}}^*)$ es una precongruencia cerrada bajo la substitución y además para todo término $M, N, P \in \Lambda$:

- si $M \leq^*_\Lambda N \wedge N \leq^*_\Lambda P$ entonces $M \leq^*_\Lambda P$
- $(\leq_\Lambda) \subseteq (\leq^*_\Lambda)$

Las relaciones $(\leq^*_\mathcal{V})$ y $(\leq_\mathcal{V})$ también son cerradas bajo la substitución y poseen las mismas propiedades recién escritas.

La tarea que nos queda, y es donde entra la genialidad de Howe, es probar que $(\leq^*_\Lambda, \leq^*_\mathcal{V}) \subseteq (\leq_\Lambda, \leq_\mathcal{V})$, y por ende, son iguales.

Para esto, Howe recurre a una propiedad sobre los constructores del lenguaje que denomina *extensionalidad*.

Definición 4.3.3 (Operadores Extensionales)**.** *Un operador τ es extensional si y solo si para cualesquiera términos cerrados $\tau(\overline{T}), \tau(\overline{T'})$, valor cerrado V y para todo $k \geq 0$ si*

- $\tau(\overline{T}) \Downarrow_k V$
- $\overline{T} \leq^*_\Lambda \overline{T'}$
- *para todo término cerrado S, P y valor cerrado W, si $S \Downarrow_{k'} W$ con $k' \leq k$ y $S \leq^*_\Lambda P$ entonces $S' \leq^*_\Lambda P$*

Entonces $\mathbf{ret}(V) \leq^*_\Lambda \tau(\overline{T'})$.

Un lenguaje es extensional si todos sus operadores son extensionales. La extensionalidad de los operadores es necesaria para probar que el candidato precongruente está contenido en la simulación aplicativa, ya que nos permite probar que si $M \leq^*_\Lambda N$ y $M \Downarrow_k V$ entonces $\mathbf{ret}(V) \leq^*_\Lambda N$. En otras palabras establece que el candidato precongruente respeta los pasos intermedios en la evaluación de términos. En nuestro cálculo se puede ver que los operadores de nuestro lenguaje son extensionales ya que la evaluación está guiada por la sintaxis y podemos hacer crecer los términos mediante las relaciones $\leq_\Lambda$ y $\leq_\mathcal{V}$ que respetan la evaluación.

Teorema 4.3.1. *Para todo término cerrado $M, N \in \Lambda_0$, $M \leq^*_\Lambda N$ implica que $M \leq_\Lambda N$.*

Esta es una conclusión del Teorema 1 (Howe 1989) ya que nuestro lenguaje es extensional. Concluyendo lo que esperábamos, que la simulación aplicativa es una precongruencia. Para una prueba en detalle de los teoremas se recomienda la lectura del artículo citado.

Finalmente, Howe estudia las propiedades que tiene que haber entre las observaciones que pueden realizarse mediante el uso de contextos en el lenguaje para poder establecer la equivalencia entre la simulación aplicativa y la equivalencia observacional. Por ejemplo, no nos alcanza con lenguajes extensionales; para ilustrarlo Howe introduce un lenguaje con un constructor no-determinista. Asumiendo que tenemos un constructor $amb(M, N) \Downarrow V$ si y solo si $M \Downarrow V$ o $N \Downarrow V$, podemos ver que $amb(I\,0, I\,1)$ y $I\;amb(\mathbf{ret}(0), \mathbf{ret}(1))$ son contextualmente equivalente, pero no podremos probar que son aplicativamente similares. En otras palabras, hay más términos contextualmente equivalentes que aplicativamente similares. Eventualmente, esa línea de investigación fue abandonada, ya que se encontraron casos de programas con efectos donde no se cumple que las nociones de equivalencia y similaridad sean equivalentes (Koutavas, P. B. Levy y Sumii 2011).

Sin embargo, Howe logró caracterizar aquellos lenguajes en los cuales sí es el caso que la aproximación observacional y la simulación aplicativa son equivalentes. Para esto, introduce entonces el concepto de *computacionalmente adecuado*.

La idea consiste en capturar cuando los contextos pueden ser representados por la substitución de variables libres en los términos por simples valores.

Teorema 4.3.2. *Si para todo $M \sqsubseteq N$ implica que $M\sigma \sqsubseteq N\sigma$ para toda substitución σ que cierre a M y N, entonces para todo par de términos $M, N \in \Lambda$, $M \sqsubseteq N \iff M \leq_\Lambda N$*

En esta sección repasamos los conceptos básicos del método de Howe y la aplicamos al lenguaje que utilizamos como vehículo con dos categorías sintácticas. Una diferencia es que el método de Howe originalmente se plantea para un sistema de evaluación lazy mientras que el nuestro es eager. Recientemente el método recibió más atención y fue actualizado y expandido a lenguajes como el que utilizamos en esta sección (P. B. Levy 2006). Además, esta nueva versión del método de Howe, permite introducir efectos algebraicos y es la técnica que usaremos en la segunda parte de la tesis. Retomamos la discusión en el Capítulo 7.

Capítulo 5

Teorías de Mejoras

La teoría de mejoras (en inglés *improvement theory*) es un cálculo inventado y desarrollado por *David Sands* (1998), como continuación de su tesis doctoral (Sands 1990). En su tesis, Sands, se dedicó a estudiar la *complejidad temporal* y el *análisis de costos* de lenguajes funcionales **de alto orden** y **perezosos**. Al poco tiempo de presentar su tesis, presentó un *draft* donde se menciona por primera vez a la teoría de mejoras (Sands 1991)

La teoría de mejoras se basa directamente en definir *qué es una mejora*, y mostrar que es una propiedad modular, es decir, que es posible *mejorar* un programa mejorando partes de él. En concreto, una modificación válida debe *mantener el comportamiento del programa* mejorando la ejecución del mismo.

Continuando con el razonamiento del capítulo anterior, buscamos definir una relación binaria de manera tal que: tiene que ser un preorden (idempotente y transitiva), y además, sustitutiva, es decir, deberíamos ser capaces de substituir mejoras por mejoras (como razonamiento ecuacional de iguales por iguales). En otras palabras, buscamos una relación que sea una *precongruencia*. Ya que la teoría se basa en un razonamiento inecuacional, una relación de mejoras es asimétrica (queremos mejorar un término, no empeorarlo), no buscamos una relación que sea una *congruencia*.

5.1. Instrumentación de Relaciones

Asumamos que T es el conjunto de términos, que alguno de estos términos son valores, $V \subseteq T$ y que tenemos una relación de evaluación $(\Downarrow) \subseteq T \times V$. La teoría de mejoras se construye a partir de *instrumentar* dicha relación de manera tal que exponga propiedades intensivas sobre la evaluación de términos. Podemos definir una familia de relaciones indexadas por los naturales de manera tal que $M \Downarrow^n V$ represente que el término M evalúa al valor V utilizando n recursos. Aquí los naturales representan simplemente un conjunto ordenado cuantificando el consumo de recursos.

Decimos que a una relación R está o es instrumentada si es posible obtener una familia de relaciones R^n de manera tal que, además de relacionar los mismos elementos que R, expone propiedades intensivas de la relación R. En nuestro caso, en vez de solo relacionar términos con el valor que se alcanzaría siguiendo la evaluación, exponemos además el costo requerido. En símbolos, dada una relación $(\Downarrow) \subseteq T \times V$, una relación instrumentada de $\Downarrow$ es una familia de

relaciones $(\Downarrow^n) \subseteq T \times V$ de manera tal que $(t, v) \in \Downarrow^n$ si y solo si la evaluación de t resulta en v siguiendo la evaluación $\Downarrow$ y además se utilizan exactamente n unidades de recursos. Para una misma relación puede haber diferentes formas de instrumentarla, dependiendo por ejemplo de qué recurso se quiere medir. En esta tesis definiremos una forma de instrumentar la relación de evaluación del lenguaje presentado y la utilizaremos para obtener una teoría de mejoras.

De esta manera podemos definir de forma genérica la noción de mejora como la observación de terminación más la mejora en el uso de recursos por parte de la evaluación de los términos. En otras palabras, refinando la noción de aproximación observacional (Definición 4.1.4) donde además de terminación se compara el costo de evaluar los términos involucrados.

Definición 5.1.1 (Mejoras en Relaciones Instrumentadas)**.** *Dada una relación instrumentada entre términos y valores* $(\Downarrow^n) \subseteq T \times V$*, y dados dos términos* M, N*. Decimos que el término* M *es mejorado por el término* N*,* $M \succ N$*, si y solo si para todo contexto* $\mathbb{C}$ *que cierra los términos* M, N $\mathbb{C}[M], \mathbb{C}[N] \in T_0$ *tenemos que:*

$$\forall n \in \mathbb{N}, U \in V, \mathbb{C}[M] \Downarrow^n U \implies \exists m \in \mathbb{N}, W \in V, \mathbb{C}[N] \Downarrow^m W \wedge n \geq m$$

En palabras, decimos que el término M es mejorado por el término N si y solo si en cualquier contexto en el que la evaluación de M alcance un valor, la evaluación de N en ese contexto también alcanza un valor, utilizando a lo sumo la misma cantidad de recursos que M. O de otro modo, no hay contexto en el cual el uso de N sea peor que el uso de M.

La definición de mejoras, al igual que la definición de equivalencia contextual de Morris, si bien define intuitivamente qué es una mejora, es complicada de utilizar. La cuantificación universal sobre los contextos introduce una dificultad al momento de realizar pruebas de equivalencias y mejoras, y por ende, se han buscado e implementado diferentes métodos para simplificar el trabajo. Por ejemplo, presentar un álgebra de ticks (Moran y Sands 1999a) o incluso un *lema de contexto*, donde se busca reducir el conjunto de contextos a aquellos relevantes para la evaluación.

La teoría de mejoras se basa en poder instrumentar la evaluación de los términos haciendo visible propiedades intensivas del proceso de evaluación. Para esto, utilizaremos todas las técnicas introducidas hasta el momento, terminando con una definición de relación de mejoras que es una precongruencia.

5.2. Mejoras en la Evaluación de Términos

A modo de ejemplo, en esta sección instrumentaremos la evaluación de términos del Capítulo 3 de manera tal que exponga la cantidad de *pasos* que se requieren para evaluar un término. En otras palabras, definimos una familia de funciones parciales $[\![-]\!]_{\mathbb{N}} : \Lambda_0 \rightharpoonup (\mathcal{V}_0 \times \mathbb{N})$. Para esto, definimos una nueva relación que asocia a cada valor un número natural representando la cantidad de pasos que se utilizaron para alcanzarlo. Durante este trabajo utilizamos el conjunto de los naturales, pero bastaría con emplear cualquier monoide ordenado, el objetivo es contar los recursos utilizados.

Primero definimos una familia de funciones auxiliares para adicionar una unidad al costo a un par de un elemento de un conjunto X y su costo.

$$\begin{aligned} &succ_X : (X \times \mathbb{N}) \to (X \times \mathbb{N}) \\ &succ_X(x, m) \mapsto (x, m + 1) \end{aligned}$$

Notamos como $succ_X^n$ con $n \in \mathbb{N}$ a la función resultante de componer a la función $succ_X$ consigo misma n veces, en palabras, adicionamos n al costo. Además, para simplificar la notación, en general siempre que se pueda determinar el subíndice por su contexto, será omitido.

A continuación definimos la evaluación instrumentada de términos donde contamos cada vez que se *consume* un constructor del lenguaje. Para esto definimos la evaluación instrumentada mediante una relación de evaluación aproximada (Definición 3.3.1).

Definición 5.2.1 (Evaluación Aproximada de Términos Instrumentada)**.** *Definimos una familia de relaciones $\Downarrow_n^{\mathbb{N}} \subseteq \Lambda_0 \times ((\mathcal{V}_0 \times \mathbb{N}) + \bot)$ con $n \in \mathbb{N}$ de la siguiente forma:*

$$\mathtt{lBot}\ \frac{}{M \Downarrow_0^{\mathbb{N}} \iota_r(\bot)} \qquad \mathtt{lRet}\ \frac{}{\mathbf{ret}(V) \Downarrow_{n+1}^{\mathbb{N}} \iota_l(V, 1)}$$

$$\mathtt{lApp}\ \frac{M[x := W] \Downarrow_n^{\mathbb{N}} X}{(\boldsymbol{\lambda}\ x\ .\ M)\, W \Downarrow_{n+1}^{\mathbb{N}} succ(X)} \qquad \mathtt{lLetD}\ \frac{M \Downarrow_n^{\mathbb{N}} \iota_r(\bot)}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow_{n+1}^{\mathbb{N}} \iota_r(\bot)}$$

$$\mathtt{lLetV}\ \frac{M \Downarrow_n^{\mathbb{N}} \iota_l(V, m) \qquad N[x := V] \Downarrow_n^{\mathbb{N}} a}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow_{n+1}^{\mathbb{N}} succ^{m+1}(a)}$$

Donde $a \in \mathcal{V}_0 + \bot$.

Utilizando la familia de relaciones resultante de instrumentar la evaluación aproximada de términos (ver Capítulo 3), obtenemos una nueva relación de evaluación utilizando el supremo de la cadena de aproximaciones y podemos obtener las siguiente ecuaciones.

Lema 5.2.1 (Evaluación Instrumentada)**.** *La relación de evaluación de términos instrumentada respeta las siguientes ecuaciones.*

$$\begin{aligned} [\![-]\!]_{\mathbb{N}} &: \Lambda_0 \rightharpoonup (\mathcal{V}_0 \times \mathbb{N}) \\ [\![\mathbf{ret}(\boldsymbol{\lambda}\ x\ .\ T)]\!]_{\mathbb{N}} &= succ(\boldsymbol{\lambda}\ x\ .\ T, 0) \\ [\![(\boldsymbol{\lambda}\ x\ .\ T)\, W]\!]_{\mathbb{N}} &= succ([\![T[x := W]]\!]_{\mathbb{N}}) \\ [\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]_{\mathbb{N}} &= succ(((v, m) \mapsto succ^m([\![N[x := v]]\!]_{\mathbb{N}}))([\![M]\!]_{\mathbb{N}})) \end{aligned}$$

De esta manera podemos definir cuando un término del lenguaje es mejorado por otro observando la terminación en la evaluación pero además la cantidad de recursos computacionales utilizados.

Definición 5.2.2 (Mejoras en $(\Lambda, \mathcal{V})$)**.** *Dado dos términos $M, N \in \Lambda$ decimos que M es mejorado por N, $M \rhd N$, si y solo si para todo contexto $\mathbb{C}$ de forma tal que $\mathbb{C}[M], \mathbb{C}[N] \in \Lambda_0$, tenemos que:*

$$[\![\mathbb{C}[M]]\!]_{\mathbb{N}} = (V, n) \implies [\![\mathbb{C}[N]]\!]_{\mathbb{N}} = (W, m) \wedge (n \geq m)$$

Definir mejoras utilizando la definición directamente es, en general, complicado debido principalmente a que se cuantifica universalmente en todo contexto que genere términos cerrados. Esto lo podemos ver en un simple ejemplo.

Ejemplo 5.2.1. *La β-reducción es una mejora. Sea x una variable, un término $T \in \Lambda_x$, y un valor $V \in \mathcal{V}_0$, veamos que $(\boldsymbol{\lambda}\ x\ .\ T)\, V \rhd T[x := V]$.*

Dado que $((\boldsymbol{\lambda}\ x\ .\ T)\, V)$ y $(T[x := V])$ son términos cerrados, el contexto en el que se los ubique no jugará ningún papel en la evaluación del término. Pero además sabemos que la evaluación ocasiona un gasto de recursos computacionales.

$$[\![(\boldsymbol{\lambda}\ x\ .\ T)\, V]\!]_{\mathbb{N}} = succ([\![T[x := V]]\!]_{\mathbb{N}})$$

Entonces podemos argumentar que en cualquier contexto que utilice el primer término emplearía mayor gasto computacional que el segundo.

5.3. Simulación de Mejoras

Intuitivamente es sencillo ver que la β-reducción es una mejora, pero mostrarlo siguiendo la definición es más complicado.

Lo que podemos hacer es encontrar una relación de mejoras entre términos que sea más sencilla de probar, y que podamos utilizarla para probar mejoras. Para esto podemos emplear una solución similar la que utilizada por Abramsky, introduciendo el concepto de simulación (Definición 4.2.3), en nuestro caso una *simulación de mejoras*.

Debido a que tenemos dos categorías sintácticas, términos y valores, deberemos definir una simulación como un par de simulaciones mutuamente recursivas. Recordemos que dos términos M, N son similares si la evaluación del término M evalúa a un valor similar al resultado de la evaluación de N y lo notamos $N \sqsubseteq M$.

Utilizando un razonamiento similar, podemos utilizar la evaluación instrumentada de términos (Definición 5.2.1) introduciendo la noción de costos en la definición.

Definición 5.3.1 (Simulación Aplicativa de Mejoras)**.** *Sean $M, N \in \Lambda_0$ términos cerrados. Decimos que M es simulado y mejorado por N, $M \succeq_\Lambda N$, si y solo si:*

$$[\![M]\!]_{\mathbb{N}} = (V, m) \implies [\![N]\!]_{\mathbb{N}} = (W, n) \wedge V \succeq_{\mathcal{V}} W \wedge (m \geq n)$$

Sean $V, W \in \mathcal{V}_0$ dos valores cerrados. Decimos que V es simulado y mejorado por W, $V \succeq_{\mathcal{V}} W$, si y solo si para todo valor cerrado U:

$$V\, U \succeq_\Lambda W\, U$$

Ahora es fácil de ver que el Ejemplo 5.2.1 es el resultado de aplicar una función es en realidad una simulación de mejora. Sea T un término, x una variable, tales que $T \in \Lambda_{\{x\}}$, y W un valor cerrado, tenemos que:

$$(\boldsymbol{\lambda}\ x\ .\ T)\, W \succeq_\Lambda T[x := W]$$

Lo que nos queda entonces por probar es que la simulación de mejoras implica que los términos realmente son una mejora. En otras palabras que $(\succeq_\Lambda) \subseteq (\rhd)$.

Veamos entonces primero como dar una definición de la simulación ($\succeq_\Lambda$) mediante un operador de relaciones. Si prestamos atención a la definición de simulación aplicativa (Definición 5.3.1) nos encontramos con dos relaciones que son mutuamente recursivas. Para ver que estamos definiendo una relación con sentido la definimos como el mayor punto fijo del siguiente par de operadores.

Definición 5.3.2 (Operador de Mejoras)**.** *Definimos el operador de mejoras sobre términos y valores del lenguaje* $(\Lambda, \mathcal{V})$ *como un par de operadores entre relaciones de términos:* $(\langle\![\_]\!\rangle_\Lambda, \langle\![\_]\!\rangle_\mathcal{V})$*. Donde dada una relación* $R \subseteq \Lambda \times \Lambda$*, tenemos las siguientes dos relaciones:*

- *Dos términos cerrados* $M, N \in \Lambda_0$*,* $M \langle\![R]\!\rangle_\Lambda N$ *si y solo si:*

$$[\![M]\!]_\mathbb{N} = (V, m) \implies [\![N]\!]_\mathbb{N} = (W, n) \wedge (m \geq n) \wedge (V \ \langle\![R]\!\rangle_\mathcal{V} \ W)$$

- *Dos valores cerrados* $V, W \in \mathcal{V}_0$*,* $V \langle\![R]\!\rangle_\mathcal{V} W$ *si y solo si:*

$$\forall U \in \mathcal{V}_0, V\ U\ R\ W\ U$$

El operador de mejoras define paso a paso lo que hace una simulación. Compara la evaluación de dos términos hasta encontrar sus valores y el costo necesario para computarlos, los cuales tiene que estar a su vez extensionalmente relacionados. El operador de mejoras está relacionado directamente con el operador de aproximación aplicativa (Definición 4.2.4).

Lema 5.3.1 (Sands 1991, Def. 2.5)**.** *El par de operadores de mejoras* $(\langle\![\_]\!\rangle_\Lambda, \langle\![\_]\!\rangle_\mathcal{V})$ *es monótono.*

Una relación R es una simulación aplicativa de mejoras si $R \subseteq \langle\![R]\!\rangle$. La simulación de mejoras (Definición 5.3.1) es de hecho una simulación. Más aún, dado que los conjuntos ordenados por la inclusión definen una reticulado completo, y el operador de mejoras es monótono, podemos definir la simulación de mejoras como la unión de todas las relaciones de mejoras.

Todavía nos falta mostrar la conexión entre la simulación de mejoras y la relación de mejoras, que en el caso de no poder hacerlo inutilizaría todo nuestro intento de definir las simulaciones de mejoras. Es decir, mostrar que dados dos términos $M, N \in \Lambda_0$ tal que $M \succeq N$, tenemos que $M \succ N$. Por suerte, ya tenemos un método que conecta estas dos definiciones: el método de Howe (Sección 4.3).

Teorema 5.3.1. *La relación de simulación de mejoras está contenida en la relación de mejoras:* $(\succeq_\Lambda) \subseteq (\rhd)$

De esta manera tenemos como resultado una teoría de mejoras para un lenguaje sin efectos y un método que nos permite probar mejoras: basta con probar que hay una simulación que muestre la mejora.

Capítulo 6

Efectos

Hasta el momento hemos desarrollado la teoría de mejoras para lenguajes **sin** manejo explícito de *efectos*[1]. Es decir, lenguajes que tienen las construcciones clásicas del cálculo lambda, valores, aplicaciones y abstracciones. Como podemos observar a lo largo de los capítulos anteriores, el único efecto computacional observado al momento de la evaluación de términos es el de terminación (o divergencia) de la misma. Sin embargo no es así como escribimos programas en lenguajes de programación, en particular, se suele escribir programas interactivos con el usuario o el entorno.

Los efectos computacionales más comunes son, por ejemplo, la interacción con el entorno, es decir, con dispositivos de entrada y salida, el acceso a memoria, como ser referencias en el lenguaje C, e incluso el manejo de errores y excepciones.

Los lenguajes funcionales buscan minimizar la presencia de efectos computaciones, ya que se consideran una gran fuente de problemas (Hughes 1989). El objetivo principal de los programas es retornar un valor al ser ejecutado, y como *efectos secundarios* (*side effects*) simplemente todo otro efecto computacional que la ejecución utilice. De esta manera, podemos pensar funciones que se concentran en el objetivo principal de los programas: obtener un valor a partir de la entrada estipulada.

En realidad no es que se minimicen los efectos secundarios sino que se busca que estos sean explícitos, por ejemplo, en el lenguaje de programación Haskell esto se logra explicitando los efectos en los tipos de las funciones (Moggi 1989). El argumento a favor es que el desarrollo de los programas se basa en incorporar efectos a medida que se necesitan y no que estén directamente en el lenguaje haciendo que el razonamiento sobre los programas sea más complicado.

A los lenguajes funcionales puros, lenguajes que no manejan efectos dentro del lenguaje, se les puede dar una denotación directa con una entidad matemática. Es decir, a los lenguajes funcionales puros se les puede definir una semántica denotacional clara y precisa, donde los programas se interpretan simplemente como funciones. Por otro lado, los lenguajes con efectos presentan programas que necesitan interactuar con el entorno de manera directa, como el caso del lenguaje imperativo C que permite acceder a la memoria del computador (fuera del entorno de los programas), o que necesitan información externa, como ser

[1] Siempre podemos manejar la representación de efectos de forma pura. Por ejemplo, codificando estructuras de datos dentro del cálculo lambda.

medir la radiación de una partícula para obtener números aleatorios. Esto último hace que los programas dentro de este tipo de lenguajes sean difíciles de modelar, y por ende, también lo sea razonar sobre ellos.

A su vez, tener programas sin efectos, lenguajes de programación **totalmente puros** vuelve dichos lenguajes totalmente inutilizables. Por ejemplo, en general queremos tener programas que sean interactivos, que acepten entradas del usuario y devuelvan una respuesta. La respuesta del lenguaje Haskell a esta problemática fue la de incorporar una mónada especial, llamada *IO*, capaz de interactuar con el entorno de ejecución.

El enfoque que tomamos en esta tesis es diferente. Para simplificar el estudio de los efectos, tomamos un enfoque más restrictivo siguiendo el camino de lo que se denomina efectos algebraicos. Los efectos algebraicos definen un conjunto de operadores capaces de *generar* efectos y estos efectos a su vez son interpretados durante la ejecución. De esta manera se restringen los efectos que puede tener el lenguaje y a su vez se determina de manera explícita como estos son interpretados.

A modo de ejemplo, introducimos el efecto de *no-determinismo* al lenguaje vehículo (Definición 3.1.1), simplemente agregando al lenguaje un nuevo operador binario de forma tal que al evaluarlo es equivalente a evaluar alguno de sus argumentos de forma no determinista.

$$M, N ::= \ldots \mid \oplus(M, N)$$

La evaluación de términos que utilicen el operador binario $\oplus$ ya no serán valores tradicionales sino que serán valores con *efectos*. En este caso, con el efecto de no-determinismo.

En éste capítulo introducimos los conceptos básicos para agregar efectos para un conjunto de lenguajes de programación funcionales, y en particular mostraremos tres ejemplos: excepciones, no-determinismo, y términos probabilísticos. En concreto, parametrizamos el lenguaje vehículo, es decir, el lenguaje tendrá como parámetro los operadores que generan efectos, y así obtenemos un lenguaje general capaz de aceptar más efectos que los ejemplificados. De esta manera, permitimos que el lenguaje acepte *efectos algebraicos*, un subconjunto (propio) de los efectos computacionales cuyas propiedades nos ayudará a derivar una noción de equivalencia observacional que eventualmente utilizaremos para obtener teorías de mejoras en los capítulos que restan. Además obtenemos una evaluación similar a la que definimos para el lenguaje vehículo, lo que nos permite desarrollar una teoría de mejoras utilizando las herramientas vistas.

6.1. Lenguajes con efectos

Introducir efectos computacionales mediante operadores del lenguaje es una técnica conocida como *efectos algebraicos* (Plotkin y Power 2003). Entre los operadores (y efectos) más usuales encontramos: $\mathsf{put}, \mathsf{get}$ para obtener un estado mutable, $\mathsf{read}, \mathsf{print}$ para obtener entrada/salida, $\mathsf{raise}(e)$ para lanzar la excepción e, $\oplus_p$ como un operador probabilístico, entre otros. En lo que resta de la tesis utilizaremos un lenguaje vehículo parametrizado en los operadores que introducen efectos algebraicos. De esta manera podemos desarrollar un teoría general que acepta diferentes efectos algebraicos. En concreto, en esta sección vemos diferentes efectos algebraicos con dos objetivos: 1) desarrollar

intuición sobre los mismos, y 2), presentar los ejemplos que utilizaremos a lo largo de la tesis para obtener teorías de mejoras. En particular estudiamos como introducir 3 diferentes efectos: excepciones, no-determinismo, y probabilidad. Además vemos el efecto de computaciones con fallo que es el caso más sencillo, y se interpreta de manera similar al efecto de excepciones. Finalmente, al observar los ejemplos podremos abstraernos de las diferencias y presentar el caso general.

Lenguaje Fallo

En esta sección, introducimos un nuevo operador constante (sin argumentos), fail, donde el efecto que genera es la terminación temprana de la evaluación. En otras palabras, al momento de evaluar el término fail, el objetivo es que la evaluación se detenga y se identifique como un fallo. Este efecto es muy utilizado para identificar el comportamiento anómalo (o de fallo) del sistema.

Tener un elemento que genere un fallo dentro del lenguaje es muy útil, y como caso de ejemplo podemos observar el lenguaje de programación *Haskell* (Marlow et al. 2010). En Haskell, disponemos de un término (**error** :: **String** −> a), que al momento de ser evaluado finaliza el proceso de evaluación. También tenemos el caso muy útil de las condiciones *asserts* en la mayoría de los lenguajes imperativos como C.

Extendemos el lenguaje vehículo del Capítulo 3 con un nuevo operador, fail, y presentamos un nuevo lenguaje con dos categorías sintácticas. A modo de recordatorio presentamos nuevamente el lenguaje completo adicionando un nuevo término:

Definición 6.1.1 (Lenguaje con fail)**.** *Dado un conjunto numerable* $\mathbb{V}$ *de variables.*

Definimos entonces el conjunto de valores $\mathcal{V}^{\perp}$ *como:*

$$V, W ::= x \mid (\boldsymbol{\lambda}\ x\ .\ M)$$

Y el conjunto de términos $\Lambda^{\perp}$ *como:*

$$M, N ::= \mathbf{ret}(V) \mid (V\ W) \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid \mathsf{fail}$$

El nuevo lenguaje con el operador de fallo fail lo notamos como $(\Lambda^{\perp}, \mathcal{V}^{\perp})$.

Veamos entonces como evaluar esta versión extendida del lenguaje incorporando el efecto del operador fail. Es necesario ver que a la relación de evaluación debemos agregarle un nuevo elemento distinguido para identificarlo con el resultado de la evaluación del termino fail. Siguiendo el mismo camino que en los capítulos anteriores, definimos una familia de relaciones que aproximan la evaluación de los términos. Primero definimos el sistema de ecuaciones que queremos que la relación de evaluación cumpla, y luego, presentaremos una solución general en la siguiente sección.

Originalmente, la familia de relaciones del Capítulo 3 (Definición 3.3.1) define relaciones entre términos cerrados y valores cerrados más un elemento adicional, $(\Downarrow_n) \subseteq \Lambda_0 \times (\mathcal{V}_0 + \perp)$ con $n \in \mathbb{N}$. El elemento adicional $\perp$ indica que no es posible aproximar un término con la relación $\Downarrow_n$. Lo que deriva en que la relación de evaluación $[\![-]\!] \subseteq \Lambda_0 \times (\mathcal{V}_0 + \perp)$ sea, en realidad, una función total que cuando relaciona un término N con $\perp$ indica que no se le puede asignar un valor al término N mediante $[\![-]\!]$.

Ahora, ¿con qué elemento debemos emparejar la evaluación del término cerrado fail? Naturalmente, surge pensar en equipar el nuevo término fail con el elemento distinguido $\bot$. Pero, no es lo mismo indicar que la evaluación de un término falla por fail a que la relación de evaluación no puede asignarle un valor. De hecho, al evaluar el término fail no queremos que la evaluación falle, sino que termine abrupta pero correctamente. Por ejemplo, no es lo mismo evaluar el término fail que el término Ω: el efecto esperado al evaluar el término fail es el de terminar la ejecución, mientras que la evaluación del término Ω no debería terminar. Para interpretar el efecto del término fail introducimos un nuevo elemento en los posibles resultados de la relación de evaluación. Notar que no es la evaluación la que falla, en todo caso es exitosa, el objetivo de introducir un nuevo término, fail, es el de agregar al lenguaje un operador que permite finalizar de forma temprana (y abrupta) la evaluación de los términos y se le otorga una semántica de fallo.

Lo que haremos entonces es agregar un nuevo componente que indica que la evaluación termina (correctamente) pero no se le asocia ningún valor en $\mathcal{V}_0$. Siguiendo el camino recorrido en la Sección 3.3, podríamos definir una familia de relaciones con el objetivo final de encontrar una solución al sistema de ecuaciones. En este caso, nos vamos a saltar el proceso detallado ya que es similar al visto en la Sección 3.3, y vamos a presentar el sistema de ecuaciones. En la Sección 6.3 vemos como podemos dar un sistema de ecuaciones que caractericen la evaluación de términos con efectos de forma general. El objetivo es tener una relación de evaluación de forma tal que para cada $T \in \Lambda_0^\bot$:

$$[\![T]\!]^\bot \in (\mathcal{V}_0^\bot + \bot) + \bot$$

Y podemos definirla como la menor relación que cumpla las siguientes ecuaciones:

$$\begin{aligned}
[\![\mathbf{ret}(V)]\!]^\bot &= \iota_l(\iota_l(V)) \\
[\![\mathsf{fail}]\!]^\bot &= \iota_l(\iota_r(\bot)) \\
[\![(\boldsymbol{\lambda}\ x\ .\ M)\, W]\!]^\bot &= [\![M[x := W]]\!]^\bot \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]^\bot &= \begin{cases} \iota_r(\bot) & \text{si } [\![M]\!]^\bot = \iota_r(\bot) \\ \iota_l(\iota_r(\bot)) & \text{si } [\![M]\!]^\bot = \iota_l(\iota_r(\bot)) \\ [\![N[x := V]]\!]^\bot & \text{si } [\![M]\!]^\bot = \iota_l(\iota_l(V)) \end{cases}
\end{aligned}$$

De esta manera tenemos que para un término cerrado $M \in \Lambda_0^\bot$ se le asigna uno de los siguientes valores:

- $[\![M]\!]^\bot = \iota_r(\bot)$ indicando que no es posible asignarle un valor a la evaluación del término M,
- $[\![M]\!]^\bot = \iota_l(\iota_r(\bot))$ indicando que el operador fail fue evaluado durante la ejecución produciendo que la evaluación termine abruptamente y no se pueda retornar un valor,
- $[\![M]\!]^\bot = \iota_l(\iota_l(V))$ indicando que se le asigna el valor V a la evaluación del término M

Lenguaje con Excepciones

En esta sección introducimos *excepciones* al lenguaje como un efecto. El modelo es similar al efecto de terminar la computación con el operador fail, pero a diferencia de simplemente fallar, en este caso queremos discernir entre diferentes posibles fallos. Por ejemplo, quisiéramos definir casos donde no tiene sentido continuar la evaluación de un término por diferentes razones: división por cero, errores inesperados, fallos en la conexión con diferentes dispositivos, etc.

Dado un conjunto de elementos E, agregamos un conjunto de operadores al lenguaje raise_e con $e \in E$. Un nuevo operador por cada fallo posible, y notamos al lenguaje como $(\Lambda^E, \mathcal{V}^E)$. De esta manera es posible definir de forma genérica (en E) un conjunto de excepciones, una por cada elemento de E, permitiendo entonces identificar los diferentes fallos durante la evaluación. Para indicar que la evaluación de un término lleva a una excepción $e \in E$, lanzada por el operador raise_e, debemos ampliar los posibles resultados de forma similar al momento de introducir la noción de fallo. En este caso, como debemos llevar información de etiquetas en vez de simplemente introducir un elemento $\bot$ para indicar que la evaluación exitosa lleva a un error, lo que haremos es introducir directamente el conjunto E como posibles elementos. En símbolos, para cada término cerrado $M \in \Lambda_0^E$, tenemos $[\![M]\!]^E \in (\Lambda_0 + E) + \bot$ [2].

Definimos entonces la relación de evaluación como la menor relación que resuelva el siguiente sistema de ecuaciones:

$$
\begin{aligned}
[\![\mathbf{ret}(V)]\!]^E &= \iota_l(\iota_l(V)) \\
[\![\mathsf{raise}_e]\!]^E &= \iota_l(\iota_r(e)) \\
[\![(\boldsymbol{\lambda}\ x\ .\ M)\, W]\!]^E &= [\![M[x := W]]\!]^E \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]^E &= \begin{cases} \iota_r(\bot) & \text{si } [\![M]\!]^E = \iota_r(\bot) \\ \iota_l(\iota_r(e)) & \text{si } [\![M]\!]^E = \iota_l(\iota_r(e)) \\ [\![N[x := V]]\!]^E & \text{si } [\![M]\!]^E = \iota_l(\iota_l(V)) \end{cases}
\end{aligned}
$$

Es fácil ver la similitud con la relación de evaluación con el efecto de fallo, donde simplemente había una única "excepción" que indicaba la detección del término fail. En éste caso podemos modelarlo directamente simplemente asumiendo que hay un único fallo posible, es decir: $E = \{\mathsf{fail}\}$.

Asumiendo que tenemos dentro del lenguaje operadores de aritmética básica, y los operadores de comparación, podemos dar un sencillo ejemplo. Tomando $E = \{\texttt{zero\_div}, \ldots\}$, podemos implementar una división segura de la siguiente forma:

$$\texttt{safe\_div} \doteq \boldsymbol{\lambda}\ (x, y)\ .\ \mathbf{ifz}(y, \mathbf{ret}(\mathbf{div}\, x\, y), \mathsf{raise}_{\texttt{zero\_div}})$$

Lenguaje con No-Determinismo

En este caso introduciremos un operador binario *no determinístico*, $\oplus$, de forma tal que dados dos términos cerrados M y N, el término cerrado $\oplus(M, N)$ al ser evaluado se comporta como M o N de forma no-determinística. Notamos a este nuevo lenguaje como $(\Lambda^{ND}, \mathcal{V}^{ND})$.

[2] Notar que es equivalente a introducir $\#E$ elementos diferentes, con $E \neq \emptyset$, $\forall A, A + E \cong A + \#E \times \bot$

Existen varias formas de modelar el comportamiento de este operador (Antoy 1997; Søndergaard y Sestoft 1992). Una forma sencilla es pensar que al momento de evaluar el término $\oplus(M, N)$, lógicamente, el programa puede tomar dos caminos, continuar con la evaluación de M o con la evaluación de N. De esta manera, podemos pensar la evaluación de un término ya no es una secuencia de instrucciones sino un *árbol binario* donde cada nodo representa una bifurcación introducida por la presencia del operador $\oplus$. Incluso, a medida que la evaluación avanza, nos alcanza simplemente con tener el conjunto de los posibles expresiones hasta el momento, uno por cada posible camino que la evaluación pueda llegar a tomar. De esta manera el no-determinismo significa simplemente no saber cuales de todas las posibles hojas del árbol de posibles valores la evaluación de un término puede retornar, por lo que se exploran todas a la vez.

Definimos entonces a la evaluación de términos con el operador de no-determinismo como la menor relación que cumpla con las siguientes ecuaciones:

$$\begin{aligned}
[\![\mathbf{ret}(V)]\!]^{ND} &= \{V\} \\
[\![\oplus(M, N)]\!]^{ND} &= [\![M]\!]^{ND} \cup [\![N]\!]^{ND} \\
[\![(\boldsymbol{\lambda}\ x\ .\ M)\ W]\!]^{ND} &= [\![M[x := W]]\!]^{ND} \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]^{ND} &= \bigcup_{V \in [\![M]\!]^{ND}} [\![N[x := V]]\!]^{ND}
\end{aligned}$$

Vemos que la evaluación de un término $\oplus(M, N)$ se define como la unión de los posibles valores que puedan tomar las evaluaciones de M y N. Lo mismo en el caso del término de ligadura **let** $x = M$ **in** ..., donde se unen todos los posibles caminos que se pueden tomar partiendo de cada uno de los posibles valores que puede tomar la evaluación de M.

Si comparamos el sistema de ecuaciones con los dos anteriores, podemos notar que la definición de la aplicación es la misma, simplemente se realiza la substitución necesaria para continuar la evaluación y que la evaluación del término **ret**(V) es simplemente presentar a la evaluación con una forma de devolver un valor. Mientras que la evaluación del término **let** $x = M$ **in** N primero obtiene el conjunto de posibles valores que puede tomar la evaluación de M, para recolectar todos los posibles valores de los resultados de continuar con la evaluación de N con los posibles valores que puede tomar x. De forma similar, las funciones de evaluación para los lenguajes con fallo y excepciones reducen el término **let** $x = M$ **in** N: primero obtienen un valor a partir de M y continuar evaluando, en el caso que lo hubiera, y sino, propagar el fallo o la excepción.

Lenguaje Probabilístico

Una forma de introducir la noción de probabilidad a los lenguajes de programación es mediante un operador binario probabilístico, $\oplus_p$, de forma tal que dados dos términos M, N, la evaluación de $\oplus_p(M, N)$ es equivalente a la evaluación de M con probabilidad p o a la evaluación de N con probabilidad $(1 - p)$. Al nuevo lenguaje con el operador probabilístico lo notamos como $(\Lambda^D, \mathcal{V}^D)$

Introducir probabilidades en la evaluación de términos es complicado y representa un problema abierto en el área de teoría de lenguajes de programación (similar al caso de no-determinismo). Una forma de hacerlo es identificar la evaluación de términos con distribuciones de probabilidades, de forma similar que con no determinismo donde se relacionan términos con un conjunto de posible valores, en este caso relacionaremos términos con una distribución de probabilidades. La distribución de probabilidad que se relaciona en la evaluación de un término indica para cada valor la probabilidad que sea el resultado final. Hay varias formas de introducir probabilidades en la evaluación de términos (Ramsey y Pfeffer 2002).

Debido a que hay términos que no aceptan valores, por ejemplo Ω, utilizamos *sub-distribuciones* de probabilidades para capturar *ambos efectos*: divergencia y evaluación probabilística. Representando las sub-distribuciones como funciones del dominio de aplicación a la probabilidad de cada elemento, podemos extender una noción de orden entre entre dos distribuciones.

Una distribución de probabilidades en un conjunto X la representamos como una función $\delta : X \to [0, 1]$ de forma tal que la suma de la probabilidad de todos los elementos en X suman 1, $\delta(X) = 1$. Una *sub-distribución* de probabilidades sobre un conjunto X es una función $\delta : X \to [0, 1]$ de forma tal que $\delta(X) \leq 1$. Al usar sub-distribuciones de probabilidades, no sólo nos permite capturar fácilmente la idea de que *no haya* un valor al cual asignarle la evaluación de un término, sino que además, nos permite comparar distribuciones siguiendo una idea similar de Scott, como funciones que están *más* definidas que otras.

Por la forma en que evaluamos los términos, nos bastará con quedarnos con la sub-distribuciones de probabilidades con conjunto soporte numerable, es decir, que la sumatoria de las probabilidades está bien definida. En símbolos, definimos para cada conjunto X, el conjunto de todas las sub-distribuciones de probabilidad en X:

$$\mathbb{D}(X) = \{\delta \mid \text{soporte}(\delta) \text{ es numerable y } \delta \text{ es una sub-distribución}\}$$

Definimos entonces la evaluación de términos probabilísticos como la menor relación que cumpla las siguientes ecuaciones:

$$\begin{aligned}
[\![\mathbf{ret}(V)]\!]^{\mathbb{D}} &= const(V) \\
[\![\oplus_p(M, N)]\!]^{\mathbb{D}} &= join(p, [\![M]\!]^{\mathbb{D}}, [\![M]\!]^{\mathbb{D}}) \\
[\![(\boldsymbol{\lambda}\ x\ .\ M)\, W]\!]^{\mathbb{D}} &= [\![M[x := W]]\!]^{\mathbb{D}} \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]^{\mathbb{D}} &= probLet(M, x, N)
\end{aligned}$$

Donde utilizamos las funciones auxiliares $const, join, probLet$ que definiremos a continuación. Podemos ver que la evaluación de la aplicación se mantiene nuevamente intacta, mientras que para los valores, la aplicación y los términos **ret** las definiciones cambian.

La familia de funciones *const* define una distribución de probabilidades constante que retorna 1 en un valor específico y 0 es cualquier otro punto.

$$const : \mathcal{V}_0 \to \mathbb{D}(\mathcal{V}_0)$$
$$const(V)(W) = \begin{cases} 1 & \text{si } V = W \\ 0 & \text{sino} \end{cases}$$

La función *join* toma dos sub-distribuciones para combinarlas ajustando las probabilidades con un parámetro de entrada.

$$join : ([0,1] \times \mathbb{D}(X) \times \mathbb{D}(X)) \to \mathbb{D}(X)$$
$$join(p, \delta, \mu)(x) = p * \delta(x) + (1-p) * \mu(x)$$

La función *probLet* es un operador que, utilizando la relación de evaluación, combina las sub-distribuciones de probabilidades siguiendo la misma idea que en las ecuaciones anteriores: busca *secuenciar* la evaluación, primero computa las sub-distribución de M, y multiplicando por $[\![M]\!]^D(w)$ la sub-distribución resultante de la evaluación de N cuando x toma el valor w.

$$probLet : (\Lambda_0 \times \mathbb{V} \times \mathcal{V}_0) \to \mathbb{D}(\mathcal{V}_0)$$
$$probLet(M, x, N)(V) = \sum_{W \in \mathcal{V}_0} ([\![M]\!]^{\mathbb{D}}(W) * [\![N[x := W]]\!]^{\mathbb{D}}(V))$$

Al definir la evaluación como sub-distribuciones de probabilidad con soporte numerable, las sumatorias por mas que sean en todo el espacio de valores cerrados se pueden acotar al conjunto de soporte de la distribución.

Para terminar las pruebas deberíamos verificar que efectivamente la relación de evaluación manipula sub-distribuciones de probabilidad, y en particular, deberíamos probar que la función *join* toma dos sub-distribuciones de probabilidades y retorna otra, mientras que la función *probLet* dado un término cerrado, una variable, y un valor cerrado, retorna una sub-distribución de probabilidad. Es fácil ver que la función *const* define una sub-distribución para cada valor $v \in \mathcal{V}_0$, también conocida como función delta (δ). La función *join* también es una función que toma dos sub-distribuciones, devolviendo otra, por simple aritmética. Mientras que la función *probLet* requiere un poco más de manipulaciones aritméticas, pero se desprende que la evaluación de un término es una sub-distribución de probabilidades, y por lo tanto, la sumatoria está acotada por 1. Por el momento no vamos a realizar dichas pruebas, ya que en la siguiente sección introduciremos un mecanismo que simplificará en gran medida las manipulaciones matemáticas necesarias.

6.2. Enfoque Genérico

Basándonos en los ejemplos antes vistos, podemos observar que para introducir efectos lo primero que necesitamos es introducir las operaciones que, al momento de ser evaluadas, generan efectos. Para esto introducimos un mecanismo que nos permite agregar operadores al lenguaje definido en el Capítulo 3 mediante la noción de signatura (Definición 2.2.1).

Para la construcción del lenguaje utilizaremos signaturas con un solo sort, ya que nuestro objetivo es utilizar los operadores como constructores de un lenguaje sin tipos, es decir, operadores que toman como argumento términos del lenguaje. En el caso que quisiéramos aumentar el lenguaje con una noción de tipos, para distinguir entre diferentes construcciones del lenguaje, podríamos utilizar entonces signaturas con múltiples sorts.

A modo de recordatorio, una signatura Σ con un solo sort es un conjunto A equipado con una función de aridad $\alpha \colon A \to \mathbb{N}$, y lo notamos como $\langle A, \alpha \rangle$.

Ejemplo 6.2.1. *Repasemos entonces cada uno de los efectos introducidos hasta el momento: fallo, excepciones, no-determinismo y probabilidades.*

Fallo. *Podemos introducir la signatura* $\langle\{\mathsf{fail}\}, \mathsf{fail} \mapsto 0\rangle$ *que define un operador* fail *sin argumentos, cuya semántica es indicar el fallo en la evaluación de un término.*

Excepciones. *Dado un conjunto* E *de excepciones, podemos definir la siguiente signatura* $\langle\{\mathsf{raise}_e \mid e \in E\}, \{\mathsf{raise}_e \mapsto 0 \mid e \in E\}\rangle$ *introduciendo una familia de operadores sin argumentos al lenguaje cuya semántica es la de lanzar una excepción.*

No-Determinismo. *Introducimos la signatura* $\langle\{\oplus\}, \oplus \mapsto 2\rangle$ *que define un operador binario* $\oplus$*, y cuya semántica es que la evaluación de* $M \oplus N$ *se puede comportar como* M *o* N *de forma no determinista.*

Operadores Probabilísticos. *Mediante la siguiente signatura definimos una familia de operadores* $\langle\{\oplus_p \mid p \in [0,1]\}, \{\oplus_p \mapsto 2 \mid p \in [0,1]\}\rangle$ *, de forma tal que para cualquier* $p \in [0,1]$ *define un operador binario,* $M \oplus_p N$*, cuya evaluación se comporta como* M *con probabilidad* p *o como* N *con probabilidad* $(1-p)$*.*

Utilizando signaturas de un único sort podemos parametrizar el lenguaje para poder introducir operadores fácilmente, simplemente definiendo cada uno de los operadores de la signatura como un constructor del lenguaje.

Definición 6.2.1 (Lenguaje con efectos algebraicos Σ)**.** *Sea* Σ *una signatura y* $\mathbb{V}$ *un conjunto numerable de nombres de variables.*

Definimos el conjunto de valores $\mathcal{V}_\Sigma$ *como:*

$$V, W ::= x \mid (\boldsymbol{\lambda}\ x\ .\ M)$$

Definimos el conjunto de términos Λ_Σ*:*

$$M, N ::= \mathbf{ret}(V) \mid (V\ W) \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid \sigma(M_1, M_2, \ldots, M_n)$$

Con $\sigma \in \Sigma$ *y* $\alpha(\sigma) = n$*.*

De esta manera lo que logramos es parametrizar el lenguaje por una signatura, y es la signatura la que define los operadores y sus argumentos. Luego, podremos definir la evaluación de los términos de la misma forma que hicimos con los diferentes efectos en la sección anterior.

La principal, y única diferencia, con el lenguaje introducido en el Capítulo 3 es la adición del mecanismo necesario para introducir nuevos términos en el lenguaje. En el caso que no haya constructores, $\Sigma = \emptyset$, obtenemos el lenguaje puro.

Podemos ver, que utilizando las signaturas del Ejemplo 6.2.1, obtenemos cada uno de los lenguajes introducidos a modo de ejemplo en las secciones anteriores.

Notación 6.2.1. *Dada una signatura* Σ*, notamos al conjunto de términos del lenguaje con efectos algebraicos en* Σ *como* Λ^Σ*, y al conjunto de valores como* $\mathcal{V}^\Sigma$*. Notaremos directamente al conjunto de términos como* Λ *y valores como* $\mathcal{V}$*, a menos que sea necesario aclararlo.*

De esta manera podemos agregar los operadores algebraicos necesarios que son los que generan los efectos durante la evaluación de términos. Lo que nos falta para completar este esquema es la interpretación de los efectos generados por la evaluación de términos. En la Sección 6.3 definimos la evaluación de lenguajes con operadores algebraicos definidos por una signatura. Para esto, utilizamos mónadas (Moggi 1989) para interpretar los efectos computacionales generados por las evaluación de los operadores algebraicos (Plotkin y Power 2003).

6.3. Evaluación con Efectos

Para la evaluación de términos del lenguaje seguiremos los mismos pasos que en los capítulos anteriores, pero en este caso, necesitaremos una forma de interpretar los efectos generados por los operadores que son introducidos. A diferencia de la evaluación para lenguajes sin efectos algebraicos presentada en el Capítulo 3, ahora tendremos nuevos operadores. Estos operadores, al evaluarlos generarán algún efecto, y por lo tanto, necesitamos algún mecanismo que los interprete. Lo que vamos a hacer entonces es introducir los efectos dentro de los evaluadores que ya definimos. En otras palabras, continuamos definiendo la evaluación de términos como una familia de aproximaciones, pero esta vez deberemos tener en cuenta cómo son interpretados los nuevos efectos generados por los operadores introducidos. Para interpretar los efectos utilizamos la noción clásica de mónadas (Moggi 1989; Plotkin y Power 2003), pero lo adaptaremos a la evaluación aproximada mediante una familia de aproximaciones (Definición 3.3.1) siguiendo el camino trazado en el Capítulo 3.

Recapitulando. Para el lenguaje sin efectos algebraicos (Capítulo 3), teníamos que la función de evaluación de términos del lenguaje quedaba definida por el supremo de la cadena de los valores aproximados dados por una familia de relaciones $(\Downarrow_n)$. Para cada $T \in \Lambda_0$ teníamos que:

$$[\![T]\!] \doteq \sup_{n\in\mathbb{N}}\{x_n \mid T \Downarrow_n x_n\}$$

La familia de relaciones está compuesta por relaciones indexadas por un número natural $n \in \mathbb{N}$, de forma tal que $(\Downarrow_n) \subseteq \Lambda_0 \times (\mathcal{V}_0 + \bot)$. Dado un término $T \in \Lambda_0$ y un número natural n, tenemos que $T \Downarrow_n \iota_l(V)$ si existe un árbol de evaluación de altura menor a n que relacione T con V, y a T con $\iota_r(\bot)$ en caso que la profundidad n sea insuficiente.

Nuestra tarea es obtener una función de evaluación que además incorpore la evaluación de efectos algebraicos. Esto lo logramos introduciendo la noción de *álgebra continua* (Goguen et al. 1977), que conecta la noción de evaluación aproximada con la interpretación de los operadores que se introduzcan al lenguaje, y además equipamos a la mónadas con una noción de orden para poder tener ω-cadenas de valores con efectos (Dal Lago, Gavazzo y Paul Levy 2017). Es decir, extendemos la noción de continuidad y orden que utilizamos para definir la evaluación de términos para interpretar a los operadores que introducen efectos.

Definición 6.3.1 (Álgebras Continuas)**.** *Sea D un dominio y Σ una signatura. Una Σ-álgebra $(D, (\cdot)^D)$ es una álgebra continua si y solo si cada símbolo $\sigma \in \Sigma$*

es interpretado como una función continua σ^D en cada uno de sus argumentos respecto al dominio D.

Esta definición lo que hace es equipar a los generadores de efectos (como son interpretados los operadores de la signatura) utilizados durante la evaluación de términos en el lenguaje con funciones que respetan la semántica de sus argumentos. La noción de respetar la evaluación semántica es lo que se caracteriza con la noción de continuidad. En otras palabras, los operadores algebraicos serán interpretados como funciones continuas dentro del dominio semántico de valores con efectos.

Para interpretar los efectos generados por las operaciones algebraicas utilizamos el concepto de mónada.

Definición 6.3.2 (Mónada)**.** *Sea $\mathcal{C}$ una categoría. Una tripleta (T, μ, η) es una mónada en la categoría $\mathcal{C}$ si y solo si T es un endofuntor equipado con dos transformaciones naturales $\eta_X : X \to T(X)$ y $\mu_X : T^2(X) \to T(X)$ de forma tal que los siguientes diagramas conmutan:*

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ {\scriptstyle \mu T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array} \qquad \begin{array}{ccccc} T & \xrightarrow{T\eta} & T^2 & \xleftarrow{\eta T} & T \\ & \searrow^{=} & \downarrow{\scriptstyle \mu} & \swarrow_{=} & \\ & & T & & \end{array}$$

En la tesis solo utilizamos la categoría de conjuntos, ya que nos concentraremos en construir una relación de mejoras entre términos. Por lo tanto, en esta tesis las mónadas en realidad consisten en endofuntores sobre la categoría de conjuntos. Más aún, podemos definir el operador $(>\!>\!=)_X : T(X) \to (X \to T(Y)) \to T(Y)$, llamado **bind**, como:

$$mx >\!>\!= f = \mu(T(f)(mx))$$

El operador **bind** es equivalente en la categoría de conjuntos a la transformación natural μ. Dentro de los lenguajes de programación es más intuitivo el uso del operador **bind** ya que nos permite introducir la noción de secuencia de efectos.

El siguiente paso es equipar a las mónadas con la estructura necesaria para poder construir ω-cadenas de elementos con efectos.

Definición 6.3.3 (Mónadas Ordenadas)**.** *Dada una mónada T, un* orden sobre *T consiste en una familia de relaciones $(\sqsubseteq)$ y elementos $\bot$, donde para cada conjunto X:*

- *hay un elemento distinguido $\bot_X \in T(X)$*
- *y una relación de orden $(\sqsubseteq_X) \subseteq T(X) \times T(X)$*

de forma tal que $(T(X), \sqsubseteq_X, \bot_X)$ forma un dominio, y el operador **bind** *de la mónada T es continuo en ambos argumentos:*

$$\bigsqcup_{n<\omega} (x >\!>\!= f_n) = x >\!>\!= (\bigsqcup_{n<\omega} f_n)$$
$$\bigsqcup_{n<\omega} (x_n >\!>\!= f) = (\bigsqcup_{n<\omega} x_n) >\!>\!= f$$

Donde $\{f_n\}_{n<\omega}, \{x_n\}_{n<\omega}$ son ω-cadenas con $f_n \colon X \to T(Y)$ y $x_n \colon T(X)$ para todo $n < \omega$, $x \colon T(X)$ y $f \colon X \to T(Y)$.

En palabras, para todo conjunto la mónada está equipada con un dominio, y más aún, en ese dominio el operador de **bind** es continuo en ambos argumentos. Nuevamente, la noción de continuidad nos está indicando, que durante la evaluación, el operador **bind** respeta el sentido semántico de la evaluación de términos en valores, en este caso, con efectos.

Lo que nos falta finalmente es conectar las álgebras continuas con mónadas ordenadas, y de esta manera interpretar los efectos generados por los operadores dentro de la mónada. En nuestro caso, interpretar operaciones es en realidad interpretar los efectos que estas describen. Por ejemplo, si tenemos operadores que *generan* efectos entrada/salida, queremos interpretarlos como operaciones de entrada/salida en la mónada.

Definición 6.3.4 (Σ-álgebra para una mónada)**.** *Sea Σ una signatura y T una mónada. Una Σ-álgebra para la mónada T es una familia de Σ-álgebras Φ tales que para cada conjunto X, equipa a $T(X)$ con una Σ-álgebra Φ_X.*

En palabras, dada una mónada T equipada con una Σ-álgebra, significa que cada operador σ definido en Σ, tiene una interpretación $(\sigma^{T(X)} \colon T(X)^{\alpha(\sigma)} \to T(X))$ para cada conjunto X. Esta forma universal de equipar a la mónada con un álgebra, tal que todo conjunto tiene una álgebra, nos fuerza a interpretar los efectos de los operadores independientemente del conjunto en sí. En otras palabras, el conjunto subyacente no introduce información alguna, sino que los efectos de los operadores del álgebra son interpretados dentro de la estructura de la mónada.

Definición 6.3.5 (Mónada Σ-continua)**.** *Sea Σ una signatura, T una mónada, y Φ una Σ-álgebra para la mónada T. Decimos que T es Σ-continua, o que lleva una Σ-álgebra continua, si T tiene un orden $(\sqsubseteq)$ tal que para cada conjunto X, Φ_X es una Σ-álgebra continua respecto al orden $(\sqsubseteq_X)$.*

En palabras, dada una signatura Σ, y una mónada T tal que es Σ-continua, entonces para cada conjunto X, $T(X)$ está equipado con un álgebra Σ-continua, o equivalentemente, para cada símbolo $\sigma \in \Sigma, \alpha(\sigma) = n$, su interpretación en $T(X)$ es continua en cada argumento:

$$\sigma^{T(X)}(\ldots, \bigsqcup_{n<\omega} a_n, \ldots) = \bigsqcup_{n<\omega} \sigma^{T(X)}(\ldots, a_n, \ldots)$$

De esta manera nos aseguramos que la interpretación de los operadores mantienen el sentido semántico de sus argumentos.

Finalmente, podemos definir una familia de relaciones indexadas por los números naturales que aproximan a la evaluación de términos en valores, esta vez, con efectos interpretados por una mónada.

Definición 6.3.6. *Sea Σ una signatura y T una mónada Σ-continua. Definimos una familia de relaciones indexadas por un número natural $n \in \mathbb{N}$ entre términos*

cerrados y valores con efectos ($\Downarrow_n^T$) *de la siguiente manera:*

$$\bot\frac{}{M\Downarrow_0^T\bot} \qquad (\texttt{ret})\frac{}{\mathbf{ret}(V)\Downarrow_{n+1}^T\eta(V)}$$

$$(\texttt{seq})\frac{M\Downarrow_n^T X \qquad N[x := V]\Downarrow_n^T Y_V}{\mathbf{let}\ x = M\ \mathbf{in}\ N\Downarrow_{n+1}^T X >\!\!>\!\!= (V \rightsquigarrow Y_V)}$$

$$(\texttt{app})\frac{M[x := W]\Downarrow_n^T X}{(\boldsymbol{\lambda}\ x\ .\ M)\,W\Downarrow_{n+1}^T X}$$

$$(\sigma\texttt{-op})\frac{M_1\Downarrow_n^T X_1 \qquad \ldots \qquad M_k\Downarrow_n^T X_k}{\sigma(M_1,\ldots,M_k)\Downarrow_{n+1}^T \sigma^T(X_1,\ldots,X_k)}$$

Donde la regla (σ-op) *es en realidad un conjunto de reglas, una para cada operador* $\sigma \in \Sigma$ *y* $k = \alpha(\sigma)$.

Las familias de aproximaciones recién definidas determinan como se evalúan los términos. Para cada operador σ en Σ introducimos una regla (σ-op) que relaciona el operador σ y sus argumentos, con la interpretación σ^T y los valores de cada uno de los argumentos.

La regla (seq) expresa que la evaluación del término (**let** $x = M$ **in** N) se reduce a evaluar *primero* el término M y luego una vez encontrado el valor V (en caso de que exista) *continuar* la evaluación por $N[x := V]$, en otras palabras es lo que define el operador de **bind** de las mónadas. La idea de *continuar la evaluación* la expresamos como una función que toma como argumento un valor, computado previamente, y produce un valor resultado con efectos luego de computar la evaluación de un término que puede utilizar el valor que toma como argumento. Esta regla es el precio a pagar por la generalidad de los efectos, en el caso de tener un efecto y una mónada concreta que los interprete la evaluación tendría un enfoque más clásico como se pueden apreciar en las definiciones de los ejemplos presentados al principio del capítulo, lenguaje con fallo, excepciones, no-determinismo, y operadores probabilísticos. Esta regla es central en la evaluación mónadica y establece una conexión directa con el operador **bind**.

Veamos ahora que la familia de evaluaciones nos alcanza para definir la evaluación de lenguajes con efectos algebraicos.

Lema 6.3.1 (Determinismo)**.** *Sea* Σ *una signatura y* T *una mónada* Σ*-continua. Sean* $n < \omega$*,* $M \in \Lambda_0^\Sigma$ *y* $v, w \in T(\mathcal{V}_0^\Sigma)$*, si* $M\Downarrow_n^T v$ *y* $M\Downarrow_n^T w$ *entonces* $v = w$*.*

Demostración. La prueba procede por inducción en n.

Caso Base Con $n = 0$, la única regla que es posible a aplicar es ($\bot$), por lo que $v = \bot = w$.

Paso Inductivo Asumiendo que $n = m + 1$ para algún $m \geq 0$. Procedemos haciendo análisis por casos en la última regla aplicada a $M\Downarrow_n^T v$.

Caso ($\bot$) no es posible dado que $n > 0$.

Caso (`ret`) En este caso, $M = \mathbf{ret}(V)$ para algún $V \in \mathcal{V}_0^\Sigma$, y $v = \eta(V)$. Podemos concluir entonces que $M = \mathbf{ret}(V)\Downarrow_{m+1}^T w$, y al aplicar la *única* regla posible, (`ret`), $w = \eta(V)$. Por lo que $v = w$.

Caso (`app`) Tenemos entonces que $M = (\boldsymbol{\lambda}\, x\, .\, Q)\, W$ con $Q \in \Lambda_{\{x\}}^\Sigma$ y $W \in \mathcal{V}_0^\Sigma$ de forma tal que $Q[x := W]\Downarrow_m^T v$. Ahora, partiendo de $M\Downarrow_n^T w$, por la definición de M y aplicando la regla `app`, tenemos que $Q[x := W]\Downarrow_m^T w$. Finalmente por hipótesis inductiva, con $m < n$, tenemos que $v = w$.

Caso (σ-`op`) En este caso tenemos para un operador $\sigma \in \Sigma$ con $\alpha(\sigma) = k$, $M = \sigma(M_1, \ldots, M_k)\Downarrow_{m+1}^T \sigma^T(X_1, \ldots, X_k)$ con $M_j \in \Lambda_0^\Sigma$ con $j \in [1, k]$ donde además $M_j\Downarrow_m^T X_j$ con $j \in [1, k]$. Por lo que también tenemos que $M = \sigma(M_1, \ldots, M_k)\Downarrow_{m+1}^T w$, con $w = \sigma^T(Y_1, \ldots, Y_k)$ y $M_j\Downarrow_m^T Y_j$ con $j \in [1, k]$. Pero finalmente, por hipótesis inductiva tenemos que $X_j = Y_j$ con $j \in [1, k]$, por lo que $\sigma^T(X_1, \ldots, X_k) = \sigma^T(Y_1, \ldots, Y_k)$, y entonces, $v = w$.

Caso (`seq`) Sea $M = \mathbf{let}\ x = P\ \mathbf{in}\ Q\Downarrow_{m+1}^T v$ con $P \in \Lambda_0^\Sigma$ y $Q \in \Lambda_{\{x\}}^\Sigma$, de forma tal que $P\Downarrow_m^T X$, y existe $V \in \mathcal{V}_0^\Sigma$ tal que $Q[x := V] \Downarrow_m^T Y_V$, tales que $v = X >\!\!>\!\!= (V \rightsquigarrow Y_V)$. Sabemos que $M = \mathbf{let}\ x = P\ \mathbf{in}\ Q\Downarrow_{m+1}^T w$ por lo que $P\Downarrow_m^T R$, y existe $W \in \mathcal{V}_0^\Sigma$ tal que $Q[x := W]\Downarrow_m^T S_W$, tales que $w = R >\!\!>\!\!= (W \rightsquigarrow S_W)$. Por hipótesis inductiva en P, tenemos que $X = R$, y más aún, los valores dentro de los efectos generados son los mismos, $V = W$. Por hipótesis inductiva en Q, y que $V = W$, tenemos que $Y_V = S_W$. Concluimos que $X >\!\!>\!\!= (V \rightsquigarrow Y_V) = R >\!\!>\!\!= (W \rightsquigarrow S_W)$, es decir, $v = w$.

□

Veamos ahora que efectivamente la familia de aproximaciones define una ω-cadena desde el punto de vista de un término, donde a medida que se incrementa la profundidad del árbol de evaluaciones, se obtiene un valor más *definido*.

Lema 6.3.2 (Cadena de aproximaciones)**.** *Sea $M \in \Lambda_0^\Sigma$, para todo $n < \omega$, si $M\Downarrow_n^T X$, para todo $N < \omega$, $M\Downarrow_{n+N}^T Y$ entonces $X \sqsubseteq Y$.*

De esta manera obtenemos una familia de relaciones indexadas por un número natural $n \in \mathbb{N}$. Utilizando está familia de relaciones, podemos definir la evaluación de términos con efectos algebraicos como el supremo de la cadena de sus aproximaciones, al igual que hicimos en la Sección 3.3.

La regla `seq` no expresa la semántica operacional de forma clara. Por el nivel de generalidad que queremos tener en el lenguaje, donde caracterizamos una familia amplia de lambda cálculos con efectos, utilizamos una regla que, si bien expresa se forma clara la semántica del constructor **let** no queda en claro que esa operacional. Para subsanar este problema definimos la aproximación de términos con efectos algebraicos de la misma manera que hicimos en el Capítulo 3. Utilizamos una familia inductiva de relaciones que aproximan a un término de manera operacional y mostramos que la evaluación aproximada respeta dicha familia.

Definición 6.3.7. *Sea Σ una signatura, T una mónada Σ-continua, y $M \in \Lambda_0$ un término. Definimos la n-ésima aproximación $M^{(n)} \in T(\mathcal{V}_0)$ de M de la siguiente forma:*

$$\begin{array}{rcl} M^{(0)} & \doteq & \bot_T \\ (\mathbf{ret}(V))^{(n+1)} & \doteq & \eta(V) \\ ((\boldsymbol{\lambda}\ x\ .\ M)\, V)^{(n+1)} & \doteq & (M[x := V])^{(n)} \\ (\mathbf{let}\ x = M\ \mathbf{in}\ N)^{(n+1)} & \doteq & M >\!\!>\!\!= \lambda V.(N[x := V])^{(n)} \\ (\sigma(M_1, \ldots, M_m))^{(n+1)} & \doteq & \sigma^T(M_1^{(n)}, \ldots, M_m^{(n)}) \end{array}$$

Mostrar inductivamente que la relación de evaluación aproximada respeta la n-ésima aproximación de términos.

Lema 6.3.3. *Dado una signatura Σ, T una mónada Σ-continua, todo término cerrado $M \in \Lambda_0^\Sigma, M \Downarrow_n^T M^{(n)}$.*

Demostración. La prueba sigue la prueba del Lema 3.3.1, salvo para el caso particular de los operadores algebraicos agregados por la signatura Σ.

Procedemos por inducción sobre n. El caso base es con $n = 0$, que por definición de la evaluación aproximada y por la n-ésima aproximación se cumple que $M \Downarrow_0^T M^{(0)}$.

Para el paso inductivo asumimos que para todo $M \in \Lambda_0, M \Downarrow_m^T M^{(m)}$, y mostramos que $M \Downarrow_{m+1}^T M^{(m+1)}$. Mostramos a continuación el caso puntual de los operadores agregados por la signatura Σ, ya que los otros casos son similares a los probados en la prueba del Lema 3.3.1.

Sea $\sigma \in \Sigma$ un operador con aridad $\alpha(\sigma) = n$. Por hipótesis inductiva tenemos que dados $M_1, \ldots, M_n \in \Lambda_0$, $M_i \Downarrow_m^T M_i^{(m)}$. Nuestro objetivo es mostrar que $\sigma(M_1, \ldots, M_n) \Downarrow_{m+1}^T (\sigma(M_1, \ldots, M_n))^{(m+1)}$. Por hipótesis inductiva tenemos que $M_i \Downarrow_m^T M_i^{(m)}$, y por la regla ($\sigma$-op), $\sigma(M_1, \ldots, M_n) \Downarrow_{m+1}^T \sigma^T(M_1^{(m)}, \ldots, M_n^{(m)})$. Finalmente, por definición de n-ésima aproximación, $\sigma(M_1, \ldots, M_n) \Downarrow_{m+1}^T \sigma(M_1, \ldots, M_n)^{(m+1)}$. □

Sea Σ una signatura y T una mónada Σ-continua, definimos la evaluación de un término cerrado $M \in \Lambda_0^\Sigma$ de la siguiente forma:

$$[\![M]\!]^T \doteq \sup_{n<\omega}\{M^{(n)}\}$$

De forma similar al capítulo 3, esta definición nos permite caracterizar explícitamente la divergencia en la evaluación. Por ejemplo, podemos ver que para el término divergente $\Omega \in \Lambda_0$ definido en el Capítulo 3, tenemos que para cualquier mónada ordenada T, $[\![\Omega]\!]^T \equiv \bot_T$. Esto es una consecuencia directa de utilizar mónadas ordenadas para definir la evaluación de términos, introduciendo un término $\bot$ para cada conjunto posible, y de esta manera, caracterizar términos que, sin importar la profundidad del árbol de derivación, nunca se les podrá asignar un valor.

6.4. Evaluación de Lenguajes con Efectos

En esta sección finalmente mostramos como combinar todas las piezas necesarias para evaluar términos de los lenguajes introducidos al principio del

capítulo. En otras palabras, definiremos la estructura necesaria para evaluar términos con: fallo, excepciones, no-determinismo, y probabilidades. De forma general lo que haremos es definir una mónada ordenada capaz de interpretar cada uno de los efectos.

Fallo/Divergencia Recapitulando, debemos definir una mónada para interpretar efectos del álgebra $\langle\{\mathsf{fail}\}, \mathsf{fail} \mapsto 0\rangle$. Definimos $\mathbb{F}(X) \doteq (\mathcal{V}_0 + \bot) + \bot$ y operaciones:

- $\eta(x) \doteq \iota_l(\iota_l(x))$
- $m >\!\!>\!\!= f \doteq \begin{cases} f(x) & \text{si } m = \iota_l(\iota_l(x)) \\ \iota_l(\iota_r(\bot)) & \text{si } m = \iota_l(\iota_r(\bot)) \\ \iota_r(\bot) & \text{si } m = \iota_r(\bot) \end{cases}$

Podemos ver que $(\mathbb{F}, \eta, >\!\!>\!\!=)$ definen una mónada.

Veamos también que es una mónada ordenada, y para esto construimos de forma universal un orden.

Sea X un conjunto, podemos definir:

- el menor elemento como $\bot_X \doteq \iota_r(\bot)$,
- mientras que dados dos elementos $x, y \in \mathbb{F}(X)$ para algún conjunto X, $x \sqsubseteq^{\bot}_X y$ si y solo si:
  - o bien, $x = \iota_r(\bot)$ es el menor elemento,
  - o si x es un fallo, también lo es y, $x = \iota_l(\iota_r(\bot)) \wedge y = \iota_l(\iota_r(\bot))$
  - o si x contiene un valor, y contiene el mismo, $\exists x', y', x = \iota_l(\iota_l(x')) \wedge y = \iota_l(\iota_l(y')) \wedge x' = y'$

Podemos ver que $\bot_X \sqsubseteq^{\bot}_X x$ para todo $x \in \mathbb{F}(X)$.

El operador de **bind** es continuo en ambos argumentos. Mostramos primero que para $x \in \mathbb{F}(X)$, y una ω-cadena de funciones $\{f_n\}_{n<\omega}$ donde $f_n \in X \to \mathbb{F}(Y)$, el supremo de aplicar el operador **bind** a la función f_n con x, es igual a aplicar el operador **bind** al supremo de las funciones con x. Para esto hacemos análisis por caso en x:

- Caso $x = \iota_r(\bot)$. Por definición de **bind** tenemos que $\iota_r(\bot) >\!\!>\!\!=^{\bot} f_n = \bot$, por lo que $\bigsqcup_{n<\omega}(\iota_r(\bot) >\!\!>\!\!= f_n) = \bigsqcup_{n<\omega} \iota_r(\bot) = \iota_r(\bot)$. Pero además, tenemos que $\iota_r(\bot) >\!\!>\!\!= (\bigsqcup_{n<\omega} f_n) = \iota_r(\bot)$. Por lo que podemos concluir que: $\bigsqcup_{n<\omega}(\iota_r(\bot) >\!\!>\!\!= f_n) = \iota_r(\bot) >\!\!>\!\!= (\bigsqcup_{n<\omega} f_n)$
- Caso $x = \iota_l(\iota_r(\bot))$. Similar al caso anterior aplicamos directamente la definición del operador **bind**. Por un lado tenemos que, para todo $n < \omega$, $\iota_l(\iota_r(\bot)) >\!\!>\!\!= f_n = \iota_l(\iota_r(\bot))$, y por ende, $\bigsqcup_{n<\omega}(\iota_l(\iota_r(\bot)) >\!\!>\!\!= f_n) = \iota_l(\iota_r(\bot))$. Mientras que por el otro lado tenemos: $\iota_l(\iota_r(\bot)) >\!\!>\!\!= (\bigsqcup_{n<\omega} f_n) = \iota_l(\iota_r(\bot))$. Por lo que concluimos que ambos términos son iguales.
- Caso $x = \iota_l(\iota_l(x')$ para algún $x' \in X$. De nuevo aplicamos directamente la definición del operador **bind**. Por un lado tenemos que, para todo $n < \omega$: $\iota_l(\iota_l(x') >\!\!>\!\!= f_n = f_n(x')$, y por ende, $\bigsqcup_{n<\omega}(\iota_l(\iota_l(x') >\!\!>\!\!= f_n) = \bigsqcup_{n<\omega}(f_n(x'))$. Por el otro lado tenemos que: $\iota_l(\iota_l(x') >\!\!>\!\!= \bigsqcup_{n<\omega} f_n =$

$(\bigsqcup_{n<\omega} f_n)(x')$. Finalmente, por continuidad de la aplicación de funciones, ambos términos son iguales.

Concluimos que la mónada está ordenada.

Lo que nos queda es definir cómo son interpretados los operadores que agrega el álgebra al lenguaje. En la Sección 6.1, introducimos el efecto de fallo, y explicamos en palabras como introducir intuitivamente un elemento del conjunto de posibles resultados de la evaluación. Este elemento, que notamos $\iota_l(\iota_r(\bot))$, representa la interpretación del operador fail, por lo que ahora podemos describirlo formalmente. Para todo conjunto X, podemos interpretar el símbolo fail dentro de $\mathbb{F}(X)$ como:

$$\mathsf{fail}^{\mathbb{F}(X)} \doteq \iota_l(\iota_r(\bot))$$

Finalmente, nos queda mostrar que la mónada $\mathbb{F}$ es continua en el álgebra de fallo. Pero esto lo obtenemos juntando la interpretación del operador y el orden de la mónada. La interpretación del operador fail define cadenas estacionarias, es decir, cadenas que contienen una secuencia de elementos $\iota_r(\bot)$, seguida de una cadena constante $\{\mathsf{fail}^{\mathbb{F}}\}_{n<\omega}$.

Por lo que podemos concluir que la función de evaluación $[\![-]\!]^{\bot}$ está bien definida.

Excepciones Dado un conjunto de símbolos E que representan diferentes excepciones, podemos agregar al lenguaje operadores de excepciones mediante el álgebra $\langle\{\mathsf{raise}_e \mid e \in E\}, \{\mathsf{raise}_e \mapsto 0 \mid e \in E\}\rangle$. En esta sección, al igual que con el operador de fallo, mostramos que tenemos una mónada capaz de interpretar cada uno de los operadores, y definimos formalmente la evaluación de término.

Definimos $\mathbb{E}(X) \doteq (X + E) + \bot$, lo que nos permite indicar cuando hemos alcanzado una excepción durante la evaluación de términos. Siguiendo los mismos pasos que la mónada $\mathbb{F}$ podemos ver que efectivamente es una mónada respecto al álgebra de excepciones, donde definimos la interpretación de los operadores de la siguiente forma. Para todo conjunto X, y $e \in E$, tenemos que:

$$\mathsf{raise}_e^{\mathbb{E}(X)} \doteq \iota_l(\iota_r(e))$$

Además, cada operador es interpretado de forma tal que solo permite obtener cadenas estacionarias, y por ende, continuas.

Finalmente, la función de evaluación $[\![-]\!]^E$, definida en la Sección 6.1, está bien definida.

No-determinismo En la Sección 6.1 introducimos no-determinismo mediante el álgebra $\langle\{\oplus\}, \oplus \mapsto 2\rangle$. Continuando con el espíritu de esta sección, a continuación vamos a ver que $\mathbb{ND}(X) \doteq \mathcal{P}(X)$ define una mónada continua.

Definición 6.4.1 (Mónada de No-determinismo)**.** *Para cada conjunto X, definimos $\mathbb{ND}(X)$ como todos los subconjunto de X. En símbolos $\mathbb{ND}(X) \doteq \{Y \mid$*

$Y \subseteq X\}$. *Definimos además las siguientes operaciones:*

$$\eta_X(x) \doteq \{x\}$$
$$A >\!\!>\!= f \doteq \bigcup_{a \in A} f(a)$$

Para cualquier conjunto Y, $x \in X$, $A \subseteq X$ *y* $f : X \to \mathbb{ND}(Y)$.

Nuevamente, mostrar que $(\mathbb{ND}, \eta, >\!\!>\!=)$ es efectivamente una mónada se deja como ejercicio para el lector.

La mónada $\mathbb{ND}$ tiene un orden. Para cada conjunto X definimos:

$$\bot_X^{\mathbb{ND}} \doteq \emptyset$$
$$X \sqsubseteq^{\mathbb{ND}} Y \iff X \subseteq Y$$

Por propiedades de conjuntos, tenemos que para todo conjunto X, y $Y \subseteq X$, $\bot_X^{\mathbb{ND}} \sqsubseteq^{\mathbb{ND}} Y$.

El operador **bind** es continuo en ambos argumentos. Esto resulta sencillo ya que el operador **bind** se define en base a la unión de conjuntos y la aplicación de funciones, tanto la unión como la aplicación, son operaciones continuas en el orden definido por la inclusión de conjuntos.

Veamos que es continuo en el primer argumento. Dado dos conjuntos P, Q, una ω-cadena, $\{X_n\}_{n<\omega}$, de subconjuntos de P, $X_n \subseteq P$ para cada $n < \omega$, y una función $f : P \to \mathbb{ND}(Q)$, veamos que $\bigsqcup_{n<\omega}(X_n >\!\!>\!= f) = (\bigsqcup_{n<\omega} X_n) >\!\!>\!= f$. Para esto nos basamos que para cualquier par de elementos X_i, X_j con $i < j$, tenemos que $X_i \subseteq X_j$, y por lo tanto, $\bigcup_{x \in X_i} f(x) \subseteq \bigcup_{x \in X_j} f(x)$. En otras palabras, estamos viendo que aplicar la función parcial $(>\!\!>\!= f)$ respeta el orden de la ω-cadena, y por ende, es continuo.

Dado un conjunto P, Q, $X \subseteq P$, y una ω-cadena de funciones $\{f_n\}_{n<\omega}$ tales que $f_n : P \to \mathbb{ND}(Q)$ para todo $n < \omega$. Por definición del operador **bind**, tenemos que:

$$X >\!\!>\!= f_n = \bigcup_{x \in X} f_n(x)$$

Por el orden de funciones, tenemos que $f_i \sqsubseteq f_j$ con $i < j$ si y solo si para todo $x \in P, f_n(x) \sqsubseteq^{\mathbb{ND}} f_j(x)$. Y aún más, tenemos que por definición de unión de conjuntos, $\bigcup_{x \in X}(f_i) \subseteq \bigcup_{x \in X}(f_j)$, para todo $i < j$. Por lo que tenemos que $\bigcup_{n<\omega}(X >\!\!>\!= f_n) = X >\!\!>\!= \bigcup_{n<\omega} f_n$. Concluyendo la prueba de que **bind** es continuo en ambos argumentos.

Definimos ahora como interpretar el operador binario de no-determinismo $\oplus$. Lo hacemos directamente mediante la unión de conjuntos, en símbolos, para cualquier conjunto P:

$$\oplus^{\mathbb{ND}(P)}(X, Y) \doteq X \cup Y$$

Y más aún, el operador de unión es continuo en el orden definido por la inclusión de conjunto. Por lo que podemos concluir entonces que la evaluación $[\![-]\!]^{ND}$ está bien definida por la mónada $\mathbb{ND}$.

Probabilístico Finalmente veamos como definir una mónada continua capaz de interpretar los efectos generados por el operador binario probabilístico $\oplus_p$. Para esto utilizamos la familia de sub-distribuciones, y vemos que forman efectivamente una mónada. En la Sección 6.1 definimos $\mathbb{D}(X)$ como el conjunto de todas las sub-distribuciones con soporte numerable en X. Sea X un conjunto, definimos las siguientes operaciones:

$$\eta(v)(x) \doteq \begin{cases} 1 & \text{si } x = v \\ 0 & \text{si } x \neq v \end{cases}$$

$$(d >>= f)(y) \doteq \sum_{v \in sup(d)} d(v) * f(v)(y)$$

Con $d \in \mathbb{D}(X)$, $f : X \to \mathbb{D}(Y)$.

El operador **bind** se define directamente a partir de la probabilidad de obtener un elemento de la sub-distribución d y la probabilidad de obtener un un nuevo elemento a través de f teniendo el cuenta el valor obtenido en d. De esta manera la probabilidad de obtener un elemento v en la sub-distribución $d >>= f$ queda definida por la probabilidad de obtener un elemento x en d, y con dicho elemento, computar $f(x)(v)$.

Dado que las sub-distribución son un caso particular de funciones, podemos utilizar el orden por funciones en codominios ordenados. Para cada conjunto X podemos definir el siguiente orden:

$$\bot^{\mathbb{D}(X)}(v) \doteq 0$$

$$d \sqsubseteq^{\mathbb{D}(X)} d' \iff \forall x \in sup(d), d(x) \leq d'(x)$$

De esta manera es fácil comprobar que $\bot^{\mathbb{D}(X)} \sqsubseteq d$ para todo conjunto X y $d \in \mathbb{D}(X)$. Más aún, como definimos el orden $(\sqsubseteq^{\mathbb{D}(X)})$ como la extensión a funciones basándonos en el orden de los reales, las operaciones aritméticas son continuas, y por lo tanto, el operador **bind** es continuo en ambos argumentos. Notar que para que $d \sqsubseteq^{\mathbb{D}(X)} d'$, necesitamos que $sup(d) \subseteq sup(d')$.

Nos queda entonces por definir la interpretación de la familia de operadores binarios $\oplus_p$. Para cualquier conjunto X y $p \in [0, 1]$, definimos la interpretación del operador probabilístico como:

$$\oplus_p^{\mathbb{D}(X)}(d, d')(v) \doteq p * d(v) + (1 - p) * d'(v)$$

para cualquiera dos sub-distribuciones $d, d' \in \mathbb{D}(X)$. Para mostrar que el operador $\oplus_p^{\mathbb{D}(X)}$ para todo $p \in [0, 1]$ es continuo basta con recurrir a la continuidad de operaciones aritméticas y de la aplicación de funciones.

Operaciones Genéricas

Finalmente, es posible extender el trabajo para obtener operaciones generalizadas. Introducir operadores mediante signaturas con un único sort no alcanza para especificar completamente algunos efectos algebraicos, como ser, el manejo de estados mutables o la operación de imprimir un mensaje en pantalla. En particular, la manera antes descripta nos permite introducir operadores con una aridad finita, pero no con aridad dependiente de un conjunto infinito. Como ejemplo podemos ver que introducimos una familia de operadores probabilísticos

$\oplus_p$ con $p \in [0,1]$. Que no es lo mismo que introducir *un* operador probabilístico $\oplus$ que pueda tomar como argumento un real indicando la probabilidad de seleccionar cada término. Un ejemplo muy utilizado en la programación funcional, es el uso de un estado global, donde se utilizan dos operadores: $\mathsf{get}, \mathsf{set}$. Donde el operador $\mathsf{get}(l, x\,.\,k)$ nos permite buscar el valor guardado en l, digamos que es v, y continuar la evaluación con el término resultante reemplazando las apariciones libres de x por v en k. Mientras que el operador $\mathsf{set}((l,v),k)$ nos permite guardar el valor v en la dirección l y continuar con la evaluación de k. Notar que tanto las direcciones como el espacio de valores pueden ser infinitos. Para aceptar dichos operadores se introducen las denominadas *operaciones genéricas* (Plotkin y Power 2003).

Definición 6.4.2 (Operación Genérica (Dal Lago y Gavazzo 2019))**.** *Sean I, X dos conjuntos. Una operación genérica en X es una función $\omega : P \times X^I \to X$. El conjunto P se denomina* parámetro del operador*, y el conjunto índice I es la aridad de la operación.*

Podemos generalizar entonces la noción de signatura Σ como un conjunto de símbolos no interpretados O con un mapa de aridades de manera tal que para todo $o \in O, o : P \rightsquigarrow I$ indica que el símbolo o representa una operación genérica con parámetro P y conjunto índice I. Semánticamente, decimos una mónada T interpreta la operación o si para todo conjunto X hay un mapa $[\![o]\!]_X : P \times (T(X))^I \to T(X)$, tal que para toda función $f : X \to T(Y)$, $p \in P$, y $\kappa : I \to T(X)$ tenemos que:

$$[\![o]\!]_X(p, \kappa) >\!\!>\!\!= f = [\![o]\!]_Y(p, (>\!\!>\!\!= f) \circ \kappa)$$

Finalmente, al igual que con operaciones algebraicas, decimos que la mónada T es Σ-algebraica si ya viene equipada con una interpretación para los símbolos definimos en Σ.

Ejemplo 6.4.1 (Estado Global)**.** *Dado un conjunto de identificadores de memoria L. Definimos la signatura $\mathbb{G} \doteq \{\mathsf{get} : L \rightsquigarrow \mathcal{V}, \mathsf{set} : L \times \mathcal{V} \rightsquigarrow 1\}$.*

Donde la operación get *se utiliza para obtener el valor de posición $l \in L$, mientras que la operación* set *se utiliza para guardar un valor en una posición dada. En la signatura dada se utilizan tres conjuntos: $\mathcal{V}$ de valores, L representando el nombre de espacios donde guardar valores, y el conjunto unitario* 1.

Para interpretar las operaciones recién definidas utilizamos la mónada de estados:

$$\begin{aligned} G(X) &\doteq (X \times S)^S \\ \eta_X(x)(s) &\doteq (x, s) \\ ma >\!\!>\!\!= f &\doteq s \mapsto f(\mathsf{fst}(ma(s)))(\mathsf{snd}(ma(s))) \end{aligned}$$

Utilizando como estado global un mapa de nombres a valores: $S = L \to \mathcal{V}$.

Interpretamos las operaciones genéricas $\mathsf{get}, \mathsf{set}$ *de la siguiente forma:*

$$\begin{aligned} [\![\mathsf{set}]\!]((l,v),k)(\sigma) &\doteq k(1)(\sigma[l \vdash v]) \\ [\![\mathsf{get}]\!](l,k)(\sigma) &\doteq k(\sigma(l))(\sigma) \end{aligned}$$

Donde $\sigma[l \vdash v]$ lo que hace es actualizar la posición l con el valor v.

De esta manera obtenemos dos operaciones dentro del lenguaje $\mathsf{set}((l,v),e)$ *y* $\mathsf{get}(l, x.e)$*. La operación* $\mathsf{set}((l,v),e)$*, al ser interpretada, guarda en la dirección l*

el valor v y continúa con la evaluación de e. Mientras que la operación $\mathsf{get}(l, x.e)$*, busca el valor v almacenado en la dirección l, y continúa con la evaluación del término $e[x := v]$.*

Capítulo 7

Aproximación de Programas con Efectos

En este capítulo adaptamos la aproximación de programas para que sea capaz de aceptar la presencia de efectos en el lenguaje y así obtener una definición de equivalencia con efectos. En el capítulo anterior vimos cómo podíamos introducir de una manera genérica efectos algebraicos al cálculo lambda, pero que luego, para evaluar los términos de dicho lenguaje, utilizábamos una mónada capaz de interpretar dichos efectos. Lo que nos falta entonces es definir una noción de aproximación y equivalencia de programas, similar a las relaciones del Capítulo 4, de manera que podamos luego incorporar finalmente una teoría de mejoras teniendo en cuenta los efectos introducidos por los operadores e interpretados por la mónada, con el objetivo de obtener una teoría de mejoras similar a la del Capítulo 5.

Las ideas siguen el camino inverso al Capítulo 4, primero definimos la relación de aproximación aplicativa y luego de aproximación observacional. Seguimos el orden inverso ya que el autor cree que es más directo ver la aplicación de un elemento que mezcle la relación de aproximación aplicativa con la evaluación de términos con efectos algebraicos. En particular, la relación de aproximación aplicativa define unas sucesivas evaluaciones de términos, y por lo tanto, necesitamos poder mezclar la relación de aproximación aplicativa de términos con valores con efectos. Recordemos que la evaluación de términos es mónadica, ya que debe ser capaz de interpretar los efectos generados por los operadores algebraicos. La idea original de cómo definir las aproximaciones en presencia de efectos fue desarrollada por Dal Lago, Gavazzo y Paul Levy (2017), donde los autores introducen cómo obtener una noción de equivalencia contextual y relaciones de bisimulación. Estos conceptos son los pilares en los cuales se basa uno de los principales aportes de este trabajo, que consiste en introducir la teoría de mejoras basada en un nuevo efecto que desarrollamos en los capítulos posteriores (Capítulo 8).

El objetivo de este capítulo es definir una relación entre programas y otra entre términos: *aproximación aplicativa* y *aproximación observacional*. La aproximación observacional se relaciona directamente con la equivalencia observacional de términos con efectos. Éste enfoque, hereda el mismo problema que la equivalencia observacional: probar que dos términos son observacionalmente equivalentes involucra probar que *universalmente* ambos presentan el mismo

comportamiento observable. Adicionalmente, la relación de aproximación aplicativa, de la misma manera que la simulación aplicativa de Abramsky, nos permite inferir la equivalencia definiendo diferentes simulaciones, y observando que se alcanzan estados equivalentes. Definir la relación de aproximación aplicativa requiere dar una semántica operacional, que es lo que nos permite a nosotros introducir nociones de costos dentro de la evaluación del lenguaje.

El contenido de este capítulo se basa en el trabajo de Dal Lago, Gavazzo y Paul Levy (2017), donde el objetivo es encontrar una relación de congruencia, y por ende, una relación de equivalencia entre programas. En nuestro caso, el objetivo es desarrollar una relación no necesariamente simétrica, sino una relación que expresa que la evaluación de un término es a lo sumo tan costosa como otra. Utilizando la nomenclatura del artículo, no buscamos una bisimulación, sino más bien una simulación entre programas. En otras palabras, no buscamos una relación congruente sino una *precongruencia*.

7.1. Relacionadores

Para obtener una precongruencia entre programas con efectos, el primer obstáculo que debemos resolver es definir cómo comparar valores con efectos. Repasando la noción de simulación de Abramsky, los valores dentro del lambda cálculo son representados como valores cerrados, es decir, que son identificados con abstracciones, i.e. funciones, que no se pueden reducir. Por lo que, dados dos valores, los podemos *comparar* de forma extensional, es decir, aplicando los valores sobre un mismo valor obteniendo dos términos, y finalmente comparando el resultado de evaluarlos, de la misma manera que haríamos con funciones. Y luego, volver a repetir el proceso las veces que sea necesario, hasta encontrar valores que, en el mejor caso, sean equivalentes. Como vimos en el Capítulo 4, el proceso antes descripto en palabras tiene una dependencia recursiva que se puede salvar definiendo cuidadosamente la relación de forma coinductiva mediante un operador monótono, utilizando el teorema de Knaster-Tarski (Teorema 2.1.6).

En esta sección, lo que veremos es como podemos incorporar efectos al proceso antes descripto, y esto lo logramos viendo cómo relacionar valores que además tienen efectos computacionales. En cada paso, dados dos términos, los evaluamos y comparamos los resultados, pero ahora la evaluación ya no nos retorna un valor sino *un valor con efectos*. Por lo que definimos una forma de mapear relaciones entre valores a relaciones entre valores con efectos, de manera tal que se extrapole correctamente la relación semántica entre los valores. Para esto introducimos la noción de *relacionador* y T-relacionador, y finalmente mostramos que podemos obtener los mismos resultados que el Capítulo 4 pero teniendo en cuenta los efectos algebraicos presentes en nuestro lenguaje.

Definición 7.1.1 (Relacionador)**.** *Sea F un endofuntor sobre $\mathcal{S}et$ y X, Y dos conjuntos. Un* relacionador Γ *para F es un mapa que asocia cualquier relación $R \subseteq X \times Y$ con una relación $\Gamma R \subseteq F(X) \times F(Y)$ tal que:*

$$\mathbf{1}_{FX} \subseteq \Gamma(\mathbf{1}_X) \tag{Rel-1}$$

$$\Gamma R \circ \Gamma S \subseteq \Gamma(R \circ S) \tag{Rel-2}$$

$$\Gamma((f \times g)^{-1} R) \subseteq (Ff \times Fg)^{-1} \Gamma R \tag{Rel-3}$$

$$R \subseteq T \implies \Gamma R \subseteq \Gamma T \tag{Rel-4}$$

Donde $R, T \subseteq X \times Y, S \subseteq Y \times Z$, $f : W \to X, g : Z \to Y$ *y el operador de inversión* $(f \times g)^{-1} R = \{(w, z) | f(w) R g(z)\}$.

La noción de relacionador fue introducida por primera vez por Kawahara (1973), aunque el concepto no fue utilizado hasta que se vuelve a presentar por Carboni, Kelly y Wood(1991). A su vez, Backhouse y sus colegas, comenzaron una serie de artículos que muestran la relevancia de los relacionadores en la computación (Backhouse, de Bruin, P.F. Hoogendijk et al. 1991; Backhouse, de Bruin, Malcolm et al. 1991; Backhouse y P. Hoogendijk 1993). En el libro *The Algebra of Programming* los autores dedican una gran parte del libro al uso de *alegorías*, que son una construcción similar a la categorías pero partiendo de generalizar las propiedades de relaciones en vez de conjuntos y funciones, siendo los relacionadores funtores dentro de las alegorías.

Los relacionadores poseen varias propiedades.

Lema 7.1.1. *Relacionadores respetan la composición de funtores.*

Demostración. Dados dos funtores F, G, y relacionadores Γ, Δ, de F y G, respectivamente. Podemos ver que la composición de Γ, Δ, $\Gamma \circ \Delta$, es un relacionador de $F \circ G$ probando cada una de las 4 propiedades enunciadas en su definición.

- Para cualquier conjunto X tenemos que la identidad de elementos mapeamos por el funtor $F \circ G$ está contenida en el resultado de mapear la identidad en X por el relacionador $\Gamma \circ \Delta$. Por ser Γ, Δ relacionadores de sus respectivos funtores, tenemos que: $\mathbf{1}_{(F(G(X)))} \subseteq \Gamma(\mathbf{1}_{G(X)})$, y además, $\mathbf{1}_{G(X)} \subseteq \Delta(\mathbf{1}_X)$, ambos por Regla Rel-1 de relacionadores. Pero además, por monotonía de relacionadores sobre la inclusión, tenemos que: $\Gamma(\mathbf{1}_{G(X)}) \subseteq \Gamma(\Delta(\mathbf{1}_X))$. Por lo que podemos concluir que $\mathbf{1}_{(F \circ G)(X)} \subseteq (\Gamma \circ \Delta)(\mathbf{1}_X)$.

- Para cualquier conjunto X, Y, Z y relaciones R, S tales que $R \subseteq X \times Y$ y $S : Y \times Z$, tenemos que la composición de mapear la relación R y S mediante el relacionador $\Gamma \circ \Delta$ está contenida en la relación resultante de mapear directamente la composición de las relaciones R y S.

$$
\begin{array}{l}
(\Gamma \circ \Delta)(R) \circ (\Gamma \circ \Delta)(S) \\
\subseteq \langle \text{Definición composición y Regla Rel-2} \rangle \\
\Gamma(\Delta(R) \circ \Delta(S)) \\
\subseteq \langle \text{Regla Rel-2 y Regla Rel-4} \rangle \\
\Gamma(\Delta(R \circ S)) \\
= \langle \text{Definición de composición} \rangle \\
(\Gamma \circ \Delta)(R \circ S)
\end{array}
$$

- Para cualquier conjunto X, Y, W, V, funciones $f : W \to X$ y $g : V \to Y$, y una relación $R \subseteq X \times Y$, tenemos que:

$$(\Gamma \circ \Delta)((f \times g)^{-1} R) \subseteq ((F \circ G)(f) \times (F \circ G)(g))^{-1} (\Gamma \circ \Delta) R$$

Por ser Δ un relacionador, tenemos que mostrar que:

$$\Delta((f \times g)^{-1} R) \subseteq (G(f) \times G(g))^{-1} (\Delta R) \qquad (L)$$

Ahora, por ser Γ un relacionador, tenemos que:

$$
\begin{array}{l}
(\Gamma \circ \Delta)((f \times g)^{-1} R) \\
= \langle \text{Definición de composición} \rangle \\
\Gamma(\Delta((f \times g)^{-1} R)) \\
\subseteq \langle \text{Regla Rel-4 y lema auxiliar } L \rangle \\
\Gamma((G(f) \times G(g))^{-1}(\Delta R)) \\
\subseteq \langle \text{Regla Rel-3} \rangle \\
(F(G(f)) \times F(G(g)))^{-1}(\Gamma(\Delta R)) \\
= \langle \text{Definición de composición} \rangle \\
((F \circ G)(f) \times (F \circ G)(g))^{-1}((\Gamma \circ \Delta) R)
\end{array}
$$

- Finalmente, aplicando sucesivamente la Regla Rel-4 de ambos relacionadores tenemos que, dados dos conjuntos X, Y y dos relaciones $R, T \subseteq X \times Y$ tales que $R \subseteq T$, $(\Gamma \circ \Delta)(R) \subseteq (\Gamma \circ \Delta)(T)$.

□

Lema 7.1.2 (Intersección de Relacionadores)**.** *Dado un funtor F y dos relacionadores Γ, Δ de F. La conjunción (intersección) de relacionadores $(\Gamma \wedge \Delta)$ es un relacionador de F. Donde, dada una relación R, la conjunción de relacionadores queda definida por:*

$$x\ (\Gamma \wedge \Delta)(R)\ y \iff x\ \Gamma R\ y \wedge x\ \Delta R\ y$$

Demostración. La intersección de los relacionadores Γ y Δ, $\Gamma \wedge \Delta$, es un relacionador para el funtor F.

- Podemos ver que $\mathbf{1}_{FX} \subseteq (\Gamma \wedge \Delta)(\mathbf{1}_X)$, ya que por definición del relacionador de conjunción tenemos que: $\mathbf{1}_{FX} \subseteq \Delta(\mathbf{1}_X) \wedge \mathbf{1}_{FX} \subseteq \Gamma(\mathbf{1}_X)$. Finalmente, sabemos que ambos predicados son correctos por ser Γ y Δ relacionadores.

- Veamos que $\Gamma \wedge \Delta$ cumple con la segunda regla de los relacionadores. Sean x, z tales que $x(\Gamma \wedge \Delta)(R) \circ (\Gamma \wedge \Delta)(S)z$, veamos que también $x(\Gamma \wedge \Delta)(R \circ S)z$.

$$
\begin{array}{l}
x\ ((\Gamma \wedge \Delta)(R) \circ (\Gamma \wedge \Delta)(S))\ z \\
\iff \langle \text{Definición composición de relaciones} \rangle \\
\exists y, x(\Gamma \wedge \Delta)(R) y \wedge y(\Gamma \wedge \Delta)(S) z \\
\iff \langle \text{Definición conjunción de relacionadores} \rangle \\
\exists y, x\Gamma(R) y \wedge x\Delta(R) y \wedge y\Gamma(S) z \wedge y\Delta(S) z \\
\implies \langle \text{Definición composición de relaciones} \rangle \\
x\Gamma(R \circ S) z \wedge x\Delta(R \circ S) z \\
\iff \langle \text{Definición conjunción de relacionadores} \rangle \\
x(\Gamma \wedge \Delta)(R \circ S) z
\end{array}
$$

- Veamos que $(\Gamma \wedge \Delta)((f \times g)^{-1} R) \subseteq (Ff \times Fg)^{-1}(\Gamma \wedge \Delta)(R)$. Para mostrar este caso, partimos de elementos relacionados por $(\Gamma \wedge \Delta)((f \times g)^{-1} R)$ y finalmente mostramos que también están relacionados por $(F\,f \times F\,g)^{-1}(\Gamma \wedge$

$\Delta)R$. Sean x, y tales que $x(\Gamma \wedge \Delta)((f \times g)^{-1}R)y$.

$$\begin{array}{rl} & x(\Gamma \wedge \Delta)((f \times g)^{-1}R)y \\ \iff & x\Gamma((f \times g)^{-1}R)y \wedge x\Delta((f \times g)^{-1}R)y \\ \implies & x(Ff \times Fg)^{-1}(\Gamma R)y \wedge x(Ff \times Fg)^{-1}(\Delta R)y \\ \iff & F\,f\,x(\Gamma R)F\,g\,y \wedge F\,f\,x(\Delta R)F\,g\,y \\ \iff & F\,f\,x((\Gamma \wedge \Delta)R)F\,g\,y \\ \iff & x\,(F\,f \times F\,g)^{-1}(\Gamma \wedge \Delta)R\,y \end{array}$$

- Finalmente para dos relaciones R, T tales que $R \subseteq T$, tenemos que $(\Gamma \wedge \Delta)R \subseteq (\Gamma \wedge \Delta)T$. Similar al caso anterior, partimos de elementos relacionados y llegamos a que también están relacionados. Sean x, y tales que $x(\Gamma \wedge \Delta)(R)y$.

$$\begin{array}{rl} & x(\Gamma \wedge \Delta)(R)y \\ \iff & x\ \Gamma(R)\ y \wedge x\ \Delta(R)\ y \\ \implies & x\ \Gamma(T)\ y \wedge x\ \Delta(T)\ y \\ \iff & x\ (\Gamma \wedge \Delta)(T)\ y \end{array}$$

□

La conjunción de relacionadores es una propiedad muy útil ya que nos permite adicionar información o restringir relacionadores adicionando más estructura. Por ejemplo, si utilizamos un relacionador Γ para observar una propiedad g y un relacionador Δ para observar una propiedad p, podemos utilizar el relacionador $\Gamma \wedge \Delta$ para observar la propiedad $g \wedge p$. En particular, como veremos más adelante, esta propiedad es la que nos permite introducir nuevas observaciones sobre la evaluación de términos con efectos.

Lamentablemente la disyunción de dos relacionadores **no** forma un relacionador. Esto es porque durante la composición de relaciones podemos estar relacionando elementos de forma incorrecta. Para esto veamos qué pasa si quisiéramos mostrar que la propiedad es válida. Dado un funtor F, Γ, Δ dos relacionadores de F, definimos a $\Gamma \vee \Delta$ como la unión de los relacionadores: dada una relación R, $(\Gamma \vee \Delta)(R) \doteq \Gamma(R) \cup \Delta(R)$. Veamos que, para dos relaciones R, S, $(\Gamma \vee \Delta)(R) \circ (\Gamma \vee \Delta)(S) \not\subseteq (\Gamma \vee \Delta)(R \circ S)$.

$$\begin{array}{rl} & x\ (\Gamma \vee \Delta)(R) \circ (\Gamma \vee \Delta)(S)\ z \\ \iff & \exists y, x\ (\Gamma \vee \Delta)(R)\ y \wedge y\ (\Gamma \vee \Delta)(S)\ z \\ \iff & \exists y, (x\ \Gamma(R)\ y \vee x\ \Delta(R)\ y) \wedge (y\ \Gamma(S)\ z \vee y\ \Delta(S)\ z) \\ \iff & \exists y, \begin{cases} x\ \Gamma(R)\ y \wedge y\ \Gamma(S)\ z \\ x\ \Gamma(R)\ y \wedge y\ \Delta(S)\ z \leftarrow \\ x\ \Delta(R)\ y \wedge y\ \Gamma(S)\ z \leftarrow \\ x\ \Delta(R)\ y \wedge y\ \Delta(S)\ z \end{cases} \end{array}$$

Como se puede observar, hay dos casos donde se mezclan las observaciones de Γ y Δ, $x\ \Gamma(R)\ y \wedge y\ \Delta(S)\ z, x\ \Delta(R)\ y \wedge y\ \Gamma(S)\ z$, señalizadas con unas flechas en la formula de arriba.

A modo de ejemplo definimos un relacionador para el funtor de fallo.

Ejemplo 7.1.1. *Sea $\mathbb{F}(X) = X + \bot$. Intuitivamente, la forma usual de mapear una relación R entre conjuntos X, Y, $R \subseteq X \times Y$, a una relación entre posibles elementos, es directamente relacionar por R cuando en ambos casos tengamos elementos para comparar.*

Sea $R \subseteq X \times Y$, definimos a $\Gamma R \subseteq \mathbb{F}(X) \times \mathbb{F}(Y)$ como:

$$x \; \Gamma R \; y \iff \begin{cases} x' R y' \wedge x = \iota_l(x') \wedge y = \iota_l(y') \\ x = \iota_r(\bot) \wedge y = \iota_r(\bot) \end{cases}$$

En este caso es fácil verificar las propiedades de los relacionadores, y se dejan como ejercicio para el lector.

7.2. Relacionador de Aproximaciones

Nuestro objetivo no es simplemente mapear relaciones, sino que queremos utilizar los relacionadores para introducir la observación realizada por la noción de aproximación de términos. El uso de relacionadores es simplemente una herramienta para poder establecer comparaciones entre valores con efectos, y en particular, efectos interpretados por mónadas ordenadas. Lo que nos falta es poder seguir dicho orden con el objetivo de eventualmente definir una teoría de mejoras con efectos.

Repasando la definición de mejoras, tenemos que un término N es mejorado por un término M, si (para todo contexto) la evaluación de N (en ése contexto) termina con un costo, entonces la evaluación de M (en ése contexto) es como mínimo tan eficiente como la de N. Si abstraemos la idea de una relación que compara costos, lo que nos queda es tener un mecanismo que nos permite comparar elementos si la evaluación de N termina entonces la de M también. La relación de *si termina entonces termina* es lo que codificaremos dentro de los relacionadores, ya que dentro de la interpretación de los efectos tenemos una noción de terminación. La mónadas ordenadas tienen un elemento distinguido, $\bot$, que nos permite distinguir entre computaciones que terminan o divergen.

Veamos a modo de ejemplo posibles relacionadores para los funtores y efectos que utilizamos dentro de esta tesis.

Ejemplo 7.2.1 (Relacionador de Fallo)**.** *Sean X, Y dos conjuntos, y $R \subseteq X \times Y$. Definimos a $\Gamma_\bot R \subseteq X_\bot \times Y_\bot$ como:*

$$x \; \Gamma_\bot R \; y \iff \begin{cases} x = \iota_l(u) \implies y = \iota_l(v) \wedge uRv \\ x = \iota_r(\bot_X) \end{cases}$$

El mapeo de relaciones definido por $\Gamma_\bot$ cumple con la definición de relacionador. Además podemos ver en la definición como se codifica la noción de *si termina entonces termina*, y en el caso de tener un término cuya evaluación diverge, identificado como $\iota_r(\bot)$, simplemente están relacionadas con todos los elementos. Formalizaremos esta noción en la siguiente sección.

Continuando con los ejemplos, definimos un relacionador para cada uno de los funtores subyacente de cada mónada vista en la sección anterior.

Ejemplo 7.2.2 (Relacionador de Excepciones)**.** *Sea E un conjunto de símbolos no interpretados que representan las diferentes excepciones. Definimos el*

relacionador $\Theta_E R \subseteq (X + E) \times (Y + E)$ *como:*

$$x\ \Theta_E R\ y \iff \begin{cases} x = \iota_l(u) \implies y = \iota_l(v) \wedge u\ R\ v \\ x = \iota_r(e) \implies y = \iota_r(e) \end{cases}$$

Utilizamos la composición de relacionadores para definir finalmente a $\Gamma_E \doteq \Theta_E \circ \Gamma_\perp$.

Definir de forma composicional relacionadores ayuda a simplificar las pruebas, y nos permite modularizar las observaciones que se irán realizando. En éste caso, primero observamos la divergencia en la evaluación de los términos con $\Gamma_\perp$, y en el caso de que las evaluaciones converjan, comparamos si han terminado correctamente o con una excepción con el relacionador Θ_E.

Ejemplo 7.2.3 (Relacionador de No-Determinismo)**.** *Dados dos conjuntos* X, Y*, y una relación* $R \subseteq X \times Y$*. Definimos el relacionador de no-determinismo* $\Gamma_{\mathcal{P}} R \subseteq \mathcal{P}(X) \times \mathcal{P}(Y)$ *como:*

$$P\ \Gamma_{\mathcal{P}} R\ Q \iff \forall x \in P, \exists y \in Q, x\ R\ y$$

El relacionador de no-determinismo define el mapeo tradicional e intuitivo de una relación $R \subseteq X \times Y$ a una relación entre partes de dichos conjuntos $\Gamma R \subseteq \mathcal{P}(X) \times \mathcal{P}(Y)$. No es la única forma de hacerlo, y como veremos más adelante, hay diferentes relacionadores para el funtor de no-determinismo. Exploraremos diferentes opciones en el Capítulo 8.

Finalmente, presentamos el relacionador para el funtor de sub-distribución de probabilidades.

Ejemplo 7.2.4 (Relacionador de Probabilidades)**.** *Para conjuntos* X, Y *y* $R \subseteq X \times Y$*. Definimos a* $\Gamma_{\mathcal{D}} R \subseteq \mathcal{D}(X) \times \mathcal{D}(Y)$ *como:*

$$\mu\ \Gamma_{\mathcal{D}} R\ \nu \iff \forall U \subseteq X, \mu(U) \leq \nu(R(U))$$

donde $R(U) = \{y \in Y \mid x\ R\ y\}$*, y para cualquier sub-distribución* δ*, y un conjunto* S*,* $\delta(S) = \sum_{s \in S} \delta(s)$.

El relacionador de probabilidades es más complejo de probar en comparación con los demás que hemos visto, y probar que es efectivamente un relacionador incluye el uso del teorema de Strassen (1965).

7.3. Aproximación de Programas con Efectos

En el Capítulo 4 vimos como mediante la simulación de Abramsky podíamos comparar términos utilizando la relación de evaluación. Se introducía una noción similar al principio extensional de equivalencia de funciones debido a que los valores dentro del cálculo lambda son interpretados como funciones.

Recapitulamos brevemente lo visto en la Sección 4.2, donde construimos una relación de aproximación aplicativa para términos del lenguaje. Debido a que el lenguaje se define utilizando dos categorías sintácticas, la relación de aproximación aplicativa nos queda de la siguiente forma:

$$V \leq_{\mathcal{V}} W \iff \forall U \in \mathcal{V}_0, V\ U \leq_\Lambda W\ U$$
$$M \leq_\Lambda N \iff \forall V \in \mathcal{V}_0, M \Downarrow V \implies \exists W \in \mathcal{V}_0, N \Downarrow W \wedge V \leq_{\mathcal{V}} W$$

Al final del capítulo, vimos que está definición es correcta, es decir, que la relación $(\leq_\Lambda, \leq_\mathcal{V})$ está bien definida de forma coinductiva.

En esta sección mostramos cómo obtener una relación de aproximación de términos pero teniendo en cuenta los efectos computacionales generados por la evaluación de términos.

Definición 7.3.1 (Simulación Γ)**.** *Sean X, Y dos conjuntos, un funtor F y Γ un relacionador de F. Una relación $R \subseteq X \times Y$ es una simulación Γ si y solo si existen funciones de inyección $\gamma_X : X \to F(X), \gamma_Y : Y \to F(Y)$, tales que:*

$$\forall x \in X, y \in Y, x \; R \; y \implies \gamma_x(x) \; \Gamma R \; \gamma_y(y)$$

Definición 7.3.2 (Similaridad Γ)**.** *Para dos conjuntos X, Y, un funtor F, dos funciones que inyectan elementos en F, $\gamma_X : X \to F(X), \gamma_Y : Y \to F(Y)$, y un relacionador Γ de F. Definimos como similaridad Γ a la simulación Γ más grande, y la notamos como $\succeq^{\Gamma,\gamma}_{X,Y}$. En símbolos, tenemos que para toda simulación Γ en X, Y con inyectores γ_X, γ_Y, $R^\Gamma_{X,Y} \subseteq \succeq^\Gamma_{X,Y}$.*

Notamos directamente a $(\succeq^\Gamma)$ como a la relación de similaridad en Γ cuando los conjuntos y funciones inyectoras se puedan deducir por contexto o puedan quedar libres.

La relación de similaridad Γ es un preorden. Si bien una relación entre programas no necesariamente tiene que ser reflexiva, en este caso sí queremos que sea transitiva. Una relación entre programas transitiva nos permite modificar nuestro programa en sucesivas etapas, simplificando las pruebas, para finalmente obtener un programa con las características que buscamos.

En nuestro caso la reflexividad es un requerimiento debido a como hemos definido la noción de mejora, *. . . la evaluación es a lo sumo tan lenta como. . .* que permite que un programa se mejore a sí mismo de forma trivial. Aunque, esto puede cambiar si refinamos la definición a *mejoras estrictas*, donde sería necesario que la evaluación de la mejora sea estrictamente más eficiente.

Lema 7.3.1. *Dado un conjunto X, F un funtor, Γ un relacionador de F, y una función, $\gamma : X \to F(X)$, de inyección de elementos en F. La relación $(\succeq^{\Gamma,\gamma}_{X,X})$ es un preorden.*

Demostración. Sea entonces X un conjunto, F un funtor, Γ un relacionador de F y $\gamma : X \to F(X)$ una función que inyecta elementos en F. La relación $\succeq^\Gamma$ es un preorden, es decir, reflexiva y transitiva. Esto lo logramos por la naturaleza coinductiva de la definición de la relación.

- La relación de similaridad Γ es reflexiva. Nos basta con probar que la identidad es una relación similar Γ, y por lo tanto, contenida en la relación similaridad Γ. Sea $x \in X$, tenemos que $\gamma(x) \; \mathbf{1}_{F(X)} \; \gamma(x)$, y por propiedad Rel-1 de relacionadores, $\gamma(x) \; \Gamma \mathbf{1}_X \; \gamma(x)$. Por lo que $\mathbf{1}_X \subseteq \Gamma \mathbf{1}_X$, y por lo tanto $\mathbf{1}_X$ es una simulación Γ.

- La relación de similaridad Γ es transitiva. Sean $x, y, z \in X$, tales que, $x \succeq y$ y $y \succeq z$. Por definición de similaridad, existe dos simulaciones, R y S, tales que: $x \; R \; y$ y $y \; S \; z$, que por composición de relaciones es equivalente a $x \; R \circ S \; z$. Dado que R, S son simulaciones, tenemos que $\gamma(x) \; \Gamma R \; \gamma(y)$ y que $\gamma(y) \; \Gamma S \; \gamma(z)$, que por composición de relaciones

tenemos que: $\gamma(x)\ \Gamma R \circ \Gamma S\ \gamma(z)$. Ahora, Γ es un relacionador, y por Rel-2 de relacionadores, tenemos que $\gamma(x)\ \Gamma(R \circ S)\ \gamma(z)$. Por lo que las simulaciones Γ son cerradas en la composición, y por lo tanto, contenidas en la relación de similaridad Γ, mostrando que $x \succeq z$.

□

Debido a que la evaluación de términos de lenguajes con efectos algebraicos es mónadica, debemos definir cómo los relacionadores interactúan con respecto a las mónadas. Para esto utilizamos relacionadores que interactúan adecuadamente con las mónadas, es decir, aquellos que extienden el comportamiento de las operaciones básicas de las mónadas.

Definición 7.3.3 (T-Relacionador)**.** *Sean X, X', Y, Y' conjuntos, T una mónada, $R \subseteq X \times Y$ y $S \subseteq X' \times Y'$ dos relaciones. Decimos que Γ es un T-relacionador para la mónada T si y solo si Γ es un relacionador para el funtor subyacente de T, y además:*

- $\forall x \in X, y \in Y, xRy \implies \eta_X(x) \Gamma R \eta_Y(y)$
- $\forall u \in T(X), v \in T(Y), \forall f : X \to T(X'), g : Y \to T(Y'),$

$$\left(\begin{array}{c}(\forall x \in X, y \in Y, xRy \implies f(x) \Gamma S g(y)) \\ \wedge \\ u\,\Gamma R\,v\end{array}\right) \implies (u >\!\!>\!\!= f) \Gamma S (v >\!\!>\!\!= g)$$

Finalmente, dado que la evaluación de términos está definida a través de aproximaciones de ω-cadenas, y dado que la simulación aplicativa incluye evaluar términos, necesitamos restringir aún más los relacionadores a aquellos que respeten las ω-cadenas. En particular, al mapear relaciones entre elementos a relaciones entre elementos con efectos, necesitamos que estos respeten la estructura de las mónadas ordenadas.

Definimos la noción de relación inductiva, que define cuando una relación respeta la estructura de un domino.

Definición 7.3.4 (Relación Inductiva)**.** *Sea $(D, \sqsubseteq, \bot)$ un dominio, E un conjunto y $R \subseteq D \times E$ una relación. Decimos que R es inductiva si y solo si:*

- *La relación R respeta el elemento $\bot$: $\forall e \in E, \bot\ R\ e$*
- *para toda ω-cadena $\{u_n\}_{n<\omega}$ en D, y $v \in E$:*

$$(\forall n < \omega, u_n\ R\ v) \implies (\bigsqcup_{n<\omega} u_n)\ R\ v$$

Extendemos la definición de relación inductiva respecto a la familia de dominios definida por el orden de la mónada.

Definición 7.3.5 (Relacionador Inductivo)**.** *Dada una mónada ordenada T por $(\sqsubseteq)$ y Γ un T-relacionador de T. Decimos que Γ es un relacionador inductivo si y solo si para cualesquiera conjuntos X e Y, y una relación $R \subseteq X \times Y$, ΓR es una relación inductiva para el dominio $(T(X), \sqsubseteq_X, \bot_X)$.*

Ya disponemos de todas las herramientas para poder definir la noción de simulación aplicativa con efectos, por lo que compactamos toda la maquinaria en una definición.

Definición 7.3.6 (Sistema Σ)**.** *Dada una signatura Σ. Llamamos sistema Σ a un par compuesto de una mónada Σ-continua T y un T-relacionador inductivo Γ para T.*

Ejemplo 7.3.1 (Sistema No-Determinista)**.** *Veamos, a modo de ejemplo, la definición del sistema para el efecto de no-determinismo.*

Por un lado veamos que el relacionador de no-determinismo (Ejemplo 7.2.3) es un T-relacionador para la mónada de no-determinismo, definida directamente como el operador partes de conjuntos $\mathcal{P}$ (Definición 6.4.1). Sean X, X', Y, Y' conjuntos, $f : X \to \mathcal{P}(X'), g : Y \to \mathcal{P}(Y')$ dos funciones, y dos relaciones $R \subseteq X \times Y, S \subseteq X' \times Y'$. Asumimos además que para todos dos elementos relacionados por R, $x\ R\ y$, la aplicación de las funciones f y g respectivamente, relaciona elementos en S a través del relacionador $\Gamma_{\mathcal{P}}$, $f(x)\ \Gamma_{\mathcal{P}} S\ g(y)$.

- *Por un lado veamos que para todos dos elementos $x \in X, y \in Y$, tales que $x\ R\ y$, tenemos que al inyectarlos η_X y η_Y respectivamente, están $\Gamma_{\mathcal{P}} R$ relacionados. Ahora por definición del morfismo η, tenemos que $\eta_X(x) = \{x\}$ y $\eta_Y(y) = \{y\}$. Finalmente, por definición del relacionador $\Gamma_{\mathcal{P}}$ y que $x\ R\ y$, tenemos que $\{x\}\ \Gamma_{\mathcal{P}} R\ \{y\}$.*

- *Por otro lado, veamos que el relacionador $\Gamma_{\mathcal{P}}$ mantiene la relación a través de la operación de* **bind**. *Sean $A \in \mathcal{P}(X)$ y $B \in \mathcal{P}(Y)$, tales que $A\ \Gamma_{\mathcal{P}} R\ B$, y por definición de $\Gamma_{\mathcal{P}}$ tenemos que, para todo a en A, existe b en B, tales que $a\ R\ b$. Nuestro objetivo es probar que: $(A >\!\!>\!= f)\ \Gamma_P S\ (B >\!\!>\!= g)$.*

  Por definición del operador **bind** *tenemos que:*

  1. $A >\!\!>\!= f \equiv \bigcup_{a \in A} f(a)$
  2. $B >\!\!>\!= g \equiv \bigcup_{b \in B} g(b)$

  Sea $x \in (A >\!\!>\!= f) \equiv \bigcup_{a \in A} f(a)$, es decir, $\exists a \in A, x \in f(a)$. Por hipótesis, tenemos que existe $b \in B$ con $a\ R\ b$, y más aún, tenemos que $f(a)\ \Gamma_{\mathcal{P}} S\ g(b)$. Por lo que tenemos lo que queríamos probar, para todo elemento $x \in (A >\!\!>\!= f)$, tenemos un elemento $y \in (B >\!\!>\!= g)$, tales que $x\ S\ y$, y por lo tanto $(A >\!\!>\!= f)\ \Gamma_{\mathcal{P}} S\ (B >\!\!>\!= g)$.

Nos falta entonces comprobar que $\Gamma_{\mathcal{P}}$ es inductivo respecto con el orden de la mónada $\mathbb{ND}$. Recordemos que, para cada conjunto X, se define el dominio de subconjuntos de X, ordenados por inclusión y con elemento $\perp$ como el conjunto vacío: $(\mathcal{P}(X), (\subseteq), \emptyset)$. Nos queda entonces por probar que para una relación $R \subseteq X \times Y$, $\Gamma_{\mathcal{P}} R$ es inductiva en dicho dominio.

- *La relación respeta el elemento $\perp$: $\forall B \subseteq Y, \perp\ \Gamma_{\mathcal{P}} R\ B$. Es fácil de comprobar, ya que se cumple por vacuidad. Tenemos que para todo elemento $x \in \emptyset$, existe un elemento $b \in B$, tal que $x\ R\ b$.*

- *La relación respeta las ω-cadenas. Sea $\{A_n\}_{n<\omega}$ una ω-cadena en $\mathcal{P}(X)$, y $B \in \mathcal{P}(Y)$, debemos mostrar que si B está relacionado por $\Gamma_{\mathcal{P}} R$ con todos los elementos de la cadena, también está relacionado con su supremo. Partimos por asumir que todo elemento de la cadena se $\Gamma_{\mathcal{P}} R$-relaciona con B: $\forall n < \omega, A_n\ \Gamma_{\mathcal{P}} R\ B$. Sea $a \in \bigsqcup_{n<\omega} A_n \equiv \bigcup_{n<\omega} A_n$, por lo que existe $m < \omega$ tal que $a \in A_m$. Por hipótesis, tenemos que $A_m\ \Gamma_{\mathcal{P}} R\ B$, y por lo tanto, existe $b \in B$, tal que $a\ R\ b$, mostrando lo que queríamos. Para todo $a \in \bigsqcup_{n<\omega} A_n$, existe $b \in B$, $a\ R\ b$, y por lo tanto, $(\bigsqcup_{n<\omega} A_n)\ \Gamma_{\mathcal{P}} R\ B$.*

Finalmente concluimos entonces que $\Gamma_{\mathcal{P}}$ es un T-relacionador inductivo para la mónada de no-determinismo $\mathbb{ND}$.

Cada uno de los ejemplos presentados hasta el momento definen un sistema, es decir, tenemos el sistema parcial dado por la mónada parcial y el relacionador $\Sigma_\perp$, el sistema de excepciones dado por la mónada de excepciones para un conjunto E y el relacionador Γ_E, y finalmente, el sistema probabilístico dado por la mónada de sub-probabilidades y el relacionador $\Gamma_{\mathbb{D}}$.

7.4. Simulación Aplicativa

Disponemos de todas las herramientas para definir la noción de simulación entremezclando los efectos generados por los operadores algebraicos introducidos en el lenguaje. A modo de recordatorio, la simulación aplicativa entre términos (sin efectos) la definimos coinductivamente utilizando la función (parcial) de evaluación, y comparando extensionalmente los valores obtenidos como resultados. En este caso modificamos la función de evaluación, ya que debe ser capaz de interpretar los efectos generados por los operadores algebraicos, y además, utilizamos los recién introducidos relacionadores para relacionar los resultados obtenidos de la evaluación.

Definición 7.4.1 (Γ simulación aplicativa con efectos para λ-términos)**.** *Sea Σ una signatura y (T, Γ) un Σ-sistema. Un par de relaciones cerradas $(R_\Lambda, R_\mathcal{V})$ es una Γ-simulación aplicativa con efectos si y solo si:*

- $\forall M, N \in \Lambda_0, M\ R_\Lambda\ N \implies [\![M]\!]\ \Gamma R_\mathcal{V}\ [\![N]\!]$
- $\forall V, W \in \mathcal{V}_0, V\ R_\mathcal{V}\ W \implies \forall U \in \mathcal{V}_0, (V\,U)\ R_\Lambda\ (W\,U)$

La definición caracteriza directamente la idea de simulación de Abramsky: por un lado, tenemos que se relacionan términos que evalúan a valores relacionados (modulo sus efectos), y por el otro, tenemos que los valores son relacionados de forma extensional. Notaremos a la relación de Γ simulación aplicativa con efectos para el cálculo lambda como $(\succ)$, y más aún, la llamaremos simplemente Γ-simulación aplicativa.

A partir de la definición de simulación Γ, podemos extraer un operador monótono, ya que los relacionadores son monótonos (Relacionadores Rel-4) respecto a la inclusión de conjuntos, de la misma forma que hicimos en la Definición 4.2.4. En este caso, simplemente daremos la definición ya que la prueba es similar a la anterior más el uso de las propiedades de los relacionadores.

Definición 7.4.2. *Sea Σ una signatura, (T, Γ) un Σ-sistema. Definimos el operador $\mathcal{O}_\Gamma$ de par de relaciones en par de relaciones:*

$$\begin{aligned}
\mathcal{O}_\Gamma &: Rel(\Lambda_0) \times Rel(\mathcal{V}_0) \to Rel(\Lambda_0) \times Rel(\mathcal{V}_0) \\
\mathcal{O}_{\mathcal{V}_0}(\mathcal{R}_{\mathcal{V}_0}, \mathcal{R}_{\Lambda_0}) &= \{(V, U) : \forall W \in \mathcal{V}_0, V\,W\ \mathcal{R}_{\Lambda_0}\ U\,W\} \\
\mathcal{O}_{\Lambda_0}(\mathcal{R}_{\mathcal{V}_0}, \mathcal{R}_{\Lambda_0}) &= \{(M, N) : [\![M]\!]\ \Gamma\ \mathcal{R}_{\mathcal{V}_0}\ [\![N]\!]\}
\end{aligned}$$

con $\mathcal{R}_{\Lambda_0} \subseteq Rel(\Lambda_0), \mathcal{R}_{\mathcal{V}_0} \subseteq Rel(\mathcal{V}_0)$.

El operador $\mathcal{O}_\Gamma$ es monótono, dado que los relacionadores son monótonos respecto a la contención de conjuntos por definición. Como resultado, tenemos que, por el teorema de Knaster-Tarski (Teorema 2.1.6), la noción de Γ similaridad está bien definida como el mayor punto fijo $\nu(\mathcal{O}_\Gamma)$ de $\mathcal{O}_\Gamma$. Tenemos además un principio de co-inducción: para probar que dos términos son aplicativos similares por Γ, basta con encontrar una simulación aplicativa Γ que los relacione.

Dado que todas las definiciones de simulación tendrán que cumplir con la segunda formula de la Definición 7.4.1, le daremos un nombre para simplificar la notación.

Definición 7.4.3 (Respeta Valores)**.** *Dada una relación cerrada de λ-términos $(R_\Lambda, R_\mathcal{V})$. Decimos que* $(R_\Lambda, R_\mathcal{V})$ respeta valores *si y solo si para todo par de valores $V, W \in \mathcal{V}_0$, tales que $V\ R_\mathcal{V}\ W$ tenemos que para todo valor $U \in \mathcal{V}_0$, $V\,U\ R_\Lambda\ W\,U$.*

Por último podemos instanciar la definición a los diferentes efectos que hemos visto a lo largo de la tesis.

Ejemplo 7.4.1 (Parcialidad)**.** *Una relación de λ-términos $(R_\Lambda, R_\mathcal{V})$ es una $\Gamma_\perp$-simulación aplicativa si y solo si respeta valores y además:*

$$\forall M, N \in \Lambda_0, M\ R_\Lambda\ N \implies \begin{cases} [\![M]\!]_\perp = \perp_\mathcal{V} \\ [\![M]\!]_\perp = V \implies [\![N]\!]_\perp = U \wedge V\ R_\mathcal{V}\ U \end{cases}$$

Ejemplo 7.4.2 (Excepciones)**.** *Dado un conjunto E de símbolos representando excepciones, tenemos que una relación de λ-términos $(R_\Lambda, R_\mathcal{V})$ es una Γ_E-simulación aplicativa si y solo si respeta valores y además:*

$$M\ R_\Lambda\ N \implies \begin{cases} [\![M]\!]_E = \iota_r(\perp_\mathcal{V}) \\ [\![M]\!]_E = \iota_l(in_r(e_1)) \implies [\![N]\!]_E = \begin{pmatrix} \iota_l(\iota_r(e_2)) \\ \wedge\ e1 = e2 \end{pmatrix} \\ [\![M]\!]_E = \iota_l(\iota_l(V)) \implies [\![N]\!]_E = \iota_l(\iota_l(U)) \wedge V\ R_\mathcal{V}\ U \end{cases}$$

Ejemplo 7.4.3 (No-determinismo)**.** *Una relación cerrada de λ-términos $(R_\Lambda, R_\mathcal{V})$ es una $\Gamma_\mathcal{P}$-simulación aplicativa si y solo si respeta valores y además:*

$$M\ R_\Lambda\ N \implies \forall V \in [\![M]\!]_{\mathbb{ND}}, \exists W \in [\![N]\!]_{\mathbb{ND}}, V\ R_\mathcal{V}\ W$$

Ejemplo 7.4.4 (Sub-distribuciones de Probabilidad)**.** *Una relación cerrada de λ-términos $(R_\Lambda, R_\mathcal{V})$ es una $\Gamma_\mathcal{D}$-simulación aplicativa si y solo si respeta valores y además:*

$$M\ R_\Lambda\ N \implies \forall \mathcal{U} \subseteq \mathcal{V}_0, [\![M]\!]_\mathcal{D}(\mathcal{U}) \leq [\![N]\!]_\mathcal{D}(R_\mathcal{V}(\mathcal{U}))$$

Como podemos observar por las diferentes nociones de simulación aplicativa que derivamos para cada uno de los efectos de ejemplos, se sigue el mismo espíritu que la clásica definición de Abramsky agregando además la comparación de valores con efectos mediante el uso de los relacionadores.

La relación de simulación aplicativa nos provee una forma práctica pero acotada para comparar términos y valores. Un caso particular es que la simulación aplicativa es entre términos y valores cerrados, aunque podemos extenderla a

términos y valores abiertos. Pero, más importante, al introducir efectos a la idea de simulación aplicativa, estamos reduciendo aún más su utilidad, en el sentido que términos que son equivalentes no siempre son evidenciados por una simulación aplicativa. La idea de simulación aplicativa podemos pensarla como una secuencia de evaluaciones hasta encontrar valores que están relacionados: sean M, N dos términos cerrados, los evaluamos con su respectiva función de evaluación para obtener valores cerrados con efectos, V, W respectivamente, los cuales tenemos que probar que están relacionados aplicativamente, es decir, que para todo valor cerrado U, $V\,U$ está relacionado con $W\,U$. El problema es que cada una de estas evaluaciones es independiente de la anterior, pero ¿qué pasaría si al tener efectos estas instancias deben estar relacionadas de alguna forma? Por ejemplo, asumiendo que el efecto conlleva a tener una noción de estado que se va construyendo a medida que se van evaluando los términos, en el caso de la simulación aplicativa, comenzamos con un estado inicial cada vez que evaluamos un término, y lo perdemos en la siguiente evaluación, en la que se comienza del estado inicial. Es decir, podemos llegar a necesitar una noción de *contexto* que sea consistente durante toda la evaluación o algún tipo de mecanismo que nos permita relacionar dichas evaluaciones sucesivas. El estudio de técnicas que permitan construir contextos para realizar pruebas de simulación aplicativa con información queda fuera del alcance de la tesis.

7.5. Aproximación Observacional

En esta sección, similar a la anterior, definimos la relación de aproximación observacional de programas con efectos. El objetivo entonces es obtener las nociones definidas en la Sección 4.1, pero en este caso, para lenguajes con efectos algebraicos.

Hemos visto que si bien las definiciones contextuales son muy expresivas, también son difíciles de probar ya que incluye en la definición una cuantificación universal sobre todos los posibles contextos. En la Sección 4.3, hemos visto como Howe desarrolló un método para construir relaciones precongruentes a partir de clausulas de compatibilidad. Esto nos permite introducir requerimientos inductivos para obtener relaciones algebraicas que sean modulares, y así, poder determinar cuando un término, M, aproxima observacionalmente a otro, N, construyendo las observaciones a partir de observar los sub-términos de M y N respectivamente. En esta sección seguiremos un camino similar, pero introduciendo además los posibles efectos que se pueden generar durante la evaluación de términos.

Comenzaremos por caracterizar los relacionadores que siguen la noción de cláusulas compatibles de Howe.

Definición 7.5.1 (Relacionador Σ-Compatible)**.** *Sea Σ una signatura, T una mónada y Γ un T-relacionador de T. Decimos que Γ es compatible con Σ (Σ-compatible) si y solo si para cualquiera relación $R \subseteq X \times Y$, para cada uno de los operadores σ en Σ con aridad $n = \alpha(\sigma)$, tenemos que:*

$$(\forall k \leq n, u_k\ \Gamma R\ v_k) \implies (\sigma^{T(X)}(u_1, \ldots, u_n))\ \Gamma R\ (\sigma^{T(Y)}(v_1, \ldots, v_n))$$

La definición de un relacionador compatible con Σ caracteriza a los T-relacionadores que respetan a los operadores de la signatura Σ. En palabras,

tenemos que al construir términos a partir de elementos relacionados, obtenemos términos relacionados.

Al momento de definir la función de evaluación establecimos ciertos requerimientos sobre la interacción entre la mónada T, el relacionador Γ y las interpretaciones de las operaciones en Γ. En particular, llamamos a un sistema Σ a una mónada Σ-continua T y un T-relacionador inductivo Γ de T. Lo que tenemos es que para toda función $f : X \to T(Y)$, su extensión de Kleisli, $f^\dagger : T(X) \to T(Y)$ es continua, y por lo tanto, Γ respeta los operadores de Σ por definición.

Lema 7.5.1 (Remark 4 (Dal Lago, Gavazzo y Paul Levy 2017))**.** *Sea Σ una signatura, T una mónada y Γ una relacionador, tales que, (T, Γ) forman un Σ-sistema. El relacionador Γ es compatible con Σ.*

Para lo que queda de la sección, asumiremos que tenemos una signatura Σ y un Σ-sistema (T, Γ).

Definimos la noción de relación abierta, ya que la propiedad de términos cerrados no es modular. Para ver la falta de modularidad, notar que en el término dentro de una lambda abstracción la variable ligada puede ocurrir de forma libre.

Definición 7.5.2 (Relación Abierta)**.** *Una relación abierta sobre términos es un par $(\overline{x}, R_\Lambda) \in \mathcal{P}(Var) \times (\Lambda \times \Lambda)$ donde $R_\Lambda \subseteq \Lambda(\overline{x}) \times \Lambda(\overline{(x)})$. Una relación abierta sobre valores es una par $(\overline{x}, R_\mathcal{V}) \in \mathcal{P}(Var) \times (\mathcal{V} \times \mathcal{V})$ donde $R_\mathcal{V} \subseteq \mathcal{V}(\overline{x}) \times \mathcal{V}(\overline{x})$.*

Notamos como $\overline{x} \vdash M \ R_\Lambda \ N$ a que $(\overline{x}, R_\Lambda)$ es una relación abierta de términos y $(M, N) \in R_\Lambda$, y $(\overline{x}, V, W) \vdash V \ R_\mathcal{V} \ W$ a que $(\overline{x}, R_\mathcal{V})$ es una relación abierta de valores y $(V, W) \in R_\mathcal{V}$.

Tenemos todas las herramientas para definir la noción de relaciones *compatibles* siguiendo el método de Howe.

Definición 7.5.3 (Compatibilidad)**.** *Sea $(R_\Lambda, R_\mathcal{V})$ una relación entre λ-términos y valores. Decimos que $(R_\Lambda, R_\mathcal{V})$ es una relación compatible si y solo si dado un conjunto de variables $\overline{x}$ se cumplen las siguientes clausulas:*

$$\textit{Comp-Var}\frac{}{\overline{x} \vdash x \ R_\mathcal{V} \ x} \qquad \textit{Comp-Abs}\frac{\overline{x} \cup \{x\} \vdash M \ R_\Lambda \ N}{\overline{x} \vdash (\lambda x.M) \ R_\mathcal{V} \ (\lambda x.N)}$$

$$\textit{Comp-Ret}\frac{\overline{x} \vdash V \ R_\mathcal{V} \ W}{\overline{x} \vdash \mathbf{ret}(V) \ R_\Lambda \ \mathbf{ret}(W)}$$

$$\textit{Comp-App}\frac{\overline{x} \vdash V \ R_\mathcal{V} \ V' \qquad \overline{x} \vdash W \ R_\mathcal{V} \ W'}{\overline{x} \vdash (V \, W) \ R_\Lambda \ (V' \, W')}$$

$$\textit{Comp-To}\frac{\overline{x} \vdash M \ R_\Lambda \ M' \qquad \overline{x} \cup \{x\} \vdash N \ R_\Lambda \ N'}{\overline{x} \vdash \mathbf{let} \ x = M \ \mathbf{in} \ N \ R_\Lambda \ \mathbf{let} \ x = M' \ \mathbf{in} \ N'}$$

$$\textit{Comp-}\sigma\frac{\overline{x} \vdash M_1 \ R_\Lambda \ N_1 \qquad \ldots \qquad \overline{x} \vdash M_k \ R_\Lambda \ N_k}{\overline{x} \vdash \sigma(M_1, \ldots, M_k) \ R_\Lambda \ \sigma(N_1, \ldots, N_k)}$$

La regla Comp-σ representa en realidad un conjunto de reglas, una por cada símbolo σ en Σ y $k = \alpha(\sigma)$.

Las reglas de compatibilidad explícitamente definen relaciones que son algebraicas, en el sentido que nos permiten relacionar términos grandes a partir de relacionar los sub-términos de estos. Cuando utilizamos constructores de términos, esto es equivalente a decir que la relación es cerrada bajo los constructores de los términos. Sea R una relación compatible, tenemos que, para todo par de términos $M, N \in \Lambda_{\overline{x}}$ y para todo $\mathbb{C}$ tal que $\mathbb{C}[M], \mathbb{C}[N] \in \Lambda(\overline{y})$:

$$\overline{x} \vdash M \; R \; N \implies \overline{y} \vdash \mathbb{C}[M] \; R \; \mathbb{C}[N]$$

Las variables libres en la precondición pueden ser diferentes a las variables libres de la poscondición, $\overline{x}$ e $\overline{y}$ respectivamente. Esto es debido a que podemos aplicar constructores que liguen ciertas variables libres e incluso también introducir nuevas variables libres a los términos. Recordar que los contextos pueden ligar variables.

Definición 7.5.4 (Precongruencia)**.** *Sea $(R_\Lambda, R_\mathcal{V})$ una relación entre λ-términos y valores. Decimos que $(R_\Lambda, R_\mathcal{V})$ es una precongruencia si y solo si $(R_\Lambda, R_\mathcal{V})$ es un preorden compatible.*

Definición 7.5.5 (Relaciones Preadecuadas)**.** *Una relación R es preadecuada si y solo si para todo par de términos cerrados $M, N \in \Lambda_0$ tenemos que:*

$$M \; R \; N \implies [\![M]\!] \; \Gamma\mathcal{U} \; [\![N]\!]$$

donde $\mathcal{U} = \mathcal{V}_0 \times \mathcal{V}_0$ es la relación que relaciona todo valor cerrado con cualquier otro.

En palabras tenemos que una relación es preadecuada si la evaluación de los términos que relaciona resultan en valores con efectos relacionados a través de Γ sin importar los valores en sí. En otras palabras, solo se observan que los efectos generados por la evaluación estén relacionados. Una relación entre λ-términos y valores $(R_\Lambda, R_\mathcal{V})$ es preadecuada si y solo si R_Λ lo es.

Definición 7.5.6. *Sea $\mathbb{CA}$ el conjunto de relaciones entre λ-términos que son compatibles y preadecuadas. Definimos entonces la relación $(\geq_\Gamma)$ como la unión de todas las relaciones que son compatibles y preadecuadas.*

La relación $(\geq_\Gamma)$ es una relación que es cerrada respecto a los constructores del lenguaje. Es decir, dados dos términos que están relacionados, al introducirse dentro de cualquier contexto mantienen la relación, pero además al evaluarlos generan efectos relacionados por Γ.

De nuevo, estamos ante una relación construida de forma coinductiva a partir de dos propiedades dentro del reticulado de los conjuntos ordenados por la inclusión. Por lo cual es fácil ver que la relación que obtenemos mantiene las propiedades establecidas.

Lema 7.5.2. *La relación $(\geq_\Gamma)$ es un preorden compatible preadecuado.*

Obtenemos el principio de coinducción: dado que la relación $(\geq_\Gamma)$ es la unión de todas las relaciones compatibles y preadecuadas, para probar que $M \geq_\Gamma N$, basta con probar que existe una relación compatible y preadecuada que relacione a M con N.

Preordenes Contextuales

Para cada unos de los sistemas que hemos visto podemos presentar como son los preordenes contextuales.

Ejemplo 7.5.1 (Parcialidad)**.** *El sistema parcial, utilizando la mónada de parcialidad equipada con el relacionador* $\Gamma_\perp$*, define un preorden contextual* $(\geq_\perp)$*. Sean* $M, N \in \Lambda$ *dos términos cualesquiera, posiblemente abiertos,* $M \geq_\perp N$ *si y solo si para todo contexto* $\mathbb{C}$ *tal que* $\mathbb{C}[M], \mathbb{C}[N] \in \Lambda_0$*, tenemos que:*

- *O bien, la evaluación del término* $\mathbb{C}[M]$ *diverge;*
- *O bien, si la evaluación del término* $\mathbb{C}[M]$ *converge a un valor, entonces también lo hace la evaluación del término* $\mathbb{C}[N]$*, en símbolos:*

$$\forall V \in \mathcal{V}_0, [\![\mathbb{C}[M]]\!]_\perp = \iota_r(V) \implies \exists W \in \mathcal{V}_0, [\![\mathbb{C}[N]]\!]_\perp = \iota_r(W)$$

El ejemplo que acabamos de ver es la definición clásica de aproximación observacional para el cálculo lambda no tipado (Felleisen 1991). No se establece ninguna relación sobre los valores resultantes de la evaluación de los términos, sino que se comparan los efectos que se producen al evaluarlos, en éste caso la divergencia. Esto se debe a que la relación es preadecuada, y por definición, $\mathcal{U} = \mathcal{V}_0 \times \mathcal{V}_0$.

Ejemplo 7.5.2 (Excepciones)**.** *El sistema de excepciones, para un conjunto de símbolos* E*, define un preorden contextual donde para cualquier par de términos* M, N*,* $M \geq_E N$ *si y solo si para todo contexto* $\mathbb{C}$ *tal que* $\mathbb{C}[M], \mathbb{C}[N] \in \Lambda_0$*, tenemos que uno de los siguientes casos es verdadero:*

- *O bien la evaluación del término* $\mathbb{C}[M]$ *termina sin excepciones, y por lo tanto, la evaluación del término* $\mathbb{C}[N]$ *también termina sin excepciones:*

$$\forall V \in \mathcal{V}_0, [\![\mathbb{C}[M]]\!]_E = \iota_l(\iota_l(V)) \implies \exists W \in \mathcal{V}_0, [\![\mathbb{C}[N]]\!]_E = \iota_l(\iota_l(W))$$

- *O bien la evaluación del término* $\mathbb{C}[M]$ *termina en una excepción, y por lo tanto, la evaluación del término* $\mathbb{C}[N]$ *termina* ***con la misma excepción****:*

$$\forall e \in E, [\![\mathbb{C}[N]]\!]_E = \iota_l(\iota_r(e)) \implies [\![\mathbb{C}[N]]\!]_E = \iota_l(\iota_r(e))$$

- *O bien la evaluación del término* $\mathbb{C}[M]$ *diverge:*

$$[\![\mathbb{C}[M]]\!]_E = \iota_r(\perp)$$

En este caso, se analizan los casos dependiendo de los diferentes efectos observables, el caso de que la evaluación converge a un valor, a una excepción, y el caso de que la evaluación diverja. Además, dado que observamos la excepción que se lanza, tenemos que observar que es exactamente la misma. Esto se debe a la definición del relacionador del sistema de excepciones.

Ejemplo 7.5.3 (No-determinismo)**.** *Siguiendo el sistema de no-determinismo, para cual quiera par de términos* M, N*,* $M \geq_{\mathcal{ND}} N$ *si y solo si, para todo contexto* $\mathbb{C}$ *tales que* $\mathbb{C}[M], \mathbb{C}[N] \in \Lambda_0$ *tenemos que:*

$$[\![\mathbb{C}[M]]\!]_{\mathcal{ND}} \neq \emptyset \implies [\![\mathbb{C}[N]]\!]_{\mathcal{ND}} \neq \emptyset$$

En este caso, a diferencia de los anteriores, solo nos basta con observar que la evaluación del término $\mathbb{C}[M]$ tenga un posible resultado, lo que es equivalente a observar que el conjunto de resultados posible no sea vacío, y en cuyo caso, la evaluación del término $\mathbb{C}[N]$ también tiene que tener al menos un resultado posible. En el caso que la evaluación de $\mathbb{C}[M]$ diverja, en cuyo caso el conjunto de resultados sería vacío, el antecedente sería falso, y por ende, se cumpliría trivialmente.

Por último vemos el caso del sistema probabilístico utilizando la mónada de sub-distribuciones de probabilidad.

Ejemplo 7.5.4 (Sub-distribuciones)**.** *Siguiendo las definiciones del sistema probabilístico, tenemos que para dados dos términos M, N, $M \geq_{\mathcal{D}} N$ si y solo si para todo contexto $\mathbb{C}$ tal que $\mathbb{C}[M], \mathbb{C}[N] \in \Lambda_0$ tenemos que:*

$$[\![\mathbb{C}[M]]\!]_{\mathcal{D}}(\mathcal{V}_0) \geq [\![\mathbb{C}[N]]\!]_{\mathcal{D}}(\mathcal{V}_0)$$

En ese caso, al igual que los demás, se busca observar directamente la probabilidad de que la evaluación de un término converja. Al tener una sub-distribución de probabilidades, simplemente se calcula la probabilidad que tiene una evaluación de un término en alcanzar cualquier valor, y por eso se utiliza todo el conjunto $\mathcal{V}_0$.

7.6. Aproximación Aplicativa y Aproximación Observacional

Finalmente, para cerrar la relación entre las dos aproximaciones, presentamos un teorema que conecta la noción de simulación aplicativa con la de preorden contextual. Por un lado tenemos la definición intuitiva de aproximación observacional que define que dos términos tienen un comportamiento observacional similar cuantificado sobre todo contexto. Por el otro lado tenemos la noción de simulación aplicativa que se centra en la evaluación y comparación del resultado mediante el uso de un relacionador del sistema. Esta última relación se basa en la idea de que los valores se pueden interpretar como funciones, y por ende, comparar extensionalmente. La simulación aplicativa es muy útil para presentar aproximaciones de forma local al término, es decir, que son independientes del contexto (y por ende en general son términos cerrados), mientras que la relación de aproximación observacional es la versión más general. Lo que nos lleva a establecer una conexión entre ambos resultados.

Teorema 7.6.1. *La relación de Γ-similaridad es una precongruencia y además es preadecuada, y por lo tanto, $(\succeq_\Gamma) \subseteq (\geq_\Gamma)$.*

Lo que nos dice el teorema es que dados dos términos que son Γ-similares, siempre podemos tomar esa relación y extenderla con los contextos necesarios para obtener una relación preadecuada que es una precongruencia.

La prueba se basa en el uso del método de Howe (1996). El método consiste en construir una nueva relación precongruente por definición basada en la relación Γ-similaridad y probar que ambas relaciones son equivalentes. Para encontrar una prueba completa se recomienda visitar la bibliografía (Dal Lago, Gavazzo y Paul Levy 2017).

En el cálculo lambda clásico se puede mostrar que los términos que las relaciones de aproximación observacional y la simulación aplicativa son equivalentes. Lamentablemente, al introducir efectos algebraicos tenemos que no siempre es el caso que las relaciones sean equivalente. Esto, depende en gran medida de los efectos que se incorporen al lenguaje, y de las observaciones que se realicen por el relacionador. Al introducir la noción de simulación de Abramsky, la introdujimos como una secuencia de evaluaciones relacionando los valores intermedios obtenidos de forma extensional y volviendo a utilizar la misma relación. Esto quiere decir que hasta llegar a valores que se puedan relacionar directamente, tal vez hayamos pasado por varias de estas simulaciones. Cada una de estas simulaciones, comienzan como una simulación fresca, en el sentido que toda la información *y efectos* que se hayan generado e interpretado en simulaciones anteriores no estarán presentes en la actual. Esta es la principal diferencia con la aproximación observacional, donde al haber una sola evaluación de los términos, todos los efectos acumulados son manejados por la misma instancia de la mónada.

Finalmente, a su vez, el teorema conectando ambas nociones de aproximaciones, nos presenta además un método válido para probar que un término aproxima observacionalmente a otro simplemente mostrando que *existe una* simulación aplicativa que los muestre.

Capítulo 8

Teoría de Mejoras con Efectos

En este capítulo vemos el aporte principal de la tesis, donde presentamos *una forma* de realizar análisis de propiedades intensivas sobre la evaluación de términos en presencia de efectos algebraicos. Esto lo logramos introduciendo primero un nuevo efecto: la computación del costo de la evaluación de un término acumulando *ticks*. Luego utilizamos un nuevo relacionador para comparar los costos de la evaluación de un término con otro, y así, obtener una relación de mejoras entre programas con efectos. Esta simple idea nos permite derivar desde la definición de equivalencia entre programas la definición de mejoras tal cual dada por Sands (1998).

8.1. Análisis Intensivo Derivado

Exploramos las definiciones derivadas a partir de introducir el análisis de costos de la evaluación de términos dentro de las relaciones de aproximación contextual y de la simulación aplicativa. Lo que hacemos es adicionar como un efecto el cómputo de costo de la evaluación, y luego, introducimos la observación del mismo para obtener dos nuevas relaciones: *similaridad aplicativa con efectos y costos* y *mejoras con efectos*. Para esto, agregamos una nueva operación llamada *tick* (✓) que agrega una unidad de costo a la evaluación de un término. De esta manera llevamos a cabo el análisis de costo simplemente acumulando los ticks necesarios por la evaluación de los términos y poder así compararlos al finalizar la evaluación.

Para simplificar la presentación de los resultados, asumiremos que tenemos una signatura Σ equipada con un sistema (T, Γ), que quedará fijo durante toda la sección.

Costos en la evaluación de Términos

Para agregar el efecto de acumular el costo de evaluar un término del lenguaje, lo que hacemos es seguir la idea intuitiva de simplemente equipar a cada elemento con un número natural que representa el costo requerido para computarlo.

Para simplificar la notación primero presentamos las operaciones sobre costo. Notamos con $\mathbb{N}_\infty$ al conjunto de los números naturales con un elemento adicional. Este elemento adicional, ∞, denota al supremo del conjunto $\mathbb{N}_\infty$.

Definición 8.1.1. *Sea X un conjunto, definimos una familia de funciones add_n indexada por $n \in \mathbb{N}$ de la siguiente forma:*

$$\begin{aligned} add_n &: (X \times \mathbb{N}_\infty) \to (X \times \mathbb{N}_\infty) \\ add_n(t, c) &= (t, c + n) \end{aligned}$$

Definimos la familia de funciones que dentro de la mónada T acumula los ticks que se han encontrado hasta el momento:

$$\begin{aligned} \checkmark^n &: T(X \times \mathbb{N}_\infty) \to T(X \times \mathbb{N}_\infty) \\ \checkmark^n &= T\, add_n \end{aligned}$$

Finalmente, definimos una mónada basándonos en la mónada T de forma tal que equipe a todo elemento dentro de la mónada con un número natural representando el costo requerido para computarlo. Valores puros (sin efectos) son introducidos con un costo 0, pero al utilizar el operador **bind**, sumamos el costo de computar su primer argumento con el costo de computar el segundo.

Definición 8.1.2 (Mónada de Costo Interno)**.** *Definimos la mónada $T_{\mathbb{N}_\infty} = (T_{\mathbb{N}_\infty}, \eta, >\!\!>\!\!=)$ de la siguiente forma:*

$$\begin{aligned} T_{\mathbb{N}_\infty}(X) &= T(X \times \mathbb{N}_\infty) \\ \eta_{T_{\mathbb{N}_\infty}}(x) &= \eta_T(x, 0) \\ m >\!\!>\!\!=_{\mathbb{N}_\infty} f &= m >\!\!>\!\!=_T (v, c) \mapsto \checkmark^c(f(v)) \end{aligned}$$

Esta forma de *transformar* una mónada en otra es un método muy conocido dentro de la comunidad del lenguaje de programación Haskell, y se la puede encontrar bajo el nombre de *transformador de mónada writer con el monoide aditivo de los números naturales* (M. Jaskelioff y Moggi 2010a; Liang, Hudak y Jones 1995). Otras combinaciones de transformadores y mónadas podrán dar lugar a otros análisis de costos, estudiaremos diferentes nociones en la Sección 9.4. De todas maneras, el transformador de mónadas writer tiene la ventaja, como veremos, que nos permite introducir análisis de costo de programas *sin tener que modificar el sistema*. En otras palabras, de esta manera derivamos de la definición de aproximación de programas un análisis de costos simplemente equipando cada elemento con el costo necesario para computarlo.

Debido a que utilizamos un transformador de mónadas, que lo que nos permite es componer dos mónadas, primero vamos a estudiar el comportamiento de forma local, introduciendo la noción de costos utilizando un monoide aditivo y luego las compondremos utilizando el transformador.

Lema 8.1.1. *Dado un monoide $(M, \cdot, e)$ aditivo y un preorden $(\sqsubseteq) \subseteq M \times M$ compatible con la operación de M. La relación obtenida del producto punto a punto con $(\sqsubseteq)$, $(- \times \sqsubseteq)$, es un relacionador monádico para la mónada producto con M, $(- \times M)$.*

Demostración. La prueba consiste de dos partes: la primera es mostrar que es efectivamente un relacionador para el funtor subyacente a la mónada producto, y luego que es un relacionador mónadico.

La relación producto con $(\sqsubseteq)$ define un relacionador para el funtor subyacente de la mónada producto con M. Para esto debemos mostrar que se cumplen las 4 propiedades de los relacionadores. Sean X, Y, Z tres conjuntos.

- Contención de la identidad con efectos: $1_{X\times M} \subseteq 1_X \times \sqsubseteq$. Sea $(x, c) \in X \times M$, dado que la relación $(\sqsubseteq)$ es un preorden, entonces es reflexiva, y por lo tanto, $(x, c)\ 1_X \times \sqsubseteq\ (x, c)$.

- Para cualquier relación $R \subseteq X \times Y$ y $S \subseteq Y \times Z$, $(R \times \sqsubseteq) \circ (S \times \sqsubseteq) \subseteq ((R \circ S) \times \sqsubseteq)$.

  Sea $(x, p) \in X \times M$ y $(z, r) \in Z \times M$ tales que

  $$(x, p)\ (R \times \sqsubseteq) \circ (S \times \sqsubseteq)\ (z, r)$$

  Por definición de composición de relaciones tenemos que existen $y \in Y$ y $q \in M$ tal que:

  $$(x, p)\ (R \times \sqsubseteq)\ (y, q) \wedge (y, q)\ (S \times \sqsubseteq)\ (z, r)$$

  Aplicando la definición del producto de relaciones tenemos que:

  $$x\ R\ y \wedge (p \sqsubseteq q) \wedge y\ S\ z \wedge (q \sqsubseteq r)$$

  Por commutatividad del operador lógico $(\wedge)$, transitividad de $(\sqsubseteq)$ y definición de composición de relaciones, tenemos finalmente que:

  $$(x, p)\ ((R \circ S) \times \sqsubseteq)\ (z, r)$$

- Para cualquier conjunto W, Z y función $f : W \to X, g : Z \to Y$, tenemos que:

  $$((f \times g)^{-1} R \times \sqsubseteq) \subseteq ((\_ \times 1_M)\, f \times (\_ \times 1_M)\, g)^{-1} (R \times \sqsubseteq)$$

  Sean además $w \in W$,$z \in Z$, $p, q \in M$, tales que:

  $$(w, p)\ ((f \times g)^{-1} R \times \sqsubseteq)\ (z, q)$$

  Aplicamos la definición de producto de relaciones

  $$w\ ((f \times g)^{-1} R)\ z \wedge p \sqsubseteq q$$

  Por definición de aplicación inversa de funciones, tenemos que existen $x \in X, y \in Y$ tales que $f(w) = x$ y $g(z) = y$ y además:

  $$f(w)\ R\ g(z) \wedge p \sqsubseteq q$$

  Más aún, por definición de funtor de la mónada de producto, tenemos que $(\_ \times 1_M)\, f(w, p) = (f(w), p)$ y $(\_ \times 1_M)\, g(z, q) = (g(z), q)$, y por lo tanto:

  $$((\_ \times 1_M)\, f\, (w, p))\ R \times \sqsubseteq\ ((\_ \times 1_M)\, g\, (z, q))$$

  Finalmente aplicamos la definición las inversas de las funciones mapeadas por el funtor de la mónada producto:

  $$(w, p)\ ((\_ \times 1_M) f \times (\_ \times 1_M) g)^{-1} (R \times \sqsubseteq)\ (z, q)$$

- Por último, para toda relación $T \subseteq X \times Y$ tal que $S \subseteq T$, $(S \times \sqsubseteq) \subseteq (T \times \sqsubseteq)$. Asumiendo que $S \subseteq T$, sean $x \in X$, $y \in Y$, y $p, q \in M$, tales que:

  $$(x, p)\ S \times \sqsubseteq\ (y, q)$$

  Por definición de producto de relaciones, tenemos que:

  $$x\ S\ y \wedge p \sqsubseteq q$$

  Más aún, dado que $S \subseteq T$, tenemos que:

  $$x\ T\ y \wedge p \sqsubseteq q$$

  Finalmente por definición de producto de relaciones, tenemos que:

  $$(x, p)\ (T \times \sqsubseteq)\ (y, q)$$

  Lo que nos permite concluir que $(S \times \sqsubseteq) \subseteq (T \times \sqsubseteq)$.

Lo que nos queda por mostrar es que el relacionador producto con $(\sqsubseteq)$ además es mónadico. Es decir, que extiende el comportamiento de las operaciones de la mónada resultante de hacer el producto con el monoide M. Sean X', Y' conjuntos, $f : X \to (X' \times M)$ y $g : Y \to (Y' \times M)$ dos funciones, y $R \subseteq X \times Y$ y $S \subseteq X' \times Y'$ dos relaciones.

- La inyección de valores respeta relaciones mediante el relacionador. Sean $x \in X$ e $y \in Y$ tales que $x\ R\ y$, tenemos que $\eta_{X \times M}(x) = (x, e)$ y $\eta_{Y \times M}(y) = (y, e)$. Al ser la relación $(\sqsubseteq)$ un preorden, es reflexiva, y por lo tanto $e \sqsubseteq e$. Concluimos que $(x, e)\ R \times \sqsubseteq\ (y, e)$.

- Sea $(x, p) \in (X \times M)$ y $(y, q) \in (Y \times M)$ tales que $(x, p)\ R \times \sqsubseteq\ (y, q)$, o equivalentemente, $x\ R\ y \wedge p \sqsubseteq q$. Asumimos que para todos $x \in X$ e $y \in Y$ tales que $x\ R\ y$, tenemos que $f(x)\ S \times \sqsubseteq\ g(y)$. Tenemos que mostrar que $((x, p) >\!\!>\!\!= f)\ S \times \sqsubseteq\ ((y, q) >\!\!>\!\!= g)$. Por definición del operador **bind** de la mónada, esto es equivalente a probar que:

  $$((x, p) >\!\!>\!\!= f)\ S \times \sqsubseteq\ ((y, q) >\!\!>\!\!= g)$$

  aplicando la definición del operador **bind** de la mónada de producto tenemos

  $$(1_X, p \cdot)(f(x))\ (S \times \sqsubseteq)\ (1_Y, q \cdot)(g(y))$$

  Sean $(x', p') \in (X' \times M)$ y $(y', q') \in (Y' \times M)$ tales que $f(x) = (x', p')$ y $g(y) = (y', q')$, y además, dado que $x\ R\ y$, $(x', p')\ (S \times \sqsubseteq)\ (y', q')$, tenemos que probar entonces:

  $$(x', p \cdot p')\ (S \times \sqsubseteq)\ (y', q \cdot q')$$

  equivalentemente

  $$x'\ S\ y' \wedge (p \cdot p') \sqsubseteq (q \cdot q')$$

  Por un lado tenemos que $x'\ S\ y'$ por ser el resultado de la aplicación de las funciones f y g. Por el otro tenemos $p \sqsubseteq q$ y además $p' \sqsubseteq q'$, y por ser el preorden $\sqsubseteq$ compatible con la operación de M tenemos que $p \cdot p' \sqsubseteq q \cdot q'$. □

Para el resto de la sección, asumimos que tenemos un monoide aditivo $(M, \cdot, e)$ con un preorden que respete la operación del monoide. Notaremos además a la mónada $T \circ (\_ \times M)$ como $\mathbb{W}_{T,M}$.

Lo que haremos entonces en lo que queda de la sección es probar que $(\mathbb{W}_{T,M}, \Gamma \circ (\_ \times \sqsubseteq))$ forma un sistema válido para la signatura Σ. En otras palabras, queremos ver que la mónada $\mathbb{W}_{T,M}$ está ordenada y además que $\Gamma \circ (\_ \times \sqsubseteq)$ es un relacionador mónadico inductivo para $\mathbb{W}_{T,M}$.

Lema 8.1.2. *Para toda mónada ordenada T, la mónada resultante $\mathbb{W}_{T,M}$ es una mónada ordenada.*

Demostración. Sea $(\preceq)$ un orden de la mónada T. Por definición de mónadas ordenadas, tenemos que para cada conjunto X hay un ω**CPPO** $(T(X), \preceq_X, \bot_X)$. En particular, para todo conjunto X, $X \times M$ es también un conjunto, y por lo tanto, podemos utilizar el orden definido por $(\preceq)$ para el conjunto $X \times M$, y así obtener un ω**CPPO** $(T(X \times M), \preceq_{X\times M}, \bot_{X\times M})$. □

Lema 8.1.3. *El relacionador definido como la composición de Γ y el relacionador definido en el Lema 8.1.1, $\Gamma \circ (\_ \times \sqsubseteq)$, es un relacionador mónadico para la mónada $\mathbb{W}_{T,M}$.*

Demostración. Dado que la composición de relacionadores definen un relacionador por la composición de funtores, tenemos que el relacionador $\Gamma \circ (\_ \times \sqsubseteq)$ es un relacionador para el funtor subyacente de la mónada $\mathbb{W}_{T,M}$.

El operador de relaciones $\Gamma \circ (\_ \times \sqsubseteq)$ respeta las operaciones de la mónada $\mathbb{W}_{T,M}$. Sean X, X', Y, Y' conjuntos, $f : X \to \mathbb{W}_{T,M}(X')$ y $g : Y \to \mathbb{W}_{T,M}(Y')$ dos funciones, y $R \subseteq X \times Y, S \subseteq X' \times Y'$ dos relaciones.

- El relacionador $\Gamma \circ (\_\times \sqsubseteq)$ respeta la transformación natural η_{TM}. Sean $x \in X$ e $y \in Y$ dos valores tales que $x\ R\ y$, queremos ver que $\eta_{TM}(x)\ (\Gamma(R \times \sqsubseteq))\ \eta_{TM}(y)$.

$$
\begin{array}{l}
\quad \eta_{TM}(x) \\
\equiv \langle \text{definición del transformador de mónadas Writer} \rangle \\
\quad \eta_T(x, e) \\
\Gamma(R \times \sqsubseteq) \langle \Gamma \text{ es un } T\text{-relacionador para la mónada } T \text{ y } (x,e)\ (R \times \sqsubseteq)\ (y,e) \\
\quad \eta_T(y, e) \\
\equiv \langle \text{definición del transformador de mónadas Writer} \rangle \\
\quad \eta_{TM}(y)
\end{array}
$$

- El relacionador $\Gamma \circ (\_ \times \sqsubseteq)$ respeta el operador **bind** de la mónada $\mathbb{W}_{T,M}$. Sean $u \in \mathbb{W}_{T,M}(X), v \in \mathbb{W}_{T,M}(Y)$ tales que $u\ \Gamma(R \times \sqsubseteq)\ v$ y f, g forman un homomorfismo entre R y $\Gamma(S \times \sqsubseteq)$.

$$
\begin{array}{l}
\quad (u >\!\!>\!\!=_{\mathbb{W}_{T,M}} f)\ \Gamma(S \times \sqsubseteq)\ (v >\!\!>\!\!=_{\mathbb{W}_{T,M}} g) \\
\equiv \langle \text{definición del transformador de mónadas Writer} \rangle \\
\quad (u >\!\!>\!\!=_T (u_X, u_M) \mapsto T\,(u_m \cdot \_)\, f(u_X)) \\
\quad\ \ \Gamma(S \times \sqsubseteq) \\
\quad (v >\!\!>\!\!=_T (v_X, v_M) \mapsto T\,(v_m \cdot \_)\, g(v_X))
\end{array}
$$

Notamos a las funciones derivadas de f y g que además acumulan los costos computados como:

$$f'(u, m) \doteq T\,(u_m \cdot \_)\, f(u_X) \tag{8.1}$$
$$g'(u, m) \doteq T\,(v_m \cdot \_)\, g(v_X) \tag{8.2}$$

Dado que Γ es un T-relacionador para la mónada T, y por hipótesis tenemos que $u\ \Gamma(R \times \sqsubseteq)\ v$, nos queda por mostrar que para todo $(x, m_X) \in (X \times M)$ e $(y, m_Y) \in (Y \times M)$ tales que $(x, m_X)\ R \times \sqsubseteq\ (y, m_Y)$, tenemos que $f'(x, m_X)\ \Gamma(S \times \sqsubseteq)\ g'(y, m_Y)$. Sea $(x, m_X) \in (X \times M)$ y $(y, m_Y) \in (Y \times M)$ tales que $(x, m_X)\ R \times \sqsubseteq\ (y, m_Y)$. El hecho que $f'(x, m_X)\ \Gamma(S \times \sqsubseteq)\ g'(y, m_Y)$ se desprende de que Γ es un T-relacionador para la mónada T, $m_X \sqsubseteq m_Y$, y que f, g forman un homomorfismo entre las relaciones R y $\Gamma(S \times \sqsubseteq)$. En símbolos tenemos que:

$$\begin{aligned}
&(x, m_X)\ R \times \sqsubseteq\ (y, m_Y) \\
\equiv&\langle \text{definición producto de relaciones} \rangle \\
&x\ R\ y \wedge m_X \sqsubseteq m_Y \\
\Longrightarrow&\langle \text{hipótesis} \rangle \\
&f(x)\ \Gamma(S \times \sqsubseteq)\ g(y) \wedge m_X \sqsubseteq m_Y \\
\Longrightarrow&\langle \Gamma \text{ es un } T\text{-relacionador de } T \rangle \\
&T\,(m_X \cdot _)\, f(x)\ \Gamma(S \times \sqsubseteq)\ T\,(m_y \cdot _)\, g(y) \\
\equiv&\langle \text{definiciones de } f', g' \rangle \\
&f'((x, m_x))\ \Gamma(S \times \sqsubseteq)\ g'((y, m_y))
\end{aligned}$$

□

La mónada $\mathbb{W}_{T,M}$ es Σ-continua. Se desprende directamente de que T sea Σ-continua. Más aún, la transformación que estamos realizando *restringe* los conjuntos posibles (los valores de la categoría Set). Como resultado de esta sección, tenemos que para un monoide $(M, \cdot, e)$ y un preorden $(\sqsubseteq) \subseteq M \times M$, que respete al monoide, la mónada $\mathbb{W}_{T,M}$ y el relacionador mónadico $\Gamma \circ (\_ \times \sqsubseteq))$ forman un Σ-sistema.

Finalmente, concluimos que dado un Σ-sistema derivamos otro sistema en el cual se puede razonar sobre los costos de la evaluación de términos.

Mapear Relaciones con nociones de Costo

En esta sección definimos cómo mapear relaciones entre valores a relaciones que además comparen los costos, con el objetivo de utilizar relaciones que razonen sobre el costo de la evaluación. Dado que equipamos valores con el costo requerido para computarlos, utilizamos el relacionador definido en el Lema 8.1.3, resultante de la composición del relacionador del sistema con el orden del monoide aditivo. Como a lo largo de la tesis utilizamos el mismo monoide aditivo, aquel de los números naturales más infinito, ∞, a continuación mostramos la definición de su relacionador.

Definición 8.1.3 (Relacionador de Costo)**.** *Sean X, Y dos conjuntos y sea R una relación entre X e Y. Definimos la relación $R_{\mathbb{N}_\infty} \subseteq (X \times \mathbb{N}_\infty) \times (Y \times \mathbb{N}_\infty)$ como el producto de la relación R y $(\geq_\infty)$. En símbolos:*

$$(x, m)\ R_\infty\ (y, n) \iff x\ R\ y \wedge m \geq_\infty n$$

Podemos ver en la definición anterior cómo son comparados los costos de los valores equipados con sus costos. De esta manera, dada una relación R, el relacionador de costos de R lo que hace es refinar la relación para *además* comparar los costos. Este relacionador fue presentado por Dal Lago y Gavazzo (2019) que codifica la definición de mejoras propuesta por Sands (1998). Notamos al nuevo sistema, $(\mathbb{W}_{T,\mathbb{N}_\infty}, \Gamma \circ R_{\mathbb{N}_{T,\mathbb{N}_\infty}})$, simplemente como $(T_{\mathbb{N}_\infty}, \Gamma_{\mathbb{N}_\infty})$.

Teniendo un sistema de costos, nos concentraremos en introducir un nuevo operador a nivel del lenguaje para marcar computaciones costosas. De esta manera le permitimos a los usuarios del lenguaje agregar notaciones sobre el costo de los programas.

A continuación, probamos que no cambiamos la semántica del lenguaje al agregar un nuevo operador para adicionar costos a nivel del lenguaje, y así poder razonar sobre programas. Es decir, introducimos un nuevo operador ✓ a la signatura que notaremos $\Sigma_\checkmark$, y mostramos que tenemos un $\Sigma_\checkmark$-sistema. El nuevo operador unario del lenguaje lo podemos interpretar como un tick dentro de la mónada de costos, es decir directamente como la función $\checkmark^1$ definida como el mapeo de la función add_1 (Definición 8.1.1).

Lema 8.1.4. *La mónada de costo interno $T_{\mathbb{N}_\infty}$ es $\Sigma_\checkmark$-continua.*

Demostración. Dado que por definición $\Sigma_\checkmark \equiv \Sigma \cup \{\checkmark\}$, podemos dividir la prueba en dos:

- La mónada $T_{\mathbb{N}_\infty}$ es Σ-continua. Sigue de que la mónada T es Σ-continua.
- La interpretación del operador unario ✓ es continua. La función de interpretación $\checkmark^1$ se define como el mapeo de una función sobre valores y costos add_1, y sumado a que las operaciones de la mónada son continuas por definición de mónada continua, tenemos que la función $\checkmark^1$ también lo es.

□

En la prueba anterior se puede ver un procedimiento que utilizaremos siempre que trabajemos con la mónada de costos interna. La mónada $\mathbb{W}_{T,\mathbb{N}_\infty}$ es el resultado de apilar los efectos, pero en particular, introducir un nuevo efecto dentro de los de T. Por lo que al momento de realizar pruebas, en general, aplicamos la definición de la mónada $\mathbb{W}_{T,\mathbb{N}_\infty}$, exponiendo el hecho que sean efectos apilados, y así obtenemos elementos dentro de T. En otras palabras, tenemos que para resultados universales dentro de T, podemos mapear dicho resultado a $\mathbb{W}_{T,\mathbb{N}_\infty}$.

Lema 8.1.5. *El relacionador $\Gamma_{\mathbb{N}_\infty}$ es un T-relacionador inductivo para la mónada $\Sigma_\checkmark$-continua $T_{\mathbb{N}_\infty}$.*

Demostración. Seguimos el mismo razonamiento que en el Lema 8.1.4. □

Teorema 8.1.1. *El par* $(T_{\mathbb{N}_\infty}, \Gamma_{\mathbb{N}_\infty})$ *es un* $\Sigma_\checkmark$*-sistema.*

Demostración. Por Lema 8.1.4 y Lema 8.1.5. □

Evaluación con Costos

Contamos con todas las herramientas para definir la relación de $\Sigma_{\mathbb{N}_\infty}$-similaridad aplicativa con efectos y aproximación observacional con costos para los términos del lenguaje $(\Lambda, \mathcal{V})$. A modo de ejemplo, introducimos una relación de evaluación instrumentada que será la encargada de ir introduciendo ticks a medida que vaya evaluando los términos en valores mónadicos, simplemente acumulando el costo de consumir un constructor del lenguaje. Una alternativa a contar reducciones sobre los términos, es contar solamente la cantidad de aplicaciones de funciones que fueron realizadas (Sands 1991) o incluso asignarles diferentes costos a operaciones que son computacionalmente pesadas, como ser: comunicarse por red o consultas a oráculos. Diferentes formas de instrumentar la relación de evaluación posiblemente lleve a diferentes relaciones de mejoras.

Definición 8.1.4 (Relación de Evaluación Instrumentada Aproximada)**.** *Definimos una familia de relaciones indexadas por un número natural* $n \in \mathbb{N}$ *entre términos cerrados y valores con efectos* $(\Downarrow_n^{\mathbb{N}T})$ *donde instrumentamos con tick la evaluación de cada uno de los constructores del lenguaje.*

$$\bot\frac{}{M \Downarrow_0^{\mathbb{N}T} \bot} \qquad (\texttt{ret})\frac{}{\mathbf{ret}(V) \Downarrow_{n+1}^{\mathbb{N}T} \checkmark \eta_{\mathbb{N}_\infty}(V)}$$

$$(\texttt{seq})\frac{M \Downarrow_n^{\mathbb{N}T} X \qquad N[x := V] \Downarrow_n^{\mathbb{N}T} Y_V}{\mathbf{let}\ x = M\ \mathbf{in}\ N \Downarrow_{n+1}^{\mathbb{N}T} \checkmark X >\!\!>\!\!=_{\mathbb{N}_\infty} (V \rightsquigarrow Y_V)}$$

$$(\texttt{app})\frac{M[x := W] \Downarrow_n^{\mathbb{N}T} X}{(\boldsymbol{\lambda}\ x\ .\ M)\, W \Downarrow_{n+1}^{\mathbb{N}T} \checkmark X}$$

$$(\checkmark\texttt{-op})\frac{M \Downarrow_n^{\mathbb{N}T} X}{\checkmark M \Downarrow_{n+1}^{\mathbb{N}T} \checkmark X} \qquad (\sigma\texttt{-op})\frac{M_1 \Downarrow_n^{\mathbb{N}T} X_1 \quad \dots \quad M_k \Downarrow_n^{\mathbb{N}T} X_k}{\sigma(M_1, \dots, M_k) \Downarrow_{n+1}^{\mathbb{N}T} \checkmark \sigma^T(X_1, \dots, X_k)}$$

Donde la regla $(\sigma\texttt{-op})$ *es en realidad un conjunto de reglas, una para cada operador* $\sigma \in \Sigma$ *y* $k = \alpha(\sigma)$*.*

Siguiendo los pasos realizados en los Capítulos 3 y 6 podemos derivar una relación de evaluación tal que para todo término cerrado $M \in \Lambda_0^\Sigma$, $[\![M]\!]_{\mathbb{N}_\infty} \in T_{\mathbb{N}_\infty}(\mathcal{V}_0)$. Para evitar la repetición dejamos las pruebas necesarias fuera del presente documento.

Lema 8.1.6 (Ecuaciones de la Relación de Evaluación Instrumentada)**.** *La relación de evaluación mónadica instrumentada del* Σ*-sistema* (T, Γ) *respeta las*

siguientes ecuaciones:

$$\begin{aligned}
[\![\mathbf{ret}(V)]\!]_{\mathbb{N}_\infty} &= \checkmark \eta_{\mathbb{N}_\infty}(V) \\
[\![(\boldsymbol{\lambda}\ x\ .\ M)\, W]\!]_{\mathbb{N}_\infty} &= \checkmark [\![M[x := W]]\!]_{\mathbb{N}_\infty} \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]_{\mathbb{N}_\infty} &= \checkmark [\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} (v \mapsto [\![N[x := v]]\!]_{\mathbb{N}_\infty}) \\
[\![\checkmark M]\!]_{\mathbb{N}_\infty} &= \checkmark [\![M]\!]_{\mathbb{N}_\infty} \\
[\![\sigma(M_1, \ldots, M_n)]\!]_{\mathbb{N}_\infty} &= \checkmark \sigma^{T_{\mathbb{N}_\infty}}([\![M_1]\!]_{\mathbb{N}_\infty}, \ldots, [\![M_n]\!]_{\mathbb{N}_\infty})
\end{aligned}$$

Las principales diferencias con la relación de evaluación dada en la Sección 6.3 son:

- la evaluación de un tick en el lenguaje se interpreta como adicionar un tick en la acumulación de costos, de esta manera evitamos que se cuenten dos veces.
- se suma un tick por cada vez que se consume un constructor del término

Siguiendo los Capítulos 4 y 7, definimos una familia de relaciones de aproximación sobre la evaluación de términos, aunque en este caso tendremos la observación extra sobre el costo de las computaciones.

En lo que queda del capítulo reproducimos los resultados obtenidos en el Capítulo 5 teniendo en cuenta los efectos con el objetivo de obtener una teoría de mejoras en lenguajes con efectos.

8.2. Simulación de Mejoras

La relación de similaridad aplicativa sigue la idea presentada por Abramsky, donde define dos términos similares si lo son modulo evaluación, y la hemos caracterizado coinductivamente para términos sin efectos en la Sección 4.2 y con efectos en la Sección 7.3. En esta sección lo que haremos es observar además de los efectos producidos por las operaciones algebraicas del lenguaje, el costo resultante de la evaluación de los términos. Llamamos *simulación de mejoras* a la relación de simulación aplicativa resultante de utilizar un sistema que lleve cuenta de los costos de las computaciones. En este sentido, la simulación de mejoras está compuesta de dos observaciones (independientemente del sistema): la comparación de términos como la simulación de Abramsky y la comparación de costos.

Definición 8.2.1 (Simulación de Mejoras)**.** *Decimos que una relación cerrada entre λ-términos y valores $(R_\Lambda, R_\mathcal{V})$ es una simulación de mejoras si y solo si respeta valores y además:*

$$\forall M, N \in \Lambda_0, M\ R_\Lambda\ N \implies [\![M]\!]_{\mathbb{N}_\infty}\ \Gamma_{\mathbb{N}_\infty} R_\mathcal{V}\ [\![N]\!]_{\mathbb{N}_\infty}$$

Sea $(R_\Lambda, R_\mathcal{V})$ una simulación de mejoras, M y N dos términos cerrados, tales que $M\ R_\Lambda\ N$. Desenvolviendo la definición de simulación de mejoras, tenemos que la evaluación de ambos términos están relacionados de forma tal que: $[\![M]\!]_{\mathbb{N}_\infty}\ \Gamma_{\mathbb{N}_\infty} R_\mathcal{V}\ [\![N]\!]_{\mathbb{N}_\infty}$. Más aún, por la definición del relacionador de costos (Definición 8.1.3) tenemos que $[\![M]\!]_{\mathbb{N}_\infty}\ (\Gamma R_\mathcal{V})_{\mathbb{N}_\infty}\ [\![N]\!]_{\mathbb{N}_\infty}$, en palabras:

- el resultado de evaluar el término M y N, $[\![M]\!]_{\mathbb{N}_\infty}$ y $[\![N]\!]_{\mathbb{N}_\infty}$, tiene los efectos relacionados mediante la definición del relacionador Γ de T.
- el costo de la evaluación de M, encapsulado en $[\![M]\!]_{\mathbb{N}_\infty}$ es mayor o igual que el costo de la evaluación de N, encapsulado en $[\![N]\!]_{\mathbb{N}_\infty}$.

Mónada de Parcialidad

La mónada de parcialidad presenta, en esta teoría, un ejemplo muy importante. Es el cálculo lambda tal cuál fue presentado al principio de la tesis (Capítulo 3) y su teoría de mejoras (Capítulo 5), donde el único efecto que se observa es el de la convergencia en la evaluación de un término. Concretamente, la mónada de parcialidad instancia la teoría en el caso que el único efecto observable es la divergencia de la evaluación de los términos, efecto necesario para poder establecer la noción de aproximación observacional. Para ver que el sistema parcial es el mismo que la noción clásica de mejoras, instanciamos las definiciones y las desenvolvemos, en particular la del relacionador $\Gamma_\perp$ definido en la Sección 7.3. Sea $(R_\Lambda, R_\mathcal{V})$ una relación entre λ-términos tal que sea una simulación de mejoras en el sistema parcial. Por la definición de simulación de mejoras (Definición 8.2.1) tenemos que para cuales quiera λ-términos cerrados M y N, si $M\ R_\Lambda\ N$ entonces:

- $[\![M]\!]_{\mathbb{N}_\infty} = \iota_r(\perp_{\mathcal{V}\times\mathbb{N}_\infty})$, o
- $[\![M]\!]_{\mathbb{N}_\infty} = \iota_l(V,n) \implies [\![N]\!]_{\mathbb{N}_\infty} = \iota_l(W,m) \wedge n \geq_{\mathbb{N}_\infty} m \wedge V\ R_\mathcal{V}\ W$

La definición resultante de instanciar la simulación de mejoras al sistema parcial es equivalente a la definición clásica de teoría de mejoras (Sands 1991). La diferencia principal yace en que esta última utiliza evaluación a *weak head normal form*, en lugar de la evaluación *eager* que seguimos en este trabajo.

8.3. Mejoras

La relación de aproximación observacional relaciona términos basándose en cómo estos se comportan en todo contexto. En otras palabras, un término A aproxima observacionalmente un término B si no hay un contexto capaz de *observar* una diferencia en el comportamiento de A y B. En los capítulos anteriores hemos explorado cuales eran las observaciones que podían realizarse, en nuestro caso sobre los efectos generados por la evaluación de operadores de efectos algebraicos. La noción de observación la encapsulamos directamente en el uso de los relacionadores, codificando primero la observación de no divergencia en la evaluación, y luego, dependiendo del sistema, los efectos generados por la evaluación. En esta sección, vamos a ver cómo al mezclar los efectos generados con la información de costos podemos derivar una noción de *mejoras* para poder hacer análisis intensivo sobre términos.

Definición 8.3.1 (Mejora)**.** *Sean M y N dos términos. Decimos que M es mejorado por N si y solo si $M \geq_{\Gamma_{\mathbb{N}_\infty}} N$.*

Notamos la relación de mejoras ($\geq_{\Gamma_{\mathbb{N}_\infty}}$) como $\succeq_\Gamma$, donde nos evitamos escribir el subíndice de costo.

Sistema Parcial

De la misma forma que utilizamos el sistema parcial para mostrar que al instanciar la definición de simulación de mejoras con efectos obtenemos la definición clásica, en éste caso vemos que al instanciar la definición de mejoras al sistema parcial obtenemos la definición de mejoras clásica. Como resultado del Lema 7.5.2 tenemos que la relación $\succeq_{\Gamma_\perp}$ es un preorden compatible y preadecuado. Por lo que, para todo término M y N y para todo contexto $\mathbb{C}$ que cierre a M y N, si $\emptyset \vdash \mathbb{C}[M] \succeq_{\Gamma_\perp} \mathbb{C}[N]$, tenemos que:

$$[\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty} \; \Gamma_{\perp\mathbb{N}_\infty}\mathcal{U} \; [\![\mathbb{C}[N]]\!]_{\mathbb{N}_\infty}$$

Más aún, por definición del relacionador de costos (Lema 8.1.3) y de $\Gamma_\perp$, dados términos M y N, $M \succeq_{\Gamma_\perp} N$ si y solo si para todo contexto $\mathbb{C}$ que cierre a M y N hay dos posibles casos:

- la evaluación de $\mathbb{C}[M]$ diverge: $[\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty} = \iota_r(\perp_{\mathcal{V}\times\mathbb{N}_\infty})$
- la evaluación de $\mathbb{C}[M]$ converge y es mejorada por la evaluación de $\mathbb{C}[N]$: $[\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty} = \iota_l(V, c_1) \implies [\![\mathbb{C}[N]]\!]_{\mathbb{N}_\infty} = \iota_l(W, c_2) \wedge c_1 \geq_\infty c_2$

En palabras, para dos términos M y N cualesquiera, tales que $M \succeq_{\Gamma_\perp} N$, para cualquier contexto $\mathbb{C}$ que los cierre, siempre que la evaluación de $\mathbb{C}[M]$ converja con costo c_1, la evaluación de $\mathbb{C}[N]$ también debe converger con un costo c_2, tal que c_1 es mayor o igual que c_2. Esta es la misma noción de mejoras dada para el cálculo lambda sin efectos (Sands 1998) como se puede observar al compararla con la definición de mejora (Definición 5.2.2) del Capítulo 5.

Finalmente por el Teorema 7.6.1, es suficiente con obtener una simulación de mejoras para mostrar mejoras. En otras palabras, nos provee un método para probar mejoras entre términos cerrados. Para probar que un término cerrado M es mejorado por otro término cerrado N, $M \succeq_\Gamma N$, basta con probar que M es similar a N. Más aún, probar que M es similar a N consiste en encontrar una simulación que evidencie la mejora. De esta manera es posible simplificar las pruebas, aunque dependiendo del sistema, como vimos al momento de introducir efectos (Sección 7.6), las simulaciones no siempre son suficientes para mostrar propiedades observacionales.

En el siguiente capítulo vemos cómo utilizar la teoría de mejoras con efectos para derivar *nuevas teorías de mejoras*. En particular, una teoría para cada uno de los efectos utilizados como ejemplo a lo largo de la tesis.

Capítulo 9

Nociones de Mejoras

Utilizando las definiciones vistas en el capítulo anterior (Capítulo 8), en este capítulo derivamos una *nueva* noción de mejoras para cada uno de los sistemas presentados como ejemplo a lo largo de la tesis. En particular derivamos nociones de mejoras y simulación de mejoras para los diferentes lenguajes presentados: excepciones, no-determinismo y operadores probabilísticos.

Primero adicionamos al sistema una noción de costos *sin tener que modificar* la definición del sistema. De esta manera reutilizamos la definición de aproximación de términos dada por el relacionador para además comparar costos sin tener que realizar pruebas adicionales. Luego, sobre el final del capítulo, analizamos mediante un ejemplo concreto presentando los límites del enfoque utilizado. Es decir, mostramos cómo la reutilización del relacionador podría llevar a una noción de mejoras poco granular. Esto nos lleva entonces a explorar una nueva alternativa a introducir costos en el lenguaje modificando el relacionador, y así, tener una observación sobre los costos más expresiva.

9.1. Excepciones

Sea E un conjunto de símbolos, $((X+E)_\perp, \Gamma_E)$ es un sistema Σ_E compatible. Más aún, siguiendo la definición de simulación de mejoras (Definición 8.2.1), una relación entre λ-términos $(R_\Lambda, R_\mathcal{V})$ es una simulación de mejoras para el sistema de excepciones si y solo si $R_\mathcal{V}$ respeta valores y para los términos cerrados M, N tenemos que, $M\ R_\Lambda\ N$ si y solo uno de los siguientes casos es verdadero:

- La evaluación de M diverge: $[\![M]\!]_{\mathbb{N}_\infty} = \iota_r(\perp)$
- La evaluación de M converge a una excepción e, y también lo hace la evaluación de N: $[\![M]\!]_{\mathbb{N}_\infty} = \iota_l(\iota_r(e)) \implies [\![N]\!]_{\mathbb{N}_\infty} = \iota_l(\iota_r(e')) \wedge e = e'$
- La evaluación de M converge, entonces la evaluación de N también converge a un valor relacionado con el resultante de la evaluación de M con menor o igual costo:

  $$[\![M]\!]_{\mathbb{N}_\infty} = \iota_l(\iota_l(V, n)) \implies [\![N]\!]_{\mathbb{N}_\infty} = \iota_l(\iota_l(W, m)) \wedge (n \geq m) \wedge (V\ R_\mathcal{V}\ W)$$

En palabras, dos términos cerrados M y N son similares si y solo si sus evaluaciones retornan valores extensionalmente similares y el costo de evaluar a M es mayor que el de N o ambas evaluaciones producen la *misma excepción*. Sin

embargo, notar que cuando la evaluación lanza una excepción *no se comparan los costos*. Esto es una consecuencia directa del uso de la mónada interna de costos *sin modificar el sistema*. Al utilizar la mónada interna de costos se equipan valores con costos, pero las excepciones no son valores, sino que son elementos que representan el efecto de lanzar dicha excepción. Exploramos una alternativa de llevar cuenta del costo de las evaluaciones en la Sección 9.4.

De la misma manera, decimos que un λ-término M es mejorado por un λ-término N, $M \succeq_E N$, si y solo si para todo contexto $\mathbb{C}$ que cierre a M y N, tenemos a lo sumo uno de los siguientes casos:

- La evaluación de $\mathbb{C}[M]$ converge, y por ende, la evaluación de $\mathbb{C}[N]$ también pero con menos costo:

$$\forall v \in \mathcal{V}_0, n \in \mathbb{N}_\infty, [\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty} = \iota_l(\iota_l(v, n)) \implies$$
$$\exists w \in \mathcal{V}_0, m \in \mathbb{N}_\infty, [\![\mathbb{C}[N]]\!]_{\mathbb{N}_\infty} = \iota_l(\iota_l(w, m)) \wedge (n \geq_\infty m)$$

- La evaluación de $\mathbb{C}[M]$ genera una excepción e, y por lo tanto, la evaluación de $\mathbb{C}[N]$ genera la misma excepción:

$$\forall e \in E, [\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty} = \iota_l(\iota_r(e)) \implies [\![\mathbb{C}[N]]\!]_{\mathbb{N}_\infty} = \iota_l(\iota_r(e))$$

- La evaluación de $\mathbb{C}[M]$ diverge:

$$[\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty} = \iota_r(\bot)$$

9.2. No-Determinismo

El sistema no-determinístico introduce un operador ($\oplus$) binario cuya evaluación retorna el valor de alguno de sus argumentos. Para interpretar este efecto utilizamos la mónada de partes donde vamos acumulando los posibles resultados que puede tomar la evaluación de un término. Existen al menos dos formas de definir la aproximación de computaciones no-determinístico que podemos codificarlas como dos relacionadores diferentes, y utilizando estas dos aproximaciones definir una tercera. Cada uno de estos relacionadores están inspirados por las diferentes relaciones sobre *powerdomains* presentadas por Søndergaard y Sestoft (1992).

Un trabajo más reciente que estudia la noción de similaridad en lenguajes con no-determinismo puede ser el de Lassen y Pitcher (1997) donde presentan dos relaciones: *may-converge* y *may-diverge*. En otras palabras, definen dos observaciones sobre la evaluación no-determinística de términos. Estas observaciones siguen un enfoque similar al que seguimos en esta tesis (Lassen y Pitcher 1997, Definición 3.1), pero combinan las observaciones de forma tal que obtienen una definición más granular. Pueden obtener definiciones más granulares ya que trabajan con una semántica operacional concreta para interpretar no-determinismo. En este trabajo, nos concentramos en cómo es observado el no-determinismo mediante el uso de relacionadores, y por lo tanto, nos basamos en las observaciones más básicas realizadas por Søndergaard y Sestoft, que se pueden comparar a las utilizadas en la relación de may-converge (Lassen y Pitcher 1997).

En esta sección instanciamos la definición del sistema no-determinístico con el objetivo de obtener una noción de simulación de mejoras y una de mejoras con efectos sobre *tres* diferentes relacionadores mónadicos para la mónada de no-determinismo.

Relación de costos de Hoare

El relacionador de Hoare representa una forma intuitiva de mapear una relación de valores a una relación entre conjunto de valores. Sea $R \subseteq X \times Y$ una relación, mapeamos la relación a conjuntos de valores, $A \subseteq X$ y $B \subseteq Y$, decimos que A está R-relacionado con B, si para cada elemento de A se relaciona con al menos uno de B. Cuando utilizamos este mapeo de relaciones para comparar la evaluación de términos no-determinístico, el relacionador de Hoare nos dice que un término M está relacionado con un término N si y solo si todo valor *alcanzable* por la evaluación de M es también alcanzable por la evaluación de N.

Definición 9.2.1 (Relacionador de Hoare (Dal Lago, Gavazzo y Paul Levy 2017))**.** *Sean X, Y dos conjuntos y R una relación entre ellos. Definimos el relacionador $\Gamma^H R \subseteq \mathcal{P}(X) \times \mathcal{P}(Y)$ donde:*

$$S \; \Gamma^H R \; T \iff \forall x \in S, \exists y \in T, x \; R \; y$$

Agregamos costos como un efecto observable sobre la evaluación de términos y equipamos el relacionador para poder comparar el costo de los valores. Utilizando las definiciones de la sección anterior podemos derivar una noción de mejora y de simulación de mejoras. Obtenemos que una relación entre λ-términos cerrados $(R_\Lambda, R_\mathcal{V})$ es una simulación de mejoras de Hoare si y solo si respeta valores y además para todo M, N términos cerrados tenemos que:

$$M \; R_\Lambda \; N \implies [\![M]\!]_{\mathbb{N}_\infty} \; \Gamma^H_{\mathbb{N}_\infty} R_\mathcal{V} \; [\![N]\!]_{\mathbb{N}_\infty}$$

Aplicando la definición del relacionador de costos, tenemos que:

$$M \; R_\Lambda \; N \implies [\![M]\!]_{\mathbb{N}_\infty} \; \Gamma^H R_{\mathcal{V}\mathbb{N}_\infty} \; [\![N]\!]_{\mathbb{N}_\infty}$$

Ahora, desenvolviendo la definición del relacionador de Hoare tenemos que:

$$M \; R_\Lambda \; N \implies \forall (V, m) \in [\![M]\!]_{\mathbb{N}_\infty}, \exists (W, n) \in [\![N]\!]_{\mathbb{N}_\infty}, V \; R_\mathcal{V} \; W \wedge m \geq_\infty n$$

En palabras, un término N es similar a un término M si y solo si siempre que la evaluación de M termine entonces también lo hace la de N, y además, para cada posible valor V que pueda tomar la evaluación de M, existe un valor W resultante de la evaluación de N, tales que W es similar a V y el costo n es menor o igual que m.

La definición de mejoras de Hoare se obtiene de instanciar la definición de mejoras con el relacionador de Hoare:

$$M \succeq_{\Gamma^H} N \iff \forall \mathbb{C} \in Ctx, \mathbb{C}[M, N] \in \Lambda_0, [\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty} \; \Gamma^H_{\mathbb{N}_\infty} \mathcal{U} \; [\![\mathbb{C}[N]]\!]_{\mathbb{N}_\infty}$$

En palabras, un término N mejora a un término M si y solo si para todo contexto $\mathbb{C}$ que los cierre, para todo valor V en la evaluación de $\mathbb{C}[M]$, hay un

valor W en la evaluación de $\mathbb{C}[N]$, tal que el costo requerido para obtener W es menor o igual al necesario para obtener V. En este caso solo se comparan los costos y los efectos generados por la evaluación de los términos, ya que los valores resultantes de la evaluación se comparan por la relación $\mathcal{U}$ que relaciona todos los valores con todos los valores.

Ejemplo 9.2.1. *Sea* $M \triangleq \checkmark^5\mathbf{ret}(I)$ *un término de costo* 6 *que retorna la identidad y* $N \triangleq (\checkmark^7\mathbf{ret}(I)) \oplus \mathbf{ret}(I)$ *un término alcanza la identidad con un costo* 9 *o* 2. *Definimos la identidad como* $I \triangleq \boldsymbol{\lambda}\, x\, .\, \mathbf{ret}(x)$. *La evaluación del término* M *tiene costo* 6 *ya que consume* 5 *ticks y luego retorna un valor, lo cual adiciona una unidad de costo. Mientras que la evaluación del término* N *tiene dos valores posibles donde a ambas posibilidades se les suma uno por consumir el operador* $\oplus$*, y siguiendo la misma cuenta que en el término* M *tenemos que una rama consume* 8 *unidades mientras que la otra solo una. Dado que* M *y* N *son términos cerrados, nos basta con encontrar una simulación de mejoras que evidencie la mejora de forma local. Afortunadamente con la relación de identidad nos basta. Intuitivamente, el término* M *evalúa siempre a la función identidad con un costo de* 6 *unidades, mientras que el término* N *evalúa a la función identidad con* 9 *o* 2 *unidades de costo. Dado que es posible obtener el mismo valor utilizando menos de* 6 *unidades de costo evaluando el término* N*, el relacionador de Hoare nos dice que* M *es mejorado por* N.

El relacionador de Hoare considera que un término M es mejorado por un término N si el mejor caso de N tiene menor costo que el mejor caso de M.

Relación de costos de Smyth

El relacionador de Smyth, comparado con el relacionador de Hoare, mapea relaciones pero en el sentido opuesto. Sea $R \subseteq X \times Y$, entonces $A \subseteq X, B \subseteq Y$ están R-relacionados siempre que todo elemento de B esté relacionado con un elemento de A.

Definición 9.2.2 (Relacionador de Smyth (Dal Lago, Gavazzo y Paul Levy 2017))**.** *Sean* X, Y *dos conjuntos, y* $R \subseteq X \times Y$ *una relación. Definimos el relacionador* $\Gamma^S R \subseteq \mathcal{P}(X) \times \mathcal{P}(Y)$ *de la siguiente forma:*

$$S\ \Gamma^S R\ T \iff \forall y \in T, \exists x \in S, x\ R\ y$$

Una relación de λ-términos cerrados $(R_\Lambda, R_\mathcal{V})$ es una simulación de mejoras de Smyth si y solo si respeta valores y para todos términos cerrados M, N, tenemos que:

$$M\ R_\Lambda\ N \implies [\![M]\!]_{\mathbb{N}_\infty}\ \Gamma^S_{\mathbb{N}_\infty} R_\mathcal{V}\ [\![N]\!]_{\mathbb{N}_\infty}$$

Al igual que antes, desenvolvemos la definición del relacionador, y obtenemos que:

$$M\ R_\Lambda\ N \implies \forall (W, n_c) \in [\![N]\!]_{\mathbb{N}_\infty}, \exists (V, m_c) \in [\![M]\!]_{\mathbb{N}_\infty}, W\ R_\mathcal{V}\ V \wedge m_c \geq_\infty n_c$$

Para obtener la definición de mejoras de Smyth lo que hacemos es instanciar la definición de mejoras al relacionador de Smyth, de la misma manera que hicimos con el de Hoare. Dados dos términos cuales quiera M, N, $M \succeq_{\Gamma^S} N$ si y solo si:

$$\forall \mathbb{C} \in Ctx, \mathbb{C}[M], \mathbb{C}[N] \in \Lambda_0, [\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty}\ \Gamma^S_{\mathbb{N}_\infty} \mathcal{U}\ [\![\mathbb{C}[N]]\!]_{\mathbb{N}_\infty}$$

En palabras, para cada posible valor alcanzable por la evaluación de $\mathbb{C}[N]$, hay un valor alcanzable por la evaluación de $\mathbb{C}[M]$ que requiere más o igual costo para ser computado.

Se puede ver la dualidad entre las definiciones de mejoras de Smyth y Hoare. Dados dos términos M, N, comparando las nociones de mejoras $M \succeq_{\Gamma^S} N$ y $M \succeq_{\Gamma^H} N$. Por un lado, la mejora de Hoare solo pide que exista un valor alcanzable en la evaluación de N que tenga un menor costo. Mientras que por el otro, la mejora de Smyth se fija que exista un valor alcanzable en la evaluación de M que tenga un costo mayor. En otras palabras, la mejora de Hoare busca mejorar el mejor caso mientras que con la mejora de Smyth se busca mejorar el peor caso.

Ejemplo 9.2.2. *Retomamos el mismo ejemplo con los términos* $M \triangleq \checkmark^5 \mathbf{ret}(I)$ *y* $N \triangleq (\checkmark^7 \mathbf{ret}(I)) \oplus \mathbf{ret}(I)$*, con* I *definiendo la identidad de la misma manera que en el ejemplo anterior. El término* M *no es (Smyth) mejorado por el término* N *ya que* M *evalúa a la función identidad en* 6 *unidades de costos, mientras que hay una posible ejecución en* N *que puede requerir* 9 *unidades. Sin embargo, podemos probar que* $N \succeq_{\Gamma^S} M$ *siguiendo el mismo procedimiento que en el ejemplo anterior.*

Intuitivamente, un término M es (Smyth) mejorado por un término N si y solo si en todo contexto $\mathbb{C}$ el peor caso en la evaluación de $\mathbb{C}[N]$ tiene que ser mejor (o igual) que el peor caso en la evaluación de $\mathbb{C}[M]$

Relación de costos de Plotkin

La relación de costos de Plotkin es una conjunción entre las dos relaciones antes vistas. En particular, se obtiene una relación donde un término M es mejorado por un término N siempre que para todo contexto $\mathbb{C}$:

- para todo valor alcanzable por la evaluación de $\mathbb{C}[M]$ hay un valor en la evaluación de $\mathbb{C}[N]$ que requiere menos (o igual) costo,
- para todo posible valor en la evaluación de $\mathbb{C}[N]$ hay un valor en la evaluación de $\mathbb{C}[M]$ que requiere más (o igual) costo.

Definición 9.2.3 (Relacionador de Plotkin (Dal Lago, Gavazzo y Paul Levy 2017))**.** *Sean* X, Y *dos conjuntos y* $R \subseteq X \times Y$ *una relación. Definimos el relacionador de Plotkin,* $\Gamma^P R \subseteq \mathcal{P}(X) \times \mathcal{P}(Y)$ *de la siguiente forma:*

$$A \, \Gamma^P R \, B \iff A \, \Gamma^H R \, B \wedge A \, \Gamma^S R \, B$$

Una relación entre λ*-términos cerrados* $(R_\Lambda, R_\mathcal{V})$ *es una simulación de mejoras de Plotkin si y solo si respeta valores y además:*

$$M \; R_\Lambda \; N \implies \begin{array}{l} (\forall V_m \in [\![M]\!]_{\mathbb{N}_\infty}, \exists V_n \in [\![N]\!]_{\mathbb{N}_\infty}, V_m \; R_{\mathcal{V}\mathbb{N}_\infty} \; V_n) \\ \wedge \\ (\forall V_n \in [\![N]\!]_{\mathbb{N}_\infty}, \exists V_m \in [\![M]\!]_{\mathbb{N}_\infty}, V_m \; R_{\mathcal{V}\mathbb{N}_\infty} \; V_n) \end{array}$$

Mientras que la noción de mejoras de Plotkin, para dos términos cuales quiera M, N, $M \succeq_{\Gamma^P} N$ si y solo si, para todo contexto $\mathbb{C}$ que cierre a M, N:

$$\begin{array}{l} (\forall (V_m, c_m) \in [\![M]\!]_{\mathbb{N}_\infty}, \exists (V_n, v_c) \in [\![N]\!]_{\mathbb{N}_\infty}, c_m \geq_\infty c_n) \\ \wedge \\ (\forall (V_n, c_n) \in [\![N]\!]_{\mathbb{N}_\infty}, \exists (V_m, v_c) \in [\![M]\!]_{\mathbb{N}_\infty}, c_m \geq_\infty c_n) \end{array}$$

La relación de costos de Plotkin es la conjunción de la de Hoare y Smyth. En otras palabras, un término N es (Plotkin) mejorado por un término M si y solo si en cualquier contexto $\mathbb{C}$, el mejor caso de la evaluación de $\mathbb{C}[N]$ es mejor que el mejor caso de la evaluación de $\mathbb{C}[M]$, mientras que el peor caso posible de la evaluación de $\mathbb{C}[M]$ es peor que el peor caso de la evaluación de $\mathbb{C}[N]$.

El ejemplo del sistema no-determinista demuestra un caso interesante del cual podemos extraer diferentes nociones de mejoras utilizando diferentes relacionadores. En este caso se derivaron tres teorías de mejoras con diferentes relacionadores para un mismo efecto algebraico, el de no determinismo, obteniendo diferentes nociones de mejoras. Derivar diferentes nociones dependiendo el relacionar utilizado nos muestra que la noción de mejora depende de la observaciones realizadas sobre el sistema.

9.3. Probabilístico

Los lenguajes funcionales probabilísticos son un campo activo de investigación donde se pueden encontrar diferentes maneras de interpretar el efecto generados por términos probabilísticos (Ramsey y Pfeffer 2002). En esta sección, derivamos la noción de mejora para la mónada de sub-distribuciones utilizada para interpretar el operador probabilístico introducido en la Sección 6.1. Dependiendo de cómo se interpreten el efecto de operadores probabilísticos es posible que se puedan obtener diferentes nociones de mejoras. En lo que queda de la sección derivamos una teoría de mejoras para un lenguaje probabilístico.

A modo de recordatorio, la evaluación de términos probabilísticos la llevamos a cabo dentro de la mónada de sub-distribuciones. La mónada de sub-distribuciones de un conjunto X es el conjunto de todas las sub-distribuciones con conjunto soporte numerable en X. La evaluación de un término es interpretada como la sub-distribución generada a partir del término donde un sub-conjunto numerable de términos tiene una probabilidad mayor a 0 de ser el resultado de la evaluación, cuya suma es menor o igual a 1. De esta manera interpretamos que la suma de las probabilidades de los valores del conjunto soporte es la probabilidad de la evaluación de converger a un valor.

Agregamos costos al lenguaje aplicando la mónada de costo interno (Definición 8.1.2) a la mónada de sub-distribuciones, y como resultado, obtenemos una nueva mónada capaz de que interpretar los términos probabilísticos pero con un sentido semántico ligeramente diferente. Dado un término cerrado M, la evaluación de M es interpretada como una sub-distribución $\mu : \mathcal{V}_0 \times \mathbb{N}_\infty \to [0, 1]$ indicando la probabilidad de que un valor v sea el resultado de la evaluación con un costo de n como $\mu(v, n)$.

Una relación entre λ-términos cerrados $(R_\Lambda, R_\mathcal{V})$ es una simulación de mejoras para el sistema probabilístico si y solo si respeta valores y además:

$$M \; R_\Lambda \; N \implies \forall U \subseteq (\mathcal{V}_0 \times \mathbb{N}_\infty), (\llbracket M \rrbracket_{\mathbb{N}_\infty}(U)) \leq (\llbracket N \rrbracket_{\mathbb{N}_\infty}(R_{\mathcal{V}\mathbb{N}_\infty}(U)))$$

Definimos la aplicación de una sub-distribución a un conjunto como la suma de las probabilidades de su conjunto soporte.

Mientras que la relación de mejoras en el sistema de sub-distribuciones es el resultado de instanciar la definición de mejoras al sistema de sub-distribuciones, es decir, que dado un término M es mejorado por un término N si y solo si

para cualquier contexto $\mathbb{C}$ que los cierre, y conjunto $U \subseteq (\mathcal{V} \times \mathbb{N})$:

$$[\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty}(U) \leq [\![\mathbb{C}[N]]\!]_{\mathbb{N}_\infty}(\mathcal{U}_{\mathbb{N}_\infty}(U))$$

Donde $\mathcal{U}_{\mathbb{N}_\infty}$ relaciona todo valor equipado con un costo con cualquier otro valor que tenga un costo menor. En símbolos podemos describirlo como:

$$\mathcal{U}_{\mathbb{N}_\infty} = \{((V, v_c), (W, w_c)) \mid V, W \in \mathcal{V}, v_c \geq_\infty w_c\}$$

Sea $U \subseteq (\mathcal{V}_0 \times \mathbb{N}_\infty)$ un conjunto de valores equipados con sus costos. Cuando aplicamos una sub-distribución de probabilidades μ obtenida de la evaluación de un término M a $\mathcal{U}_{\mathbb{N}_\infty}$, $\mu(\mathcal{U}_{\mathbb{N}_\infty}(U))$, obtenemos la probabilidad de que la evaluación de M converja a un valor con un costo menor o igual que al costo supremo en U. Por ejemplo, si quisiéramos saber cual es la probabilidad de que la evaluación de un término converja a un valor con un costo menor a 42, bastaría con tomar a $U = \{(\mathbf{ret}(I), 42)\}$ con I la función identidad.

Un término M es mejorado en el sistema probabilístico por un término N si y solo si, para todo contexto $\mathbb{C}$ que los cierre, si para todo conjunto de valores V, la probabilidad que la evaluación de $\mathbb{C}[M]$, $[\![\mathbb{C}[M]]\!]_{\mathbb{N}_\infty}$, alcance un valor en V con costo menor a n es menor o igual que la probabilidad de que la evaluación de $\mathbb{C}[N]$, $[\![\mathbb{C}[N]]\!]_{\mathbb{N}_\infty}$, converja con costo menor o igual a n. Notar que al igual que en los casos anteriores los valores alcanzados son ignorados por el uso de la relación $\mathcal{U}$, ya que lo que observamos son los efectos, probabilidad de convergencia, y el costo utilizado por la evaluación.

9.4. Análisis de Costos Alternativos

El análisis de costos presentado en la Sección 8.1 introduce una manera de interpretar costos en lenguajes con efectos: equipar los resultados de computaciones con su correspondiente costo. Sin embargo, también podríamos estar interesados en tener en cuenta el costo de los efectos de otra manera, especialmente cuando hay resultados de computaciones que pueden no tomar forma de valores, y por lo tanto, no llevar consigo ninguna información de costo.

Un ejemplo claro es el caso del sistema de excepciones donde las excepciones son interpretadas como valores distinguidos que no llevan ninguna información de costos (son simplemente etiquetas). Por lo tanto, la noción de mejoras derivada en la Sección 8.1 no compara los costos de las computaciones que terminan en una excepción. Esto es causado por el uso de la mónada de costo interno (Definición 8.1.2), donde para una mónada T, la mónada de costo interno equipa a valores con costos dentro de T, $T_{\mathbb{N}_\infty} \doteq T \circ (\_ \times \mathbb{N}_\infty)$. Más aún, derivamos el análisis de costos guiados por un relacionador mónadico para T donde se comparan valores y costos; sin embargo, si la mónada T interpreta algunos efectos como elementos sin información, como es el caso de la excepciones, entonces va a haber elementos en $T_{\mathbb{N}_\infty}$ sin costo asignado.

Ejemplo 9.4.1. *Sea E un conjunto no vacío de excepciones y $e \in E$. Podemos probar que* $\mathsf{raise}_e \succeq_{\Gamma^E} \checkmark\mathsf{raise}_e$ *dado que la relación entre λ-términos* $(\mathbf{1}_\Lambda \cup \{(\mathsf{raise}_e, \checkmark\mathsf{raise}_e)\}, \mathbf{1}_\mathcal{V})$ *es una simulación de mejoras.*

El problema se origina por equipar los valores con un costo. Por lo que planteamos que en vez de llevar una noción de costo local a cada valor, llevar

un registro del costo total requerido para evaluar un término. En otras palabras, cambiar el foco a llevar un contador global de los ticks que se produjeron durante la evaluación para eventualmente compararlo al final de la computación. A diferencia del enfoque anterior, para obtener una noción de mejoras, necesitamos proveer al sistema con una mónada equipada con un relacionador mónadico que formen un sistema compatible. Comenzamos por definir un funtor que simplemente agregue un número natural al sistema.

Definición 9.4.1 (Funtor de Costo Externo)**.** *Sea T un endo-funtor. Definimos el funtor de costo externo como:*

$$O_T(X) \doteq (T(X) \times \mathbb{N}_\infty)$$

En el caso que T sea un funtor, el funtor de costo externo también es un funtor. Pero lamentablemente, no es un transformador de mónadas: dada una mónada T, O_T no es necesariamente una mónada. De todas formas, para el caso de la mónada de excepciones, sí obtenemos una mónada. Esto es equivalente a utilizar el transformador mónadico de excepciones en la mónada de costos, no al revés como hicimos en el Capítulo 8.

Definición 9.4.2 (Transformador mónadico de Excepciones)**.** *Sea E un conjunto de símbolos sin interpretar representando excepciones y M una mónada. La siguiente construcción es una mónada capaz de interpretar excepciones y posiblemente otros efectos dependiendo de la definición de M:*

$$\mathit{ExceptT}_M(X) \doteq M(E + X)_\perp$$

Donde para cada $e \in E$, definimos el operador raise_e *interpretando de la siguiente manera:*

$$\mathsf{raise}_e \doteq \eta(\iota_l(e))$$

Finalmente, la función **lift** *a continuación promueve computaciones mónadicas de M a computaciones en $\mathit{ExceptT}_M$:*

$$\mathbf{lift} \doteq M\, \iota_r$$

Podemos entonces utilizar la mónada de costos, $\mathbb{D}(A) \doteq (A \times \mathbb{N}_\infty)$ (Dal Lago y Gavazzo 2019), para incorporar costos en el sistemas de excepciones, $\mathtt{ExceptT}_{\mathbb{D}}$, de forma tal que podemos interpretar excepciones directamente con raise y el operador de tick lo interpretamos simplemente como $\mathbf{lift}\, add_1$. Si desenvolvemos la definición del transformador mónadico de excepciones vemos que la definición es equivalente a la del funtor de costo externo. Para todo conjunto E de excepciones:

$$\mathtt{ExceptT}_{\mathbb{D}}(X) \doteq (X + E)_\perp \times \mathbb{N}_\infty$$

Por lo tanto, un relacionador válido para dicho sistema es el resultado de la composición del relacionador de costos y el relacionador de excepciones:

$$\Gamma_{EC} \doteq \Gamma_{\mathbb{N}_\infty} \circ \Gamma_E$$

Finalmente, el par $(\mathtt{ExceptT}_{\mathbb{D}}, \Gamma_{EC})$ forma un sistema compatible de excepciones, y más aún, ya tenemos todos los ingredientes para implementar una función de evaluación con costos (Sección 8.1). Esta nueva definición que obtenemos de mejoras es más granular dado que puede diferenciar computaciones que terminan en la misma excepción, pero que necesitan diferentes costos.

Ejemplo 9.4.2 (Ejemplo 9.4.1 revisado)**.** *Sea E un conjunto no vacío de excepciones y $e \in E$. Podemos ver que* $\mathsf{raise}_e \not\succeq_{\Gamma_{EC}} \checkmark\mathsf{raise}_e$ *simplemente ejecutándolos. Asumamos que* $\mathsf{raise}_e \succeq_{\Gamma_{EC}} \checkmark\mathsf{raise}_e$*, podemos evaluarlos dentro del contexto trivial:*

$$[\![\mathsf{raise}_e]\!]_{EC} \equiv \mathbf{lift}\, add_1\, (\iota_l(e), 0) \equiv (\iota_l(e), 1)$$

mientras que

$$[\![\checkmark\mathsf{raise}_e]\!]_{EC} \equiv \mathbf{lift}\, add_1\, [\![\mathsf{raise}_E]\!]_{EC} \equiv \mathbf{lift}\, add_1\, (\iota_l(e), 1) \equiv (\iota_l(e), 2)$$

Finalmente, no existe una relación $R \subseteq X \times Y$ tal que al mapearla con el relacionador Γ_{EC} se cumpla que $(\iota_l(e), 1)\ \Gamma_{EC}(E)\ (\iota_l(e), 2)$, por definición de Γ_{EC} y el relacionador de costos.

Parecería ser que podríamos obtener una *mónada de costo externo* simplemente apilando un transformador de mónada, con los efectos esperados, sobre la mónada de costos. Sin embargo, no es tan claro como obtener un sistema compatible a partir del transformador de mónadas. El transformador mónadico de excepciones está definido simplemente como la composición de dos funtores, $\mathtt{ExceptT}_M \doteq M \circ E$, y por ende, podemos obtener un sistema compatible por la composición de funtores. Sin embargo, la mayoría de los transformadores no encajan en este patrón.

Una característica esencial de los transformadores de mónadas es su habilidad de mapear computaciones (M. Jaskelioff 2009; M. Jaskelioff y Moggi 2010b), y por lo tanto, deberíamos construir el sistema de costos alrededor de la operación de mapeo de computaciones **lift**. En otras palabras, deberíamos requerir que la operación sea un operador continuo. Podríamos, entonces, ir un paso más y construir nuestro sistema comenzando con los dos efectos básicos: divergencia y costos, para luego agregar nuevos efectos simplemente apilando transformadores de mónadas. El problema se encuentra cuando tratamos de definir un relacionador para dicho sistema. Por lo que deberíamos entonces producir una nueva (en principio) noción de *transformador de relacionadores* que nos permita comparar elementos con efectos, pero no esta claro como forzarlos a tener en cuenta la restricción sobre los costos impuesta por el relacionador de costos. Un candidato podría ser la siguiente definición.

Definición 9.4.3 (Transformador de relacionadores mónadicos)**.** *Sea M una mónada y $\hat{T}$ un transformador de mónadas. Definimos un transformador de relacionadores mónadicos Θ como un operador que toma un relacionador mónadico Γ para M, y retorna un relacionador mónadico para $\hat{T}M$, de forma tal que el operador* **lift** *mapea valores relacionados en valores con efectos relacionados. Formalmente, dados dos conjunto X, Y y una relación $R \subseteq X \times Y$, tenemos que:*

$$\forall a \in M(X), b \in M(Y), a\ \Gamma R\ b \implies \mathbf{lift}(a)\ \Theta\Gamma R\ \mathbf{lift}(b)$$

Lamentablemente, esta definición si bien es prometedora, deberíamos ser capaces de adicionar una propiedad de forma tal que nos permita comparar costos. En símbolos, que si tenemos que $p\ \Theta\Gamma\ q$ nos garantice que el costo en q es menor o igual que el costo en p. Notar que al nivel semántico estamos cubiertos: si podemos aproximar términos dentro de la mónada M, podemos aproximar los mismo términos en la mónada $\hat{T}M$ dado que el operador **lift** es

continuo. El problema está en que la noción de costos no es parte de $\hat{T}$ y por ende el transformador puede ignorarla totalmente. Por ejemplo, el relacionador $\Theta\Gamma$, puede olvidar estructura, y así, relacionar términos ignorando su costo.

Finalmente, otro problema con este enfoque, es que algunas mónadas no tienen un transformador mónadico asociado. Un ejemplo concreto es el de la mónada de sub-probabilidades. En cuyo caso se deberían estudiar como introducir nuevos efectos al sistema.

El caso de estudio de la excepciones es interesante ya que nos presenta con una limitación y la necesidad de introducir un costo global de la evaluación de términos. Si bien pudimos introducir una noción de costo externo en el sistema de excepciones, lamentablemente no queda claro como hacerlo para el caso general y por lo tanto debemos analizar el caso de cada efecto por separado.

Capítulo 10

Optimizaciones como Mejoras con Efectos

En este capítulo, revisamos dos ejemplos de optimizaciones como mejoras en lenguajes con efectos algebraicos. Un ejemplo general donde hipotetizamos sobre como se observan los efectos en el lenguaje y mostramos que la eliminación de código muerto es una optimización en la presencia de efectos. Otro ejemplo donde mostramos que la eliminación de sub-expresiones comunes es una optimización bajo el efecto de no-terminación.

10.1. Eliminación de Código Inobservable

El objetivo de este ejemplo es presentar al lector una optimización general en presencia de efectos y utilizamos este resultado para mostrar una simple optimización dentro del área de computaciones probabilísticas. Hasta el momento que se realizaron los estudios (M. A. Ceresa y M. J. Jaskelioff 2022), y hasta donde el autor conoce, representa la primer mejora probada formalmente dentro del área de computaciones probabilísticas, por lo que, este ejemplo muestra progreso sobre el estudio de lenguajes probabilísticos.

Definimos la *eliminación de código inobservable* como una generalización de eliminación de código muerto en la presencia de efectos algebraicos. La eliminación de código muerto buscar eliminar código que no modifica el resultado final de la computación. En presencia de efectos, podemos estar eliminando código que genera efectos observables, aunque no modifique el resultado final de la computación. Un ejemplo sencillo de código que no influye en la computación de un valor, pero que produce un efecto, es un fragmento de código que simplemente muestra un mensaje por pantalla. En caso de eliminar dicho fragmento de código, estaría eliminando el efecto de mostrar el mensaje por pantalla, y por lo tanto, estaría cambiando el comportamiento observable del programa aunque no su resultado.

A lo largo de la sección asumimos que tenemos una signatura Σ y un Σ-sistema (T, Γ). Dado el sistema (T, Γ) tenemos una relación de similaridad ($\sqsubseteq$) (Dal Lago, Gavazzo y Paul Levy 2017) y una relación de mejoras derivada ($\succeq$) (Sección 8.1).

Teorema 10.1.1 (Eliminación de Código Inobservable)**.** *Sean M, N dos términos cerrados tales que* **let** $x = M$ **in** $N \sqsubseteq N$*, entonces* **let** $x = M$ **in** $N \succeq N$*.*

Demostración. Sean $M, N \in \Lambda_0$, tales que **let** $x = M$ **in** $N \sqsubseteq N$. La prueba se basa en dos hechos: primero, N es cerrado, entonces, la variable x no aparece libre en N; y segundo, el hecho que N aproxime con efectos a **let** $x = M$ **in** N significa que la evaluación de M no agrega una diferencia observable en los efectos generados por la evaluación de N.

Desde el punto de vista de aproximación de términos tenemos que, por ser $\sqsubseteq$ una relación preadecuada:

$$[\![\textbf{let } x = M \textbf{ in } N]\!]\ \Gamma\mathcal{U}\ [\![N]\!]$$

Más aún, dado que la evaluación es determinística y que N es un término cerrado:

$$[\![\textbf{let } x = M \textbf{ in } N]\!]\ \Gamma\mathbf{1}_{\mathcal{V}_0}\ [\![N]\!]$$

En otras palabras, el relacionador mónadico Γ no puede observar ninguna diferencia entre la evaluación de **let** $x = M$ **in** N y N, son equivalentes modulo Γ.

Mientras que al momento de agregar costos y utilizar la evaluación instrumentada, además de evaluar el término, se acumulan los ticks generados por la evaluación del término M, antes de continuar con la evaluación del término N. En el caso que la evaluación de M diverja, tenemos que independientemente de que término sea N, es una mejora ya que Γ es un relacionador inductivo. Dado que la evaluación es equivalente en términos semánticos por hipótesis, podemos restar los ticks generados por la computación de un valor que no se utiliza, y así probar que la eliminación de código inobservable es una mejora.

Formalmente, dado que los términos M y N son cerrados, podemos construir una λ-relación que relaciona términos idénticos y además **let** $x = M$ **in** N con N, y la relación identidad para valores, en símbolos:

$$(\Lambda_0 \cup \{(\textbf{let } x = M \textbf{ in } N, N)\}, \mathcal{V}_0)$$

Dicha relación es una simulación aplicativa de mejoras. Además de la identidad de términos (que es una simulación aplicativa de mejoras), tenemos un solo caso adicional:

$$\begin{aligned}
&[\![\textbf{let } x = M \textbf{ in } N]\!]_{\mathbb{N}_\infty} \\
\equiv&\langle \text{Evaluación} \rangle \\
&\checkmark [\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} \lambda_ \mapsto [\![N]\!]_{\mathbb{N}_\infty} \\
\equiv&\langle \text{Mónada de Costos} \rangle \\
&\checkmark [\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!= \lambda(_, vc) \mapsto \checkmark^{vc} [\![N]\!]_{\mathbb{N}_\infty} \\
\Gamma_{\mathbb{N}_\infty}\mathbf{1}_{\mathcal{V}_0}&\langle \text{Aritmética} \rangle \\
&\checkmark [\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!= \lambda(_, _) \mapsto [\![N]\!]_{\mathbb{N}_\infty} \\
\Gamma_{\mathbb{N}_\infty}\mathbf{1}_{\mathcal{V}_0}&\langle \text{Hipótesis código inobservable } \rangle \\
&\checkmark [\![N]\!]_{\mathbb{N}_\infty}
\end{aligned}$$

□

Dependiendo de los efectos y las observaciones que se hagan sobre los mismos, algunos sistemas aceptan como una transformación válida la eliminación de

código muerto, es decir, aquellos sistemas que para todos dos términos cerrados $M, N \in \Lambda_0, \mathbf{let}\ x = M\ \mathbf{in}\ N \sqsubseteq N$. Para aquellos sistemas, por el lema que acabamos de mostrar, podemos probar que es efectivamente una mejora sujeta a la definición derivada por la noción de equivalencia y el relacionador de costos internos.

A modo de ejemplo concreto de eliminación de código inobservable, mostramos que el lenguaje probabilístico equipado con el sistema de sub-distribuciones para interpretar los efectos acepta dicha optimización. Utilizamos la evaluación instrumentada (Definición 8.1.6) donde se suma un tick por cada constructor del lenguaje y el operador binario durante la evaluación de un término.

Lema 10.1.1. *Dados dos términos probabilísticos cerrados M y N, entonces para toda variable x,* $\mathbf{let}\ x = M\ \mathbf{in}\ N \sqsubseteq N$.

Demostración. Sean M y N dos términos cerrados y x una variable. La prueba consiste en mostrar que el relacionador $\Gamma^{\mathcal{D}}$ relaciona las sub-distribuciones generadas por la evaluación de los términos $\mathbf{let}\ x = M\ \mathbf{in}\ N$ y N. En símbolos podemos evaluar ambos términos para obtener las sub-distribuciones que se generan y ver que son equivalentes modulo $\Gamma^{\mathcal{D}}$:

$$\begin{array}{ll}
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!] & \equiv \\
[\![M]\!] >\!\!>\!\!= \lambda V \mapsto [\![N[x := V]]\!] & \equiv \\
[\![M]\!] >\!\!>\!\!= \lambda V \mapsto [\![N]\!] & \equiv \\
\lambda U \mapsto \Sigma_{W \in \mathcal{V}_0} [\![M]\!](W) \times (\lambda V \mapsto [\![N]\!])(W)(U) & \equiv \\
\lambda U \mapsto \Sigma_{W \in \mathcal{V}_0} [\![M]\!](W) \times [\![N]\!](U) &
\end{array}$$

Por definición de la mónada de sub-distribuciones, la evaluación de un término en todos los valores posibles es menor o igual que 1, y por lo tanto:

$$\forall V \in \mathcal{V}_0, \Sigma_{W \in \mathcal{V}_0} [\![M]\!](W) \times [\![N]\!](V) \leq 1 \times [\![N]\!](V) \leq [\![N]\!](V)$$

Aplicando la definición de $\Gamma^{\mathcal{D}}$:

$$[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]\ \Gamma^{\mathcal{D}} \mathbf{1}_{\mathcal{V}_0}\ [\![N]\!]$$

Y por lo tanto podemos concluir que $\mathbf{let}\ x = M\ \mathbf{in}\ N \sqsubseteq N$. □

La transformación de código inobservable es una mejora para el lenguaje probabilístico, ya que aplica el teorema de eliminación de código inobservable. En palabras, lo que estamos mostrando es que eliminar código que no contribuye a la construcción del resultado, pero que puede introducir mayores posibilidades de fallar (o divergir), es una optimización. Es decir, que eliminar código que mejora la probabilidad de convergencia de la evaluación y que *no* modifica el resultado es una optimización. Podemos replicar un lema similar para el lenguaje no-determinista, pero no podríamos hacerlo con otros efectos como mensajes de salida, donde eliminar operaciones que imprimen mensajes llevarían a diferentes observaciones sobre el sistema.

10.2. Eliminación de Sub-expresiones Comunes

La eliminación de sub-expresiones comunes, CSE de su siglas en inglés [1], es una transformación que busca eliminar múltiples ocurrencias de una sub-expresión en un término. Consiste en identificar una sub-expresión común M, y utilizar una nueva variables fresca x, de forma tal que podamos guardar en x el resultado de la evaluación de M y así consultarlo siempre que sea necesario *sin tener que re-evaluar la expresión* M. La transformación CSE es considerada una optimización ya que evita tener que evaluar múltiples veces una misma expresión, y por lo tanto, el costo total de la evaluación se reduce. Lamentablemente, no siempre es el caso cuando hay efectos computacionales. Por ejemplo, en la presencia de entrada y salida, al eliminar la múltiple evaluación de una sub-expresión que genera un mensaje en pantalla, elimina también que dicho mensaje aparezca múltiples veces.

En esta sección, primero exploramos cómo definir la optimización CSE como una mejora trabajando desde la intuición sin tener en cuenta los efectos, y segundo, probamos que CSE es una optimización cuando el único efecto observable es la no-terminación.

Dada una signatura Σ, y un sistema compatible con Σ, (T, Γ). Enunciamos *intuitivamente* CSE de la siguiente forma: para todo contexto $\mathbb{C}$, término cerrado M, y una variable fresca x, el término $\checkmark\mathbb{C}[\checkmark M]$ es mejorado por el término $\mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{C}[\mathbf{ret}(x)]$ (Hackett y Hutton 2019). Los ticks que se agregan son necesarios para pagar por adelantado la transformación: un tick por guardar el valor de M en x, $\mathbf{let}\ x = M\ \mathbf{in}\ \ldots$, y un tick adicional por cada vez que se busca el valor de x. Esta manera de agregar ticks necesarios dependen de la definición de la función de evaluación instrumentada de términos con costos. En particular, en la Definición 8.1.6, la evaluación de términos $\mathbf{ret}(x)$ y $\mathbf{let}\ x = M\ \mathbf{in}\ N$ agregan un tick al costo de evaluación de un término, y por eso, agregamos un tick por cada construcción agregada al término original.

Si bien la transformación de CSE antes planteada codifica la intuición detrás de CSE, es muy general para ser una mejora ya que incluye contextos en los cuales dicha transformación *empeora* el término, incrementando el costo total de la evaluación. Por ejemplo, si el contexto no requiere la evaluación de su agujero, CSE evaluará la expresión prematuramente de forma innecesaria.

Ejemplo 10.2.1. *Sean N, M dos términos cerrados y $\mathbb{C} \doteq N$.*

El contexto $\mathbb{C}$ es un contexto sin agujeros. Si realizamos la transformación que describe CSE intuitivamente tendríamos que:

$$\checkmark\mathbb{C}[\checkmark M] \equiv \checkmark N[\checkmark M] \succeq_\Gamma \mathbf{let}\ x = M\ \mathbf{in}\ N[\mathbf{ret}(x)]$$

Aplicando la definición de llenado de agujeros, tenemos que:

$$\checkmark N \succeq_\Gamma \mathbf{let}\ x = M\ \mathbf{in}\ N$$

Dicha formula sólo se cumple si la evaluación del término N no termina, pero es falsa en cualquier otro caso.

Una forma de resolver el problema, y así refinar la definición de CSE como una mejora, es introducir la noción de que un contexto utiliza su agujero durante

[1] *Common Sub-expression Elimination*

la evaluación. Seguimos la literatura e introducimos el concepto de *contextos de reducción*, o *contextos de evaluación* (Felleisen y Hieb 1992). Siguiendo la función de evaluación podemos definir contextos que usan sus agujeros durante la evaluación, y así definir correctamente CSE como una mejora.

Definición 10.2.1 (Contextos de Reducción)**.** *Sea Σ una signatura. Definimos los contextos de reducción utilizando la siguiente gramática.*

$$\mathbb{R} ::= [-] \mid (\mathbf{let}\ x = \mathbb{R}\ \mathbf{in}\ \mathbb{C}) \mid (\boldsymbol{\lambda}\ x\ .\ \mathbb{R})\,\mathbb{D} \mid \sigma(\mathbb{C}_0, \ldots, \mathbb{C}_{n-1})$$

Donde $\sigma \in \Sigma, n = \alpha(\sigma)$, y al menos uno de los contextos $\mathbb{C}_i$ con $i < n$ es de reducción, y el contexto $\mathbb{D}$ no necesariamente es de reducción.

Los contextos de reducción son contextos que usan o requieren que sus agujeros sean llenados para completar su evaluación, en otras palabras, la evaluación de un contexto de reducción sin llenar sus agujeros quedaría bloqueada. Es fácil ver que el contexto del Ejemplo 10.2.1 no es un contexto de reducción ya que no tiene ningún agujero en él. Los contextos de reducción garantizan que la expresión con la que llenaremos el agujero será evaluada, exactamente lo que necesitamos para definir CSE.

Definición 10.2.2 (Eliminación de Sub-expresiones Comunes)**.** *Sea Σ una signatura, (Γ, T) un Σ-sistema, y $\mathbb{R}$ un contexto de reducción.*

Definimos a CSE tal que para todo término cerrado $M \in \Lambda_0$ y variable fresca x vale que:

$$\checkmark\mathbb{R}[\checkmark M] \succeq_{\Gamma} \mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{R}[\mathbf{ret}(x)]$$

La transformación CSE es una simple optimización de compilador en lenguajes sin efectos, sin embargo, fue probada correcta por primera vez para un cálculo lambda utilizando una estrategia de reducción *call-by-need* utilizando teorías de mejoras (Moran y Sands 1999b), y desde entonces es el principal ejemplo de las teorías desarrolladas (Hackett y Hutton 2019; Schmidt-Schauß y Sabel 2015). Cuando se agregan efectos algebraicos al lenguaje, la transformación CSE puede no ser correcta, ya que puede cambiar el comportamiento observable de los programas. Por ejemplo, cambiar el orden de dos llamadas a una operación para imprimir un mensaje en pantalla, cambia el orden en que estos mensajes aparecen en pantalla.

Teniendo una definición de CSE, presentamos una prueba de CSE cuando el *único efecto observable es la no-terminación*, es decir, para el sistema parcial, siendo éste el caso de estudio del área. Primero mostramos un lema más simple que conlleva la misma idea y luego lo utilizamos para mostrar que CSE es una mejora.

Desde ahora hasta el final de la sección asumiremos que estamos trabajando con el sistema parcial.

Lema 10.2.1 (CSE Directo)**.** *Sea M un término cerrado, $\mathbb{C}$ un contexto tal que $\mathbb{C}[M] \in \Lambda_0$, y x una variable fresca, tenemos que:*

$$\mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{C}[\checkmark M] \succeq_{\Gamma_\perp} \mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{C}[\mathbf{ret}(x)]$$

El lema CSE Directo establece que, ya que vamos a evaluar la expresión M y guardarla en x, podemos utilizar directamente el valor que está en x y

así evitar evaluar M de nuevo. Pero además, establece como se tendrían que reordenar los efectos producidos por la evaluación de los términos. En otras palabras, encapsula además como se tienen que reestructurar los efectos en el caso que quisiéramos evitar tener que evaluar la expresión M, asumiendo que ya hemos computado su valor. Se puede ver que CSE Directo es un caso particular de CSE donde se utiliza el contexto de reducción $\mathbf{let}\ x = [-]\ \mathbf{in}\ \mathbb{C}$. La prueba procede por inducción en la estructura del contexto $\mathbb{C}$ y se puede encontrar en el Apéndice A.

Volviendo a la prueba de CSE, esta consiste en la observación de que la evaluación de contextos de reducción llevan a la evaluación de una expresión *let* (donde CSE directo se puede aplicar) o a su agujero (donde la mejora es evidente). Esta observación es similar a lo que Moran y Sands (1999b) llaman *open uniform computation lemma* donde los autores muestran que (en su sistema) la evaluación de sus contextos lleva a una variable libre, un valor, o un agujero.

El sistema parcial no tiene ninguna operación, $\Sigma = \emptyset$. Esto reduce los posibles contextos de reducción, y en particular, no hay contextos de reducción de la forma $\sigma(\ldots)$. Eliminar el caso particular del contexto de reducción, y reordenar los efectos en la prueba de CSE directo (Apéndice A), son los únicos momentos donde utilizamos que trabajamos con el sistema parcial.

Lema 10.2.2 (CSE para el sistema parcial)**.** *Sea M un término cerrado, $\mathbb{R}$ un contexto de reducción tal que $\mathbb{R}[M] \in \Lambda_0$, y x una variable fresca, entonces:*

$$\checkmark\mathbb{R}[\checkmark M] \succeq_{\Gamma_\perp} \mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{R}[\mathbf{ret}(x)]$$

Demostración. La prueba consiste en construir una simulación aplicativa que exponga la mejora. Sea M un término cerrado, y $\mathbb{R}$ un contexto, tal que $\mathbb{R}[M] \in \Lambda_0$. Definimos la siguiente relación entre λ-términos:

$$\begin{array}{rcl}
\mathbb{V}_\Lambda \doteq & \Lambda_0 \\
\quad \cup & \{(\checkmark\mathbb{R}[\checkmark M], \mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{R}[\mathbf{ret}(x)]) \\
& \mid \mathbb{R} \text{ es un contexto de reducción}, x \text{ es fresca}\} \\
\quad \cup & \{(\mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{C}[\checkmark M], \mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{C}[\mathbf{ret}(x)]) \\
& \mid \mathbb{C} \text{ es un contexto}, x \text{ es fresca}\} \\
\mathbb{V}_\mathcal{V} \doteq & \mathbf{1}_{\mathcal{V}_0}
\end{array}$$

La relación $\mathbb{V}$ describe la transformación CSE, y además, la transformación CSE directo que utilizaremos como lema dentro de la demostración. Utilizaremos los principios de coinducción y análisis por casos para probar que la relación $\mathbb{V}$ es una simulación aplicativa, y nos concentramos en probar los casos relacionados a los de la transformación de CSE. La transformación CSE directo es probada en el apéndice (Apéndice A).

La relación $\mathbb{V}_\mathcal{V}$ es la identidad sobre valores cerrados, y por lo tanto, respeta valores.

Lo que nos queda por probar es que la evaluación de términos cerrados nos lleva a valores relacionados. En símbolos, dado $\mathbb{R}$ un contexto de reducción y x una variable fresca, nos queda probar que:

$$[\![\checkmark\mathbb{R}[\checkmark M]]\!]_{\mathbb{N}_\infty}\ \Gamma_{\perp_{\mathbb{N}_\infty}}\ \mathbb{V}_\mathcal{V}\ [\![\mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{R}[\mathbf{ret}(x)]]\!]_{\mathbb{N}_\infty}$$

Procedemos haciendo análisis por casos en la estructura de $\mathbb{R}$. En el caso de que el contexto de reducción $\mathbb{R}$ es el contexto trivial, $\mathbb{R} = [-]$, el resultado

se obtiene directamente. Además ya que estamos en el sistema parcial, el conjunto de operadores es vacío, por lo que no existen contextos de reducción con operadores algebraicos. Como resultado solo tenemos dos casos: el caso que $\mathbb{R}$ sea un let-binding o una aplicación.

- El caso en que el contexto de reducción sea una expresión let-binding: sea $\mathbb{R} = \mathbf{let}\ y = \mathbb{S}\ \mathbf{in}\ \mathbb{C}$, para algún contexto de reducción $\mathbb{S}$ y contexto (no necesariamente de reducción) $\mathbb{C}$.

$$
\begin{array}{rl}
 & [\![\checkmark(\mathbf{let}\ y = \mathbb{S}\ \mathbf{in}\ \mathbb{C})[\checkmark M]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Llenado de agujeros, } M \text{ es cerrado}\rangle \\
 & [\![\checkmark(\mathbf{let}\ y = \mathbb{S}[\checkmark M]\ \mathbf{in}\ \mathbb{C}[\checkmark M])]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Evaluación}\rangle \\
 & \checkmark^2([\![\mathbb{S}[\checkmark M]]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} V \mapsto [\![\mathbb{C}[\checkmark M][y := V]]\!]_{\mathbb{N}_\infty}) \\
\equiv & \langle \text{Asociatividad de la mónada}\rangle \\
 & \checkmark^2[\![\mathbb{S}[\checkmark M]]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} V \mapsto [\![\mathbb{C}[\checkmark M][y := V]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Evaluación}\rangle \\
 & \checkmark[\![\checkmark\mathbb{S}[\checkmark M]]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} V \mapsto [\![\mathbb{C}[\checkmark M][y := V]]\!]_{\mathbb{N}_\infty} \\
\Gamma_{\perp\mathbb{N}_\infty} & \langle \text{Hipótesis coinductiva, } x \text{ es una variable fresca}\rangle \\
 & \checkmark[\![\mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{S}[\mathbf{ret}(x)]]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} V \mapsto [\![\mathbb{C}[\checkmark M][y := V]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Evaluación}\rangle \\
 & \checkmark(\checkmark[\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto [\![\mathbb{S}[\mathbf{ret}(x)][x := W]]\!]_{\mathbb{N}_\infty}) \\
 & >\!\!>\!\!=_{\mathbb{N}_\infty} \\
 & V \mapsto [\![\mathbb{C}[\checkmark M][y := V]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Asociatividad de las mónadas}\rangle \\
 & \checkmark^2[\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto [\![\mathbb{S}[\mathbf{ret}(x)][x := W]]\!]_{\mathbb{N}_\infty} \\
 & >\!\!>\!\!=_{\mathbb{N}_\infty} \\
 & V \mapsto [\![\mathbb{C}[\checkmark M][y := V]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Varios pasos en la evaluación}\rangle \\
 & [\![\mathbf{let}\ x = M\ \mathbf{in}\ \mathbf{let}\ y = \mathbb{S}[\mathbf{ret}(x)]\ \mathbf{in}\ \mathbb{C}[\checkmark M]]\!]_{\mathbb{N}_\infty} \\
\Gamma_{\perp\mathbb{N}_\infty} & \langle \text{CSE directo, } x \text{ es fresca}\rangle \\
 & [\![\mathbf{let}\ x = M\ \mathbf{in}\ \mathbf{let}\ y = \mathbb{S}[\mathbf{ret}(x)]\ \mathbf{in}\ \mathbb{C}[\mathbf{ret}(x)]]\!]_{\mathbb{N}_\infty}
\end{array}
$$

- El caso en el que el contexto $\mathbb{R}$ sea un reducción aplicación: $\mathbb{R} = (\boldsymbol{\lambda}\ y\ .\ \mathbb{S})\ \mathbb{V}$, para algún contexto de reducción $\mathbb{S}$, contexto valor $\mathbb{V}$ y variable y.

$$
\begin{array}{rl}
 & [\![\checkmark((\boldsymbol{\lambda}\ y\ .\ \mathbb{S})\ \mathbb{V})[\checkmark M]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Llenado de agujeros, } M \text{ es un término cerrado}\rangle \\
 & [\![\checkmark((\boldsymbol{\lambda}\ y\ .\ \mathbb{S}[\checkmark M])\ (\mathbb{V}[\checkmark M]))]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Evaluación}\rangle \\
 & \checkmark^2[\![\mathbb{S}[\checkmark M][y := \mathbb{V}[\checkmark M]]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Llenado de agujeros, } M \text{ es un término cerrado}\rangle \\
 & \checkmark^2[\![(\mathbb{S}[y := \mathbb{V}])[\checkmark M]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Evaluación}\rangle \\
 & \checkmark[\![\checkmark(\mathbb{S}[y := \mathbb{V}])[\checkmark M]]\!]_{\mathbb{N}_\infty} \\
\Gamma_{\perp\mathbb{N}_\infty} & \langle \text{Hipótesis coinductiva, } x \text{ es una variables fresca}\rangle \\
 & \checkmark[\![\mathbf{let}\ x = M\ \mathbf{in}\ (\mathbb{S}[y := \mathbb{V}])[\mathbf{ret}(x)]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Llenado de agujeros, } M \text{ es un término cerrado}\rangle \\
 & \checkmark[\![\mathbf{let}\ x = M\ \mathbf{in}\ ((\mathbb{S}[\mathbf{ret}(x)])[y := (\mathbb{V}[\mathbf{ret}(x)])])]\!]_{\mathbb{N}_\infty}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \langle \text{Evaluación} \rangle \\
& \checkmark[\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto [\![((\mathbb{S}[\mathbf{ret}(x)])[y := (\mathbb{V}[\mathbf{ret}(x)])])[x := W]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Substitución, } x \notin FV(\mathbb{S}, \mathbb{V}), x \neq y \rangle \\
& \checkmark[\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto [\![((\mathbb{S}[\mathbf{ret}(W)])[y := (\mathbb{V}[\mathbf{ret}(W)])])]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Tick es una operación pura mapeada} \rangle \\
& [\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto \checkmark[\![((\mathbb{S}[\mathbf{ret}(W)])[y := (\mathbb{V}[\mathbf{ret}(W)])])]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Evaluación} \rangle \\
& [\![M]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto [\![(\boldsymbol{\lambda}\ y\ .\ \mathbb{S}[\mathbf{ret}(W)])\ (\mathbb{V}[\mathbf{ret}(W)])]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Substitución, } x \text{ es una variable fresca} \rangle \\
& [M]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto [\![((\boldsymbol{\lambda}\ y\ .\ \mathbb{S}[\mathbf{ret}(x)]\ )(\mathbb{V}[\mathbf{ret}(x)]))[x := W]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Evaluación} \rangle \\
& [\![\mathbf{let}\ x = M\ \mathbf{in}\ (\boldsymbol{\lambda}\ y\ .\ \mathbb{S}[\mathbf{ret}(x)])\ (\mathbb{V}[\mathbf{ret}(x)])]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Llenado de agujeros, } x \text{ es una variable fresca} \rangle \\
& [\![\mathbf{let}\ x = M\ \mathbf{in}\ ((\boldsymbol{\lambda}\ y\ .\ \mathbb{S})\ \mathbb{V})[\mathbf{ret}(x)]]\!]_{\mathbb{N}_\infty}
\end{array}
$$

□

De esta manera podemos ver que la teoría de mejoras derivada de la aproximación de programas con el efecto observable de que la evaluación es la no-terminación nos permite probar que la CSE es una optimización.

Introducir efectos en CSE

En la sección anterior mostramos una prueba de CSE donde el único efecto observable es no-terminación, posicionando la teoría de mejoras derivada de la equivalencia al mismo nivel que sus predecesores, pero nos queda una pregunta en el tintero: ¿CSE es una mejora en presencia de otros efectos? En nuestra teoría, hay dos entidades que participan al momento de comparar dos programas con efectos: relacionadores y mónadas. Esto hace que tengamos dos maneras de probar que una transformación sobre un programa es efectivamente una mejora:

- mapear propiedades de la mónada que interpreta los efectos algebraicos generados por los operadores
- ajustar las observaciones realizadas sobre los efectos generados, estableciendo propiedades sobre los relacionadores.

La transformación CSE reemplaza múltiples evaluaciones de un sub-término compartido por una sola evaluación. Al transformar un término siguiendo CSE, los efectos producidos por la evaluación de una sub-expresión compartida pueden ser *eliminados* o *ignorados*. Podemos eliminar efectos si la mónada que interpreta los efectos tiene alguna propiedad (e.g. idempotencia) que asegure la equivalencia entre términos. Mientras que podemos ignorar efectos si la observación realizada por el relacionador simplemente no los distingue, e.g., si las cadenas de caracteres a mostrar en pantalla son ignorados. En otras palabras, los efectos pueden ser ignorados si los relacionadores olvidan estructura.

Todavía al momento de realizar la tesis, no hemos encontrado condiciones necesarias y suficientes para establecer cuando CSE es una mejora en presencia de efectos, pero sabemos que una noción de idempotencia es necesaria. Sea Σ una signatura y (Γ, T) un sistema compatible tal que CSE es una mejora. Entonces,

podemos mostrar que el sistema (Γ, T) es (observacionalmente) idempotente si para todo término cerrado $M \in \Lambda_0$:

$$\textbf{let } x = M \textbf{ in } \checkmark M \succeq \textbf{let } x = M \textbf{ in } \textbf{ret}(x) \succeq M$$

Por lo que podemos concluir que: efectos que no sean *observacionalmente* idempotentes no aceptan CSE como una mejora. Además, pudimos notar que una noción débil de commutatividad es necesaria, pero estos estudios han quedado inconclusos.

La mónada parcial es conmutativa e idempotente, y por lo tanto, podemos reordenar o incluso remover múltiples evaluaciones del mismo término, pero lamentablemente, no hay tantas mónadas que tengan ambas propiedades. Sin embargo, podemos aprender algo de la prueba de CSE utilizada en esta sección, necesitamos dichas propiedades en dos lugares particulares de la prueba: en CSE directo, donde los efectos son reordenados; y en los contextos de reducción de operaciones. El resto de la prueba es genérica en los efectos. Esto indica que puede llegar a ser posible ajustar más las definiciones de manera tal que sea posible obtener una noción aunque sea observacional de cómo reordenar los efectos generados, y además estudiar cómo la interpretaciones de los operadores consumen los efectos ya generados.

En este capítulo vimos dos ejemplos concretos de cómo utilizar la teoría de mejoras para mostrar que ciertas transformaciones de programas son optimizaciones. En el caso de eliminación de sub-expresiones comunes, primero definimos la transformación utilizando contexto de reducción y luego utilizamos una simulación de mejoras para mostrar que CSE es efectivamente una mejora. Mientras que en el caso de eliminación de código inobservable primero establecimos los requerimientos del sistema y lo probamos de forma general. Luego mostramos que el sistema probabilístico cumple con el requerimiento, y por ende, la eliminación de código inobservable es una mejora en dicho sistema.

Capítulo 11

Trabajo Futuro y Conclusiones

Optimizar programas es difícil. Mostrar que al transformar un programa en otro estamos mejorando una métrica de la ejecución incluye mostrar que se mantiene el sentido semántico, y más aún, que se mejora la ejecución del programa. En la tesis vimos que hay varias interpretaciones posibles para los programas, pero no todas permiten introducir de manera natural propiedades intensivas de la ejecución de los mismos. Esta separación entre la interpretación de los programas y su evaluación nos lleva a instrumentar la evaluación de los mismos para así tener la información relevante al consumo de recursos.

Los lenguajes funcionales presentan una semántica más clara que los lenguajes no funcionales. Esto nos permite realizar pruebas de equivalencia de programas, pero ocultan propiedades de la ejecución. Esta es la motivación principal para el desarrollo de la teorías de mejoras, presentando por primera vez una teoría relacional de costos de programas funcionales.

En esta tesis caracterizamos exitosamente teorías de mejoras para lenguajes funcionales con efectos algebraicos. Por un lado, agregamos más expresividad a los lenguajes funcionales introduciendo efectos algebraicos, un subconjunto de efectos computacionales. Mientras que por el otro, siguiendo la literatura, identificamos la definición de mejoras entre programas como un refinamiento de la equivalencia observacional. El refinamiento de la equivalencia observacional se logró fácilmente mediante el uso de relacionadores en la evaluación de lenguajes con efectos algebraicos. Definimos además la relación de simulación mejoras simplificando las pruebas para mejoras locales de programas. Derivamos teorías de mejoras para varios de los efectos algebraicos más utilizados: no-determinismo, excepciones, y operadores probabilísticos. Probamos que dos transformaciones son efectivamente optimizaciones, una general y otra particular, utilizando las nociones de mejoras derivadas durante la tesis. Identificamos una limitación del enfoque en donde se deriva la teoría de mejoras y mostramos cómo evitarlo. Finalmente, exploramos una posible solución general de la limitación.

A continuación presentamos trabajo futuro y luego concluimos la tesis.

11.1. Trabajo Futuro

La tesis plantea el principio de la investigación del estudio de análisis de propiedades intensivas sobre lenguajes funcional con efectos algebraicos abriendo

más preguntas que soluciones. En esta sección simplemente listamos posibles extensiones o trabajos inconclusos.

Observaciones Estudiar el impacto de las observaciones realizadas sobre las evaluaciones y cómo se interpretan los efectos para realizar un análisis más profundo sobre las propiedades intensivas de la evaluación de términos. En otras palabras, la interacción entre los relacionadores y las mónadas que interpretan los efectos, presentan en éste trabajo un rol fundamental para explorar diferentes formas de obtener teorías de mejoras. Por ejemplo, en la Sección 9.4 pudimos obtener una noción más granular de costos modificando cómo se interpretan y observan los efectos.

Una línea de investigación sería entonces explorar las relaciones entre las mónadas y sus relacionadores, de manera tal que podamos garantizar propiedades sobre el análisis intensivo de programas. Uno de los trabajos iniciales sobre la teoría de mejoras proponía utilizar un álgebra de ticks (Moran y Sands 1999b), es decir, propiedades sobre el operador tick del lenguaje. En este caso proponemos ir en el sentido opuesto, ya que sabemos cuáles son (algunas de) las propiedades que esperamos que cumplan los sistemas de mejoras, podemos intentar caracterizar de forma genérica los sistemas que cumplen con ellas. Por ejemplo, de la misma manera que utilizamos la noción de idempotencia de las mónadas para probar que es una propiedad necesaria para poder caracterizar a CSE como una mejora, es posible que haya otras relaciones y propiedades necesarias para probar optimizaciones conocidas.

Álgebra de Ticks Establecer propiedades algebraicas sobre el operador tick es extremadamente útil al momento de mostrar mejoras, y además, para definirlas. Al poner al operador tick como bloque fundamental de la teoría de costos nos permite además introducir propiedades sobre él y los diferentes operadores del lenguaje. De esta manera sería posible obtener una álgebra de tick (Moran y Sands 1999b), y además, se podría obtener teoremas como el *Teorema de Mejora* (Sands 1991), donde nos facilita además probar propiedades sobre funciones recursivas.

Mejoras Explorar el uso de las teorías de mejoras derivadas para encontrar o probar optimizaciones sobre lenguajes funcionales con efectos. Mientras que en la tesis se desarrolla un cuerpo teórico capaz de probar optimizaciones, falla al momento de presentar ejemplos y aplicaciones de la misma. Al ser el área de lenguajes con efectos algebraicos joven, es complicado al momento de buscar optimizaciones o incluso transformaciones sobre términos de dichos lenguajes. Diferente es el caso de optimizaciones de compiladores donde ya los lenguajes suelen tener efectos que inhabilitan ciertas transformaciones, un ejemplo mencionado durante la tesis es el de mostrar mensajes por pantalla.

Conectando con el punto anterior, incluso serviría mostrar que es posible obtener mejoras en sistemas donde los efectos están pero no son utilizados por el término. Por ejemplo, si bien tenemos la posibilidad de mostrar mensajes por pantalla, y en general CSE no es válida, un término que no utilice los operadores que generan ése efecto se podría optimizar beneficiando la evaluación general del programa.

Finalmente, una de las principales motivaciones de la teoría de mejoras clásica es la posibilidad de trabajar con recursión y probar mejoras dentro de programas recursivos. En esta tesis evitamos trabajar teniendo en cuenta la posibilidad de tener recursión dentro del lenguaje. Una posible extensión al lenguaje sería agregar un operador de punto fijo y estudiar si es posible caracterizar optimizaciones dentro de funciones recursivas. Un trabajo reciente (Veltri y Voorneveld 2022), además de mecanizar gran parte de las pruebas, siguen el enfoque mencionado, aunque no buscan determinar mejoras dentro de funciones recursivas.

Costo Externo: Transformador de Mónadas En Sección 9.4 exploramos levemente como podría derivarse un sistema si consideramos tener un costo global de la ejecución, proponiendo un camino que dejamos como trabajo futuro. En este caso, se podría estudiar como introducir de forma externa a la evaluación y a los relacionadores, el concepto de costo. El costo externo se podría ver como el costo absoluto que realmente toma la operación, por ejemplo, si bien hay múltiples posibles caminos en una evaluación paralela, eventualmente un camino es el que se toma y se computa. Con lo que podría llevar a análisis de costos más granulares, y podríamos incluso pensar que esta es la noción de costos más intuitiva. Finalmente, mientras en este trabajo se utilizaron transformadores de mónadas, hay otra técnicas que podrían explorarse al momento de componer efectos (Lüth y Ghani 2002).

Técnicas de Bisimulación La idea central de la tesis se basa en refinar la noción observacional de aproximación de programas para realizar análisis intensivo. El uso de contextos y bisimulaciones es un área ya con resultados que se podrían incorporar dentro de la teoría de mejoras. Un clásico ejemplo es introducir un lema de contexto, similar a como hicimos en la Sección 10.2 al introducir contextos de reducción, donde se busca reducir los universales inherentes a las definiciones observacionales. Otras técnicas, como ser *up-to* contextos (Pous y Sangiorgi 2019), permitirían simplificar las pruebas y expandir las herramientas para mostrar mejoras en los diferentes sistemas.

Comparación de Teorías de Mejoras Nos encontramos con la posibilidad de obtener diferentes teorías de mejoras, y por ende, con la pregunta si es posible encontrar un criterio que nos permita decidir una sobre la otra. Un criterio sencillo sería obtener teorías más granulares, el cual seguimos en el caso del sistema de excepciones, donde se podría ver que hay una estricta contención entre las mejoras del sistema de excepciones con costos externos en el sistema de excepciones con costos internos. Pero no está muy claro como podríamos comparar mejoras que utilicen distintos relacionadores, o incluso, por que no, mónadas.

11.2. Conclusiones

Durante la tesis estudiamos el desarrollo de teorías de mejoras desde un cálculo lambda sin efectos, repasando el trabajo realizado por Sands, pero utilizando técnicas modernas que nos permitieron luego adaptar el mismo entorno de trabajo para introducir efectos. Desde la creación de la teoría de

mejoras en 1990 se exploraron diferentes aristas y aplicaciones de dicha teoría, siendo principalmente utilizada para el estudio de optimizaciones en lenguajes funcionales, mostrando su mayor aplicación en la demostración de que CSE es efectivamente una optimización. Con el pasar de los años, y sin tener un uso claro, la teoría quedó en desuso, volviendo a ser sujeto de estudio en los últimos años.

Desde su creación hasta el día de hoy el quehacer científico de la comunidad ha cambiado, y el área de estudio de lenguajes de programación ha avanzado significativamente, trayendo consigo nuevas técnicas, y por sobre todo, un mayor esfuerzo en el estudio de efectos computacionales, en particular los efectos algebraicos. En esta tesis lo que buscamos fue unir estos dos caminos, comenzar a explorar el uso de teorías de mejoras para realizar análisis intensivo sobre programas en lenguajes que acepten efectos.

La tesis comienza desde la exploración de la teoría de mejoras en lenguajes sin efectos (Capítulo 5), añadiendo efectos algebraicos al lenguaje (Capítulo 6), logrando finalmente producir una teoría de mejoras con efectos (Capítulo 8). La teoría la derivamos con el mismo espíritu original de la teoría de mejoras de Sands, donde se busca hacer un refinamiento sobre la definición de equivalencia observacional de programas, en este caso, con efectos (Sección 8.1). Esto era posible ya que podíamos manipular las observaciones sobre el sistema introduciendo un nuevo relacionador, el relacionador de costos (Definición 8.1.3), y además haciendo uso del transformador mónadico de Writer con el monoide aditivo de los naturales, definimos una mónada capaz que llevar cuenta del costo necesario para computar los valores (Definición 8.1.2). Finalmente, revisitamos los ejemplos de efectos algebraicos desarrollados a lo largo de la tesis para derivar teorías de mejoras para cada uno de ellos.

Para mostrar la utilidad de la teoría introducimos dos ejemplos. Uno de carácter general tratando de describir una optimización general sobre lenguajes con efectos algebraicos, que instanciamos en particular para un lenguaje con operadores probabilísticos. Mientras que el otro, más concreto, nos permite situarnos al mismo nivel en principio que las demás teorías de mejoras desarrolladas desde sus comienzos.

Sin embargo, las definiciones derivadas, y los ejemplos mencionados, sirvieron para introducir nuevos problemas sobre las teorías de mejoras. Primero, y más importante, nos encontramos con una forma de obtener varias teorías de mejoras, dependiendo su definición en las observaciones realizadas sobre la evaluación de términos con operadores que introducen efectos en el lenguaje. Disparando aquí una pregunta, rememorando el trabajo de Howe (1989), donde se busca establecer una conexión entre lo que el lenguaje puede observar sobre los términos, y lo que podemos explorar nosotros con la metateoría. Las definiciones de aproximación observacional dependen directamente de las observaciones que los contextos puedan hacer. Segundo, al tener varias teorías posibles nos encontramos con la opción de elegir, es decir, como podríamos desarrollar un criterio de elección de teorías de mejoras. Tercero, continuando con los criterios, y basándonos en los ejemplos visto, podría tener sentido establecer propiedades algebraicas generales a todas las teorías de mejoras. Por ejemplo, que $M \not\gtrsim \checkmark M$ independientemente cual sea el sistema con el que se trabaja. Pero no está del todo claro, en el caso de las excepciones, si no importa realmente cuando las computaciones levantan excepciones, se puede querer optimizar casos utilizando la teoría derivada sin tener que refinar las observaciones como se hizo en la Sección 9.4.

Para concluir con la tesis, esta presenta el principio de la aplicación de teorías de mejoras en lenguajes con efectos. Afortunadamente no es posible dar por cerrado el tema, la tesis explora diferentes alternativas pero no podemos concluir con una teoría de mejoras. El estudio de optimizaciones entre programas es un campo complejo y más aún al adicionar la posibilidad de tener efectos algebraicos. Esto lo logramos estableciendo de forma clara las observaciones que se realizan sobre el sistema, definiendo un relacionador, y cómo se administra la acumulación del costo, de forma interna o externa.

Bibliografía

Abramsky, Samson (1990). «The Lazy Lambda Calculus». En: *Research Topics in Functional Programming*. USA: Addison-Wesley Longman Publishing Co., Inc., págs. 65-116. ISBN: 0201172364.

Abramsky, Samson y Guy McCusker (1999). «Game Semantics». En: *Computational Logic*. Ed. por Ulrich Berger y Helmut Schwichtenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, págs. 1-55. ISBN: 978-3-642-58622-4.

Allison, L. (1987). *A Practical Introduction to Denotational Semantics*. Cambridge Computer Science Texts. Cambridge University Press. DOI: 10.1017/CBO9781139171892.

Antoy, Sergio (1997). «Optimal non-deterministic functional logic computations». En: *Algebraic and Logic Programming*. Ed. por Michael Hanus, Jan Heering y Karl Meinke. Berlin, Heidelberg: Springer Berlin Heidelberg, págs. 16-30. ISBN: 978-3-540-69555-4.

Backhouse, R.C., P.J. de Bruin, P.F. Hoogendijk et al. (1991). «Polynomial Relators (Extended Abstract)». En: *Proceedings of the Second International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology*. AMAST '91. Berlin, Heidelberg: Springer-Verlag, págs. 303-326. ISBN: 3540197974.

Backhouse, R.C., P.J. de Bruin, G.R. Malcolm et al. (1991). *Relational catamorphisms*. English. Computing science notes. Technische Universiteit Eindhoven.

Backhouse, R.C. y P. Hoogendijk (1993). «Elements of a relational theory of datatypes». En: *Formal Program Development: IFIP TC2/WG 2.1 State-of-the-Art Report*. Ed. por Bernhard Möller, Helmut Partsch y Steve Schuman. Berlin, Heidelberg: Springer Berlin Heidelberg, págs. 7-42. ISBN: 978-3-540-48197-3. DOI: 10.1007/3-540-57499-9_15. URL: https://doi.org/10.1007/3-540-57499-9_15.

Barendregt, Hendrik Pieter (1985). *The lambda calculus - its syntax and semantics*. Vol. 103. Studies in logic and the foundations of mathematics. North-Holland. ISBN: 978-0-444-86748-3.

Benton, Nick (ene. de 2004). «Simple Relational Correctness Proofs for Static Analyses and Program Transformations». En: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, pág. 43. URL: https://www.microsoft.com/en-us/research/publication/simple-relational-correctness-proofs-for-static-analyses-and-program-transformations/.

Bergman, C. (ene. de 2011). *Universal algebra: Fundamentals and selected topics*, págs. 1-299.

Bird, Richard y Oege de Moor (sep. de 1997). *The Algebra of Programming*. Prentice Hall, pág. 295. ISBN: 013507245X.

Carboni, A., G. M. Kelly y R. J. Wood (1991). «A 2-categorical approach to change of base and geometric morphisms I». en. En: *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 32.1, págs. 47-95. URL: http://www.numdam.org/item/CTGDC_1991__32_1_47_0/.

Ceresa, Martín, Felipe Gorostiaga y César Sánchez (2020). «Declarative Stream Runtime Verification (HLola)». En: *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*. Ed. por Bruno C. d. S. Oliveira. Vol. 12470. Lecture Notes in Computer Science. Springer, págs. 25-43. DOI: 10.1007/978-3-030-64437-6_2. URL: https://doi.org/10.1007/978-3-030-64437-6%5C_2.

Ceresa, Martin A. y Mauro J. Jaskelioff (2022). «Effectful improvement theory». En: *Science of Computer Programming* 217, pág. 102792. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2022.102792. URL: https://www.sciencedirect.com/science/article/pii/S0167642322000259.

Cicek, Ezgi et al. (ene. de 2017). «Relational Cost Analysis». En: *Symposium on the Principle of Programming Languages, ACM*.

Cormen, Thomas H. et al. (2009). *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press. ISBN: 0262033844.

D'Angelo, Ben et al. (jun. de 2005). «Lola: Runtime Monitoring of Synchronous Systems». En: 12^{th} *International Symposium on Temporal Representation and Reasoning (TIME'05)*. Burlington, Vermont: IEEE Computer Society Press, págs. 166-174.

Dal Lago, Ugo y Francesco Gavazzo (abr. de 2019). «Effectful Normal Form Bisimulation». En: *ESOP 2019 - European Symposium on Programming*. Prague, Czech Republic. URL: https://hal.inria.fr/hal-02386004.

Dal Lago, Ugo, Francesco Gavazzo y Paul Levy (2017). «Effectful applicative bisimilarity: Monads, relators, and Howe's method». En: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, págs. 1-12. DOI: 10.1109/LICS.2017.8005117.

Felleisen, Matthias (1991). «On the Expressive Power of Programming Languages». En: *Sci. Comput. Program.* 17.1-3, págs. 35-75. DOI: 10.1016/0167-6423(91)90036-W. URL: https://doi.org/10.1016/0167-6423(91)90036-W.

Felleisen, Matthias y Robert Hieb (sep. de 1992). «The Revised Report on the Syntactic Theories of Sequential Control and State». En: *Theor. Comput. Sci.* 103.2, págs. 235-271. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90014-7. URL: https://doi.org/10.1016/0304-3975(92)90014-7.

Floyd, Robert W. (1993). «Assigning Meanings to Programs». En: *Program Verification: Fundamental Issues in Computer Science*. Ed. por Timothy R. Colburn, James H. Fetzer y Terry L. Rankin. Dordrecht: Springer Netherlands, págs. 65-81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4. URL: https://doi.org/10.1007/978-94-011-1793-7_4.

Forrest, Peter (2020). «The Identity of Indiscernibles». En: *The Stanford Encyclopedia of Philosophy*. Ed. por Edward N. Zalta. Winter 2020. Metaphysics Research Lab, Stanford University.

Goguen, J. A. et al. (ene. de 1977). «Initial Algebra Semantics and Continuous Algebras». En: *J. ACM* 24.1, págs. 68-95. ISSN: 0004-5411. DOI: 10.1145/321992.321997. URL: https://doi.org/10.1145/321992.321997.

Grieco, Gustavo, Martín Ceresa y Pablo Buiras (2016). «QuickFuzz: an automatic random fuzzer for common file formats». En: *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. Ed. por Geoffrey Mainland. ACM, págs. 13-20. DOI: 10.1145/2976002.2976017. URL: https://doi.org/10.1145/2976002.2976017.

Grieco, Gustavo, Martín Ceresa, Agustín Mista et al. (2017). «QuickFuzz testing for fun and profit». En: *J. Syst. Softw.* 134, págs. 340-354. DOI: 10.1016/j.jss.2017.09.018. URL: https://doi.org/10.1016/j.jss.2017.09.018.

Hackett, Jennifer y Graham Hutton (jul. de 2019). «Call-by-Need is Clairvoyant Call-by-Value». En: *Proc. ACM Program. Lang.* 3.ICFP. DOI: 10.1145/3341718. URL: https://doi.org/10.1145/3341718.

Handley, Martin A. T., Niki Vazou y Graham Hutton (dic. de 2019). «Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell». En: *Proc. ACM Program. Lang.* 4.POPL. DOI: 10.1145/3371092. URL: https://doi.org/10.1145/3371092.

Hoare, C. A. R. (oct. de 1969). «An Axiomatic Basis for Computer Programming». En: *Commun. ACM* 12.10, págs. 576-580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

Howe, Douglas J. (1989). «Equality In Lazy Computation Systems». En: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, págs. 198-203. DOI: 10.1109/LICS.1989.39174. URL: https://doi.org/10.1109/LICS.1989.39174.

— (1996). «Proving Congruence of Bisimulation in Functional Programming Languages». En: *Information and Computation* 124.2, págs. 103-112. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1996.0008. URL: https://www.sciencedirect.com/science/article/pii/S0890540196900085.

Hughes, J. (1989). «Why Functional Programming Matters». En: *Computer Journal* 32.2, págs. 98-107.

Hutton, Graham (2016). *Programming in Haskell*. 2nd. USA: Cambridge University Press. ISBN: 1316626229.

Jaskelioff, Mauro (2009). «Modular Monad Transformers». En: *Programming Languages and Systems*. Ed. por Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, págs. 64-79. ISBN: 978-3-642-00590-9.

Jaskelioff, Mauro y Eugenio Moggi (2010a). «Monad transformers as monoid transformers». En: *Theoretical Computer Science* 411.51-52, págs. 4441-4466.

— (2010b). «Monad transformers as monoid transformers». En: *Theoretical Computer Science* 411.51-52, págs. 4441-4466.

Kawahara, Yasuo (1973). «Notes On The Universality Of Relational Functors». En: *Memoirs of the Faculty of Science, Kyushu University. Series A, Mathematics* 27.2, págs. 275-289. DOI: 10.2206/kyushumfs.27.275.

Koutavas, Vasileios, Paul Blain Levy y Eijiro Sumii (2011). «From Applicative to Environmental Bisimulation». En: *Electronic Notes in Theoretical Computer Science* 276. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII), págs. 215-235. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2011.09.023. URL: https://www.sciencedirect.com/science/article/pii/S1571066111001149.

Lahiri, Shuvendu K. et al. (2018). «Program Equivalence (Dagstuhl Seminar 18151)». En: *Dagstuhl Reports* 8.4. Ed. por Shuvendu K. Lahiri et al., págs. 1-19. ISSN: 2192-5283. DOI: 10.4230/DagRep.8.4.1. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9758.

Lassen, Søren B. y Corin Pitcher (1997). «Similarity and Bisimilarity for Countable Non-Determinism and Higher-Order Functions». En: *Electron. Notes Theor. Comput. Sci.* 10, págs. 246-266. DOI: 10.1016/S1571-0661(05)80704-2. URL: https://doi.org/10.1016/S1571-0661(05)80704-2.

Leibniz, Gottfried Wilhelm (1989). «Discourse on Metaphysics». En: *Philosophical Papers and Letters*. Ed. por Leroy E. Loemker. Dordrecht: Springer Netherlands, págs. 303-330. ISBN: 978-94-010-1426-7. DOI: 10.1007/978-94-010-1426-7_36. URL: https://doi.org/10.1007/978-94-010-1426-7_36.

Levy, Paul Blain (2006). «Infinitary Howe's Method». En: *Electronic Notes in Theoretical Computer Science* 164.1. Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS 2006), págs. 85-104. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2006.06.006. URL: https://www.sciencedirect.com/science/article/pii/S1571066106004713.

Levy, PaulBlain, John Power y Hayo Thielecke (2003). «Modelling environments in call-by-value programming languages». En: *Information and Computation* 185.2, págs. 182-210. ISSN: 0890-5401. DOI: https://doi.org/10.1016/S0890-5401(03)00088-9. URL: https://www.sciencedirect.com/science/article/pii/S0890540103000889.

Liang, Sheng, Paul Hudak y Mark Jones (1995). «Monad Transformers and Modular Interpreters». En: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, págs. 333-343. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199528. URL: http://doi.acm.org/10.1145/199448.199528.

Lüth, Christoph y Neil Ghani (2002). «Composing Monads Using Coproducts». En: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. Pittsburgh, PA, USA: Association for Computing Machinery, págs. 133-144. ISBN: 1581134878. DOI: 10.1145/581478.581492. URL: https://doi.org/10.1145/581478.581492.

Marlow, Simon et al. (2010). «Haskell 2010 language report». En: *Available online http://www. haskell. org/(May 2011)*.

Moggi, Eugenio (1989). «Computational Lambda-Calculus and Monads». En: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, California, USA: IEEE Press, págs. 14-23. ISBN: 0818619546.

— (1991). «Notions of computation and monads». En: *Information and Computation* 93.1. Selections from 1989 IEEE Symposium on Logic in Computer Science, págs. 55-92. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(91)90052-4. URL: https://www.sciencedirect.com/science/article/pii/0890540191900524.

Moore, G. E. (2006). «Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.» En: *IEEE Solid-State Circuits Society Newsletter* 11.3, págs. 33-35. DOI: 10.1109/N-SSC.2006.4785860.

Moran, Andrew y David Sands (1999a). «Improvement in a Lazy Context: An Operational Theory for Call-by-Need». En: *Proceedings of the 26th ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, págs. 43-56. ISBN: 1581130953. DOI: 10.1145/292540.292547. URL: https://doi.org/10.1145/292540.292547.

— (1999b). «Improvement in a Lazy Context: An Operational Theory for Call-by-Need». En: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. por Andrew W. Appel y Alex Aiken. ACM, págs. 43-56. DOI: 10.1145/292540.292547. URL: https://doi.org/10.1145/292540.292547.

Moran, Andrew, David Sands y Magnus Carlsson (1999). «Erratic Fudgets: A Semantic Theory for an Embedded Coordination Language». En: *Coordination Languages and Models, Third International Conference, COORDINATION '99, Amsterdam, The Netherlands, April 26-28, 1999, Proceedings*. Ed. por Paolo Ciancarini y Alexander L. Wolf. Vol. 1594. Lecture Notes in Computer Science. Springer, págs. 85-102. DOI: 10.1007/3-540-48919-3_8. URL: https://doi.org/10.1007/3-540-48919-3%5C_8.

Muchnick, Steven S. (1998). *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1558603204.

Okasaki, Chris (1998). *Purely Functional Data Structures*. USA: Cambridge University Press. ISBN: 0521631246.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st. The MIT Press. ISBN: 0262162091.

Pitts, Andrew M. (dic. de 1994). *Some Notes on Inductive and Co-Inductive Techniques in the Semantics of Functional Programs*. Notes Series BRICS-NS-94-5. vi+135 pp, draft version. Department of Computer Science, University of Aarhus: BRICS.

— (2002). «Operational Semantics and Program Equivalence». En: *Applied Semantics*. Ed. por Gilles Barthe et al. Berlin, Heidelberg: Springer Berlin Heidelberg, págs. 378-412. ISBN: 978-3-540-45699-5.

— (2012). «Howe's method for higher-order languages». En: *Advanced Topics in Bisimulation and Coinduction*.

Plotkin, Gordon (1977). «LCF considered as a programming language». En: *Theoretical Computer Science* 5.3, págs. 223-255. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(77)90044-5. URL: https://www.sciencedirect.com/science/article/pii/0304397577900445.

Plotkin, Gordon y John Power (2003). «Algebraic Operations and Generic Effects». En: *Applied Categorical Structures* 11, pág. 2003.

— (oct. de 2004). «Computational Effects and Operations: An Overview». En: *Electr. Notes Theor. Comput. Sci.* 73, págs. 149-163. DOI: 10.1016/j.entcs.2004.08.008.

Pous, Damien y Davide Sangiorgi (2019). «Bisimulation and Coinduction Enhancements: A Historical Perspective». En: *Formal Aspects of Computing* 31.6, págs. 733-749. ISSN: 0934-5043. DOI: 10.1007/s00165-019-00497-w. URL: https://doi.org/10.1007/s00165-019-00497-w.

Ramsey, Norman y Avi Pfeffer (ene. de 2002). «Stochastic Lambda Calculus and Monads of Probability Distributions». En: *SIGPLAN Not.* 37.1, págs. 154-165. ISSN: 0362-1340. DOI: 10.1145/565816.503288. URL: http://doi.acm.org/10.1145/565816.503288.

Sands, David (sep. de 1990). «Calculi for Time Analysis of Functional Programs». Tesis doct. University of London: Department of Computing, Imperial College.

— (ago. de 1991). «Operational Theories of Improvement in Functional Languages (Extended Abstract)». En: *Proceedings of the Fourth Glasgow Workshop on Functional Programming*. Workshops in Computing Series. Skye: Springer-Verlag, págs. 298-311.

— (1998). «Improvement Theory and Its Applications». En: *Higher Order Operational Techniques in Semantics*. Ed. por A. D. Gordon y A. M. Pitts. Publications of the Newton Institute. Cambridge University Press, págs. 275-306.

Schmidt-Schauß, Manfred y David Sabel (2015). «Improvements in a Functional Core Language with Call-by-Need Operational Semantics». En: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP '15. Siena, Italy: Association for Computing Machinery, págs. 220-231. ISBN: 9781450335164. DOI: 10.1145/2790449.2790512. URL: https://doi.org/10.1145/2790449.2790512.

Scott, Dana S. (1976). «Data Types as Lattices». En: *SIAM J. Comput.* 5.3, págs. 522-587. DOI: 10.1137/0205037. URL: https://doi.org/10.1137/0205037.

— (1982). «Lectures on a Mathematical Theory of Computation». En: *Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School, directed by F. L. Bauer, E. W. Dijkstra and C. A. R. Hoare*. Ed. por Manfred Broy y Gunther Schmidt. Dordrecht: Springer Netherlands, págs. 145-292. ISBN: 978-94-009-7893-5. DOI: 10.1007/978-94-009-7893-5_9. URL: https://doi.org/10.1007/978-94-009-7893-5_9.

Søndergaard, H. y P. Sestoft (oct. de 1992). «Non-determinism in Functional Languages». En: *The Computer Journal* 35.5, págs. 514-523. ISSN: 0010-4620. DOI: 10.1093/comjnl/35.5.514. URL: https://doi.org/10.1093/comjnl/35.5.514.

Strassen, V. (1965). «The Existence of Probability Measures with Given Marginals». En: *The Annals of Mathematical Statistics* 36.2, págs. 423-439. DOI: 10.1214/aoms/1177700153. URL: https://doi.org/10.1214/aoms/1177700153.

Turing, A. M. (1937). «On Computable Numbers, with an Application to the Entscheidungsproblem». En: *Proceedings of the London Mathematical Society* s2-42.1, págs. 230-265. DOI: https://doi.org/10.1112/plms/s2-42.1.230. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

Van Stone, Kathryn (2003). «A Denotational Approach to Measuring Complexity in Functional Programs». AAI3262826. Tesis doct. USA. ISBN: 9780549000112.

Veltri, Niccoló y Niels Voorneveld (2022). «Streams of Approximations, Equivalence of Recursive Effectful Programs». En: *Mathematics of Program Construction*. Ed. por Ekaterina Komendantskaya. Cham: Springer International Publishing, págs. 198-221. ISBN: 978-3-031-16912-0.

Apéndice A

Prueba CSE Directo

En este apéndice mostramos una prueba del lema CSE directo utilizado en la prueba de CSE (Sección 10.2). Dividimos la prueba en dos partes: primero construimos una simulación que utilizaremos como un lema auxiliar, y luego definiremos una simulación de mejora que evidencia que CSE Directo es una mejora. Asumimos que estamos trabajando dentro del sistema parcial donde el único efecto observable es la divergencia en la evaluación de términos.

Llamamos a un contexto *cerrado de términos* si es un contexto sin variables libres que al llenar su agujero con un término retorna un término, mientras que llamamos a un contexto *cerrado de valores* si es un contexto sin variables libres que al llenar su agujero con un término nos retorna un valor.

Lema A.1 (CSE Directo)**.** *Sea M un término cerrado, $\mathbb{C}$ un contexto tal que $\mathbb{C}[M] \in \Lambda_0$, y x una variable fresca. Tenemos que:*

$$\textbf{let } x = M \textbf{ in } \mathbb{C}[\checkmark M] \succeq_{\Gamma_\perp} \textbf{let } x = M \textbf{ in } \mathbb{C}[\textbf{ret}(x)]$$

Demostración. Dado M un término cerrado, $\mathbb{C}$ un contexto cerrado de términos tal que y x una variable fresca.

Si observamos los términos $\textbf{let } x = M \textbf{ in } \mathbb{C}[\checkmark M]$ y $\textbf{let } x = M \textbf{ in } \mathbb{C}[\textbf{ret}(x)]$, podemos ver que ambos términos evalúan el término M. Por lo que procedemos por análisis por casos en el resultado de la evaluación del término cerrado M. O bien $[\![M]\!]_{\mathbb{N}_\infty}$ alcanza un valor con un costo o bien la evaluación diverge.

- La evaluación del término M diverge: $[\![M]\!]_{\mathbb{N}_\infty} = \iota_r(\perp)$. Ya que ambas expresiones evalúan la expresión M, y por definición de la mónada de parcialidad, tenemos que ambas expresiones evalúan a $\iota_r(\perp)$. Para evidenciar que es una mejora en este caso nos basta con definir la siguiente relación entre λ-términos y valores.

$$\begin{aligned} DCSE_{\perp\Lambda} &\triangleq Id_{\Lambda_0} \\ &\quad \cup \{(\textbf{let } x = M \textbf{ in } \mathbb{C}[\checkmark M], \textbf{let } x = M \textbf{ in } \mathbb{C}[\textbf{ret}(x)]) \\ &\quad \mid \mathbb{C} \text{ es cerrado de términos}, x \text{ es una variable fresca}\} \\ DCSE_{\perp\mathcal{V}} &\triangleq Id_{\mathcal{V}_0} \end{aligned}$$

  La relación entre λ-términos y valores $(DCSE_{\perp\Lambda}, DCSE_{\perp\mathcal{V}})$ es una simulación de mejoras.

- La evaluación del término M alcanza un valor $V \in \mathcal{V}_0$ con un costo $m \in \mathbb{N}$: $[\![M]\!]_{\mathbb{N}_\infty} = \iota_l(V, m)$.

  Definimos un par de relaciones entre λ-términos y valores en la cual se relacionan todos los posibles términos en donde puede aparecer M con el mismo término pero se utiliza el resultado de la evaluación de M directamente.

  $$\mathcal{R}_\Lambda \triangleq Id_{\Lambda_0} \cup \{(\mathbb{C}[\checkmark M], \mathbb{C}[\mathbf{ret}(V)]) \mid \mathbb{C} \text{ es cerrado de términos }\}$$
  $$\mathcal{R}_\mathcal{V} \triangleq Id_{\mathcal{V}_0} \cup \{(\mathbb{V}[\checkmark M], \mathbb{V}[\mathbf{ret}(V)]) \mid \mathbb{V} \text{ contexto cerrado de valores}\}$$

  Nuestra tarea ahora es mostrar que el par de relaciones $(\mathcal{R}_\Lambda, \mathcal{R}_\mathcal{V})$ forman una simulación de mejoras. Concretamente tenemos que mostrar que respetan valores y que relacionan términos que evalúan a valores relacionados respectando la observación definida por el relacionador $\Gamma_\perp$.

  Sea $\mathbb{C}$ un contexto cerrado de términos , veamos que su evaluación está relacionada por $\Gamma_\perp \mathcal{R}_\mathcal{V}$:

  $$[\![\mathbb{C}[\checkmark M]]\!]_{\mathbb{N}_\infty} \ \Gamma_\perp \mathcal{R}_\mathcal{V} \ [\![\mathbb{C}[\mathbf{ret}(V)]]\!]_{\mathbb{N}_\infty}$$

  Ya que la evaluación es guiada por la sintaxis, procedemos entonces a mostrar que es una simulación aplicativa utilizando el principio coinductivo y análisis por casos sobre el contexto $\mathbb{C}$. A continuación mostramos el caso más informativo donde $\mathbb{C} = \mathbf{let}\ x = \mathbb{D}\ \mathbf{in}\ \mathbb{E}$ para contextos $\mathbb{D}$ y $\mathbb{E}$. El resto de los casos se procede de forma similar.

  De las hipótesis tenemos que $M \in \Lambda_0$ y que $\mathbb{C}$ es un contexto cerrado de términos. Por lo que tenemos que $\mathbb{D}$ es cerrado de términos y pero que $\mathbb{E}$ no lo es, sino que $\mathsf{FV}(\mathbb{E}[M]) \subseteq \{x\}$.

  Nuestro objetivo es mostrar que:

  $$[\![(\mathbf{let}\ x = \mathbb{D}\ \mathbf{in}\ \mathbb{E})[\checkmark M]]\!]_{\mathbb{N}_\infty} \ \Gamma_{\perp_{\mathbb{N}_\infty}} \mathcal{R}_\mathcal{V} \ [\![(\mathbf{let}\ x = \mathbb{D}\ \mathbf{in}\ \mathbb{E})[\mathbf{ret}(V)]]\!]_{\mathbb{N}_\infty}$$

  Por definición de la operación de llenado de agujero, y dado que $M \in \Lambda_0$ y $V \in \mathcal{V}_0$, podemos dar un paso con la evaluación de los términos:

  $$\begin{array}{c} \checkmark [\![\mathbb{D}[\checkmark M]]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto [\![\mathbb{E}[\checkmark M][x := W]]\!]_{\mathbb{N}_\infty} \\ \Gamma_{\perp_{\mathbb{N}_\infty}} \mathcal{R}_\mathcal{V} \\ \checkmark [\![\mathbb{D}[\mathbf{ret}(V)]]\!]_{\mathbb{N}_\infty} >\!\!>\!\!=_{\mathbb{N}_\infty} W \mapsto [\![\mathbb{E}[\mathbf{ret}(V)][x := W]]\!]_{\mathbb{N}_\infty} \end{array}$$

  Podemos eliminar los ticks al comienzo de ambos términos ya que el operador de tick es inyectivo. Además, por definición de T-relacionador (Definición 7.3.3) podemos dividir la prueba en dos.

  - Por un lado $[\![\mathbb{D}[\checkmark M]]\!]_{\mathbb{N}_\infty} \ \Gamma_{\perp_{\mathbb{N}_\infty}} \mathcal{R}_\mathcal{V} \ [\![\mathbb{D}[\mathbf{ret}(V)]]\!]_{\mathbb{N}_\infty}$ se sigue del hecho que $\mathbb{D}$ es un contexto cerrado de términos y de las hipótesis coinductivas.
  - Por el otro lado tenemos que para cuales quiera $W_0, W_1 \in \mathcal{V}_0$ tales que $W_0 \ \mathcal{R}_\mathcal{V} \ W_1$, tenemos que mostrar que:

    $$[\![\mathbb{E}[\checkmark M][x := W_0]]\!]_{\mathbb{N}_\infty} \ \Gamma_{\mathbb{N}_\infty} \mathcal{R}_\mathcal{V} \ [\![\mathbb{E}[\mathbf{ret}(V)][x := W_1]]\!]_{\mathbb{N}_\infty}$$

En éste caso **no** podemos aplicar la hipótesis coinductiva directamente dado que $\mathbb{E}$ puede tener como variable libre a x. Por definición de $\mathcal{R}_\mathcal{V}$ tenemos que si $W_0\ \mathcal{R}_\mathcal{V}\ W_1$ implica dos posibles casos: o bien $W_0 = W_1$ o existe un contexto cerrado de valores $\mathbb{V}$ tal que $W_0 = \mathbb{V}[\checkmark M]$ y $W_1 = \mathbb{V}[\mathbf{ret}(V)]$. Ambos casos tienen la misma estructura por lo que mostramos uno de ellos, dejando el otro como ejercicio para el lector. Sea $\mathbb{V}$ un contexto cerrado de valores y W_0, W_1 tales que $W_0 = \mathbb{V}[\checkmark M]$ y $W_1 = \mathbb{V}[\mathbf{ret}(V)]$.

$$
\begin{array}{rl}
 & [\![\mathbb{E}[\checkmark M][x := W_0]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{Definición de } W_0 \rangle \\
 & [\![\mathbb{E}[\checkmark M][x := \mathbb{V}[\checkmark M]]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle M \in \Lambda_0 \text{ y definición de llenado de agujeros} \rangle \\
 & [\![\mathbb{E}[x := \mathbb{V}][\checkmark M]]\!]_{\mathbb{N}_\infty} \\
\Gamma_{\perp\mathbb{N}_\infty}\mathcal{R}_\mathcal{V} & \langle \mathbb{E}[x := \mathbb{V}] \text{ es cerrado de valores e hipótesis coinductiva} \rangle \\
 & [\![\mathbb{E}[x := \mathbb{V}][\mathbf{ret}(V)]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle V \in \mathcal{V}_0 \text{ y definición de llenado de agujeros} \rangle \\
 & [\![\mathbb{E}[\mathbf{ret}(V)][x := \mathbb{V}[\mathbf{ret}(V)]]]\!]_{\mathbb{N}_\infty} \\
\equiv & \langle \text{ definición de } W_1 \rangle \\
 & [\![\mathbb{E}[\mathbf{ret}(V)][x := W_1]]\!]_{\mathbb{N}_\infty}
\end{array}
$$

Como resultado tenemos que:

$$
\begin{array}{c}
[\![\mathbb{D}[\checkmark M]]\!]_{\mathbb{N}_\infty} >\!\!>\!\!= W \mapsto [\![\mathbb{E}[\checkmark M][x := W]]\!]_{\mathbb{N}_\infty} \\
\Gamma_{\perp\mathbb{N}_\infty}\mathcal{R}_\mathcal{V} \\
[\![\mathbb{D}[\mathbf{ret}(V)]]\!]_{\mathbb{N}_\infty} >\!\!>\!\!= W \mapsto [\![\mathbb{E}[\mathbf{ret}(V)][x := W]]\!]_{\mathbb{N}_\infty}
\end{array}
$$

Los demás casos siguen una estructura similar: intercambiando la substitución con el llenado de agujeros utilizando el hecho que M es cerrado y aplicando la hipótesis coinductiva.

Finalmente mostramos que la relación $\mathcal{R}_\mathcal{V}$ respeta valores. Sea $\mathbb{V}$ un contexto cerrado de valores tal que $\mathbb{V}[\checkmark M]\mathcal{R}_\mathcal{V}\mathbb{V}[\mathbf{ret}(V)]$, y $W \in \mathcal{V}_0$, veamos que:

$$(\mathbb{V}[\checkmark M]\, W)\ \mathcal{R}_\Lambda\ (\mathbb{V}[\mathbf{ret}(V)]\, W)$$

Que se desprende del hecho que al ser $W \in \mathcal{V}_0$ y $\mathbb{V}$ es un contexto cerrado de valores, tenemos que $\mathbb{V}\, W$ es un contexto cerrado de valores. Por hipótesis coinductiva y definición de la operación de llenado de agujeros, tenemos que:

$$(\mathbb{V}\, W)[\checkmark M]\ \mathcal{R}_\Lambda\ (\mathbb{V}\, W)[\mathbf{ret}(V)]$$

Como consecuencia de la prueba anterior podemos concluir que la relación entre λ-términos y valores cerrados $(\mathcal{R}_\Lambda, \mathcal{R}_\mathcal{V})$ es una simulación de mejoras.

Dando un paso hacía atrás, todavía nos queda por probar que CSE directo es una mejora cuando la evaluación del término cerrado M alcanza un valor. Para esto, definimos la siguiente relación entre términos y valores del lenguaje:

$$
\begin{array}{l}
DCSE_\Lambda \triangleq \mathcal{R}_\Lambda \\
\qquad \cup \{(\mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{C}[\checkmark M], \mathbf{let}\ x = M\ \mathbf{in}\ \mathbb{C}[\mathbf{ret}(x)]) \\
\qquad \mid \mathbb{C} \text{ es cerrado de términos y } x \text{ es una variable fresca}\} \\
DCSE_\mathcal{V} \triangleq \mathcal{R}_\mathcal{V}
\end{array}
$$

Para ver por que el par de relaciones recién definidas son una simulación de mejoras, nos concentramos en el nuevo caso que agregamos, ya que el resto es el lema auxiliar ya probado. Sea $\mathbb{C}$ un contexto cerrado de términos y x una variable fresca. Recordemos además la hipótesis sobre $M{:}[\![M]\!]_{\mathbb{N}_\infty} = \iota_l(V, m)$. Veamos que:

$$[\![\textbf{let } x = M \textbf{ in } \mathbb{C}[\checkmark M]]\!]_{\mathbb{N}_\infty} \; \Gamma_{\perp \mathbb{N}_\infty} DCSE_{\mathcal{V}} \; [\![\textbf{let } x = M \textbf{ in } \mathbb{C}[\textbf{ret}(x)]]\!]_{\mathbb{N}_\infty}$$

Aplicando la definición de la evaluación, y por transitividad, es equivalente a mostrar que:

$$\checkmark^{m+1}[\![\mathbb{C}[\checkmark M]]\!]_{\mathbb{N}_\infty} \; \Gamma_{\perp \mathbb{N}_\infty} DCSE_{\mathcal{V}} \; \checkmark^{m+1}[\![\mathbb{C}[\textbf{ret}(V)]]\!]_{\mathbb{N}_\infty}$$

Dado que $\mathbb{C}[\checkmark M] \; \mathcal{R}_\Lambda \; \mathbb{C}[\textbf{ret}(V)]$ y por definición de simulación de mejoras, tenemos que:

$$[\![\mathbb{C}[\checkmark M]]\!]_{\mathbb{N}_\infty} \; \Gamma_{\perp \mathbb{N}_\infty} \mathcal{R}_{\mathcal{V}} \; [\![\mathbb{C}[\textbf{ret}(V)]]\!]_{\mathbb{N}_\infty}$$

Ya que $\mathcal{R}_{\mathcal{V}} \subseteq DCSE_{\mathcal{V}}$ y $\Gamma_{\perp \mathbb{N}_\infty}$ es un relacionador tenemos que:

$$[\![\mathbb{C}[\checkmark M]]\!]_{\mathbb{N}_\infty} \; \Gamma_{\perp \mathbb{N}_\infty} DCSE_{\mathcal{V}} \; [\![\mathbb{C}[\textbf{ret}(V)]]\!]_{\mathbb{N}_\infty}$$

Agregamos los ticks que faltan ($\checkmark^{m+1}$) ya que la operación de tick es monotónica, probando que $(DCSE_\Lambda, DCSE_{\mathcal{V}})$ es una simulación de mejoras.

De esta manera, acumulando los resultados, mostramos que CSE Directo es una mejora proveyendo una simulación de mejora para cada contexto en el sistema parcial. □