UNIVERSIDAD NACIONAL DE ROSARIO

TESINA DE GRADO
PARA LA OBTENCIÓN DEL GRADO DE
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

Alef: Un cálculo de efectos algebraicos con tipado bidireccional.

Autor:
Antonio Locascio

Director:
Dr. Mauro Jaskelioff
Co-Director:
Dr. Exequiel Rivas

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Av. Pellegrini 250, Rosario, Santa Fe, Argentina

13 de julio de 2020

Resumen

El modelado de *efectos computacionales* mediante teorías matemáticas es un instrumento fundamental para facilitar el razonamiento sobre programas y poder garantizar su correctitud. Uno de estos modelos es el basado en la teoría de *efectos algebraicos*, que representa los efectos mediante operaciones de una teoría algebraica que captura su comportamiento.

Para incorporar este modelo al diseño de lenguajes de programación se introducen los *sistemas de efectos*, que extienden los sistemas de tipos con información acerca de los efectos que pueden ser causados por un programa. Sin embargo, las implementaciones de estos sistemas suelen ser muy complejas, perdiendo en gran medida la claridad conceptual provista por los fundamentos teóricos.

En esta tesina se describe un sistema de efectos implementable para un cálculo básico con efectos algebraicos y handlers, usando una nueva variación del *tipado bidireccional* para preservar la claridad y la estrecha relación con la teoría matemática subyacente. Este sistema se prueba seguro respecto a la semántica operacional y se lo implementa en Haskell, mostrando tanto la simplicidad de su formulación como su gran poder expresivo.

Índice general

Resumen III

Índice general IV

1 Introducción **1**
- 1.1. Estado del arte . . . 4
- 1.2. Contribuciones . . . 5
- 1.3. Estructura de la tesis . . . 5

2 Efectos algebraicos y handlers **7**
- 2.1. Introducción . . . 7
- 2.2. Fundamentos teóricos . . . 9
- 2.3. Sistemas de efectos . . . 14

3 Formalismo **17**
- 3.1. Términos . . . 17
- 3.2. Tipos . . . 18
- 3.3. Sistema declarativo . . . 20
- 3.4. Sistema bidireccional . . . 28
- 3.5. Correspondencia de ambos sistemas . . . 35

4 Comparación con otros sistemas **37**
- 4.1. Eff . . . 37
- 4.2. Frank . . . 39
- 4.3. Koka . . . 40

5 Implementación **43**
- 5.1. Estructura del código . . . 43
- 5.2. Azúcar sintáctico . . . 49
- 5.3. Ejemplos . . . 50
- 5.4. Testing . . . 54

6 Demostraciones **57**
- 6.1. Seguridad . . . 57
- 6.2. Correspondencia . . . 64

7 Conclusiones y trabajo futuro **73**
7.1. Trabajo futuro 74

Bibliografía **77**

Capítulo 1

Introducción

La búsqueda de instrumentos que garanticen la confiabilidad y estabilidad de los sistemas informáticos se ha convertido en una tarea de vital importancia. Una manera de afrontar este desafío se basa en aprovechar teorías matemáticas rigurosas para describir e intentar controlar estos sistemas.

Para alcanzar estos objetivos, se introduce la idea de abstraer los programas mediante funciones matemáticas. Esta visión se conoce usualmente como el paradigma de *programación funcional pura* y se caracteriza por considerar a las funciones como simples asociaciones de valores de un dominio a valores de un co-dominio, los cuales se especifican en su tipo. Esto implica que el resultado de la aplicación de una función solo depende de su entrada, siendo determinista sobre esta. En contraste, el resultado de un programa *impuro* puede depender de sus interacciones con el entorno en el que se ejecuta. Concretamente, este podría leer un archivo, pedir entrada por teclado, imprimir un resultado por pantalla o generar un error inesperado, sin que esto se refleje en su tipo. Todos estos comportamientos son ejemplos de *efectos computacionales*, que se pueden describir como las consecuencias laterales del cómputo del valor resultado de un programa.

Restringiendo estos efectos, el paradigma funcional busca simplificar el razonamiento sobre los programas. Sin embargo, para que un sistema de cómputo sea de alguna utilidad, este debe contar con un mecanismo que le permita interactuar con su entorno. De lo contrario, parafraseando a Simon Peyton Jones, un programa es una caja negra, de la cual al ejecutarse solo se sabe que su temperatura aumenta. Por esto es que se busca dar un fundamento teórico a los efectos computacionales que los incorpore a la formalización de los programas. No hacerlo puede dar origen a errores costosos que, además de perjudicar la funcionalidad de un sistema, pueden comprometer su seguridad.

En consecuencia, el estudio de modelos matemáticos para efectos computacionales, como los mencionados, es un área central del campo de la teoría de lenguajes de programación. Estos no solo proveen un fundamento sólido sobre el cual razonar sobre programas con efectos, o *impuros*, sino que también pueden ser aplicados para el diseño de lenguajes de programación.

Uno de estos modelos de efectos computacionales fue introducido por Moggi en su trabajo fundacional [20], en el que propone modelar computaciones encapsulándolas en mónadas, que proveen la estructura necesaria para componer computaciones con efectos. Usando esto, se expande la expresividad de las funciones: lo que antes era una función que tomaba valores de A y devolvía valores de B, es decir, del tipo $A \to B$, ahora se representa en lenguajes *call-by-value* con el tipo $A \to T\ B$, donde la mónada T captura los efectos de esta computación.

En base a este concepto, originado en teoría de categorías, Moggi consigue proveer una semántica denotacional para programas impuros. Adicionalmente, estas ideas fueron llevadas a la práctica del diseño de lenguajes de programación por Wadler, que mostró su utilidad para expresar efectos laterales en lenguajes funcionales [32].

Efectos algebraicos

El enfoque monádico, sin embargo, deja de lado la semántica operacional de los programas con efectos. Como respuesta, Plotkin y Power proponen un nuevo abordaje para el modelado de computaciones, en el que se pone el foco en las *operaciones* que dan origen a los efectos computacionales [25]. Estas forman una *signatura* o *firma*, desde la cual se construyen los términos del lenguaje.

Las operaciones se presentan junto a un conjunto de *ecuaciones* que describen su comportamiento, resultando en un álgebra de programas que provee su semántica. Plotkin y Power muestran, adicionalmente, que estas teorías ecuacionales sobre las signaturas determinan las mismas mónadas que se utilizaban para modelar estos efectos, consiguiendo relacionar de manera teórica ambos enfoques [26].

Esta nueva teoría, sin embargo, no es suficiente para modelar el comportamiento de ciertas operaciones, como el manejo de excepciones. Para poder hacerlo, Plotkin y Pretnar la expanden introduciendo el concepto de *handlers de efectos* [24], que permiten manejar las operaciones de manera análoga a como los handlers de excepciones tradicionales manejan estas últimas. Haciendo esto, los handlers son destructores de efectos, que dan interpretaciones concretas a las operaciones de una teoría algebraica. En consecuencia, esta nueva construcción constituye una abstracción que le permite al programador separar la interfaz de un efecto de su implementación.

Hasta este punto, el desarrollo mostrado de la teoría de efectos algebraicos se focalizó en el objetivo de describir matemáticamente programas con efectos. Para poder utilizar estas ideas para controlar los efectos causados por un programa escrito en un lenguaje dado, se debe embeber esta teoría en el sistema de tipos de este último. Esto implica que los tipos, además de describir la colección de valores en los que puede resultar una computación, tienen que llevar información sobre qué operaciones pueden ser invocadas al ejecutarse esta.

Agregar esta información a los tipos permite garantizar estáticamente que ninguna operación será invocada de manera inesperada en tiempo de ejecución. Esto, además de ser útil para que los usuarios tengan un mejor entendimiento de sus pro-

gramas y les sea más fácil encontrar errores en estos, puede ser usado por compiladores para optimizar y paralelizar programas de forma segura [14].

De esta manera surgen los llamados sistemas de tipos y efectos, o *sistemas de efectos* para abreviar. Para que estos tengan alguna utilidad, se espera que sean precisos respecto a la información de tipos. Esto implica que el sistema no requiera agregar operaciones al tipo de un término cuando estas no son necesarias. Si bien esto se puede expresar de manera sencilla en un sistema de tipos *declarativo*, el cual no determina el mecanismo por el que se chequean los tipos, las implementaciones del chequeo de tipos suelen ser significativamente más complejas.

Por esta razón, el problema de cómo implementar estos sistemas de efectos ha recibido mucha atención en los últimos años. Si bien se han presentado diversas soluciones potenciales, mencionadas en la Sección 1.1, el problema sigue estando abierto. El objetivo de esta tesina, entonces, es formular un sistema de efectos implementable para un cálculo básico con efectos algebraicos, llamado *Alef* (o $\aleph$), que se mantenga conceptualmente cercano a los sistemas declarativos.

Tipado bidireccional

Existen distintas formas de implementar un sistema de tipos. Por un lado se puede hacer chequeo, en el cual el tipo de la expresión es siempre una entrada al juicio de tipado. Esto implica que el código debe contar con anotaciones de tipos en muchos términos, lo que hace que estos sean poco usables en la práctica.

Opuestos a estos se encuentran los sistemas de inferencia, para los que el tipo de una expresión siempre es una salida del juicio de tipado. Si bien estos son ampliamente usados en la práctica, requieren mecanismos complejos, perdiendo cierta claridad conceptual. Adicionalmente, para poder permitir inferencia la expresividad del sistema de tipos puede verse limitada.

Pierce y Turner describen una técnica que busca tomar lo mejor de los dos métodos anteriores, y la llaman *tipado bidireccional* [21]. Esta consiste en reemplazar el juicio de tipado por dos juicios: uno de chequeo, que tiene el tipo como entrada y otro de síntesis, que devuelve el tipo como salida. Intercalando entre estos dos modos se busca propagar información de tipado hacia las sub-expresiones de los términos, reduciendo la necesidad de anotaciones.

Este método simplifica la tarea de hacer implementables ciertos sistemas declarativos, ya que su formulación resulta en reglas dirigidas por sintaxis. Sin embargo, a diferencia de los algoritmos de inferencia, los sistemas bidireccionales suelen mantener en mayor medida la simplicidad conceptual de los declarativos, al no depender de mecanismo de resolución de restricciones externos al tipado como los primeros [28]. Por esta razón, para hacer el sistema de efectos de *Alef* implementable se utilizará la técnica de tipado bidireccional, cuyo resultado se presenta en la Sección 3.4.

1.1. Estado del arte

La relación establecida entre el enfoque monádico y el de efectos algebraicos también se ve reflejada en la práctica, ya que ambas teorías sirven de base para implementar efectos computacionales modulares en lenguajes funcionales. Si bien ambos enfoques tienen marcadas diferencias —principalmente que el monádico requiere determinar el modelo de computación previo a expresar los programas, mientras que en el segundo esto se hace mediante la aplicación de handlers de manera posterior— Schrijvers, Piróg, Wu y Jaskelioff [31] muestran una correspondencia entre estos, logrando expresar, bajo ciertos supuestos, cada uno en términos del otro.

La teoría de efectos algebraicos y handlers se ha incorporado en la práctica en distintos niveles: nuevos lenguajes se han diseñado alrededor de esta, se han extendido lenguajes para añadirles sus construcciones y se han desarrollado librerías que la implementan en diversos lenguajes.

Lenguajes con efectos algebraicos

Entre estos lenguajes se encuentra *Eff* [2, 3], de Bauer y Pretnar, que concretiza los desarrollos teóricos mencionados en la sección anterior. Este lenguaje es presentado junto a un sistema de efectos declarativo [2], que será la base del de *Alef*. Posteriormente, Pretnar propone su implementación mediante un sistema de inferencia [28], pero solo para un subconjunto del lenguaje. Adicionalmente, Saleh, Karachalias, Pretnar y Schrijvers, se basan en el anterior para formular un nuevo sistema de inferencia [30] basado en la elaboración de Eff a un lenguaje con subtipado explícito, que facilita su compilación eficiente. Sin embargo, la implementación actual de Eff no cuenta con ningún sistema de efectos.

Otro lenguaje diseñado en base a la teoría de efectos algebraicos es *Frank* [19, 8], de Lindley, McBride y McLaughlin. En contraste con el anterior, este se aleja del planteo original generalizando las funciones y los handlers mediante una única abstracción, los *operadores*, que permiten manejar múltiples computaciones simultáneamente. Su sistema de efectos hace uso de la técnica de tipado bidireccional e incorpora polimorfismo de efectos mediante una variación de polimorfismo de fila [29]. A pesar de que la forma de los tipos de este lenguaje difieren de los usados en *Alef*, su sistema de efectos fue el fundamento de la aplicabilidad del tipado bidireccional en el contexto de efectos algebraicos.

Diseñado por Leijen, *Koka* [17] también incorpora la teoría de efectos algebraicos en el diseño de su sistema de control de efectos. Si bien su presentación inicial no incluye una construcción para manejar operaciones, esta fue introducida en posteriores versiones del lenguaje [18]. Koka cuenta con un sistema de efectos con inferencia Hindley-Milner, que también hace uso de una variación de polimorfismo de filas para expresar polimorfismo de efectos. Las filas de efectos de Koka sirvieron como inspiración para las de *Alef*.

Extensiones de lenguajes y librerías

Este segundo grupo incluye la extensión a *Links* [11] de Hillerström y Lindley y la extensión a OCaml, *Multicore OCaml* [9], de Dolan y otros. Ambas se construyen en base a efectos algebraicos y handlers similares a los presentados teóricamente, pero la segunda no cuenta con un sistema de efectos. Adicionalmente, como se mencionó al inicio de esta sección, existen librerías que implementan estas ideas en Haskell [12, 15, 22], Idris [6], PureScript [10], Scala [5] y más.

En el Capítulo 4 se hace una comparación más detallada de los principales sistemas de efectos mencionados en esta sección con el sistema de efectos desarrollado para *Alef*.

1.2. Contribuciones

Las contribuciones concretas de esta tesina son:

- Dos sistemas de efectos nuevos para un lenguaje basado en el núcleo de Eff, llamado *Alef*. Uno de estos es declarativo y adapta el sistema de Eff [2] para acomodar polimorfismo de fila en los efectos. El otro es implementable mediante tipado bidireccional, y da una versión dirigida por sintaxis del anterior.

- Proveer síntesis de efectos en el sistema bidireccional, diferenciándose de usos previos de esta técnica en las cuales los efectos son siempre chequeados. Esto permite alcanzar tipos precisos con menos anotaciones y mejores mensajes de error.

- Una nueva forma de ver el tipado bidireccional, agregando instanciaciones explícitas generadas por ambos juicios. Al estar incluido el operador de unificación en la descripción del sistema, se puede razonar sobre el mismo sistema que se implementa, a diferencia de otros sistemas cuyas implementaciones dependen de mecanismos no presentes en sus descripciones formales.

- Pruebas de correctitud, tanto de la seguridad de tipos para el sistema de efectos declarativo, como de correspondencia entre los dos sistemas de efectos.

- Una implementación en Haskell de *Alef* (`Alef`), que agrega azúcar sintáctico al cálculo básico para simplificar su escritura. Adicionalmente se proveen ejemplos de programas que muestran las distintas características de la programación en `Alef`.

1.3. Estructura de la tesis

El resto de la tesis tiene la siguiente estructura. El Capítulo 2 contiene una introducción a la teoría de efectos algebraicos. En el Capítulo 3 se introduce la formalización de *Alef*. Este último es comparado con otros lenguajes con sistemas de efectos

en el Capítulo 4. El Capítulo 5, por su parte, presenta la implementación Haskell de *Alef*. En el Capítulo 6 se desarrollan las demostraciones de los resultados del Capítulo 3. Finalmente, el Capítulo 7 se presentan las conclusiones finales de la tesis junto al trabajo futuro.

Capítulo 2

Efectos algebraicos y handlers

El presente capítulo sirve de introducción a los conceptos fundamentales de la teoría de efectos algebraicos y handlers. En primer lugar, se exponen las ideas centrales intuitivamente por medio de ejemplos. Luego, se desarrollan brevemente los fundamentos matemáticos de esta teoría. Finalmente, se describen los sistemas de efectos que llevan estos conceptos a la práctica.

2.1. Introducción

El modelo de efectos algebraicos le da central importancia a las operaciones que interactúan con el ambiente en el cual se ejecuta el programa. Algunos ejemplos usuales de estas son `Read` y `Print` para entrada y salida, `Get` y `Set` para computaciones con estado, y `Throw` para excepciones. Estas operaciones son la fuente de efectos secundarios y son un componente básico del lenguaje.

En concreto, un ejemplo de una computación en este modelo se presenta en la Figura 2.1. Si bien es posible inferir una interpretación para c, esta expresión solo describe tres llamados secuenciales a una operación abstracta `Print`, sin mencionar su significado.

$$c = \texttt{Print}(1);\ \texttt{Print}(2);\ \texttt{Print}(3)$$

Figura 2.1: Definición de la computación c.

Para darle una interpretación concreta, se introduce otro concepto central: los handlers. Estos corresponden a construcciones que definen cómo manejar los llamados a las operaciones como las anteriores y pueden verse como una generalización de los manejadores de excepciones usuales. De esta manera, los handlers representan destructores de computaciones que invocan operaciones de efectos.

Para plasmar esta noción en una caso concreto, la Figura 2.2 presenta como primer ejemplo un handler que cuenta el numero de invocaciones a la operación `Print`

de un programa.

$$\begin{aligned} count = \ &\texttt{handler}\ \texttt{val}\ x \mapsto \texttt{val}\ 0, \\ &\{\texttt{Print}\ x\ k \mapsto \texttt{let}\ n := k\ ()\ \texttt{in}\ \texttt{val}\ (n+1)\} \end{aligned}$$

Figura 2.2: Definición del handler *count*.

Esto se lee de la siguiente manera: *count* es un handler que frente a una computación que solo retorna un valor, la maneja devolviendo el valor 0. Si la computación manejada es una invocación a `Print` con argumento x y el resto del programa está capturado en una continuación k, se ejecuta esta continuación guardando su resultado en n, el cual se retorna luego de sumarle 1.

Al aplicar este handler a la computación anterior ($\texttt{with}\ count\ \texttt{handle}\ c$), se obtiene el valor 3, que es el número de veces que se ejecuta la operación.

Sin embargo, esta no es la única interpretación posible para `Print`. Para dar otra interpretación distinta basta con definir un nuevo handler que la maneja. Por ejemplo, en la Figura 2.3 se define *add*. Este handler es muy similar a *count*, pero en este caso se suman los argumentos pasados a cada invocación de `Print`. Si se aplicara a c, en este caso se obtendría como resultado el valor 6.

$$\begin{aligned} add = \ &\texttt{handler}\ \texttt{val}\ x \mapsto \texttt{val}\ 0, \\ &\{\texttt{Print}\ x\ k \mapsto \texttt{let}\ n := k\ ()\ \texttt{in}\ \texttt{val}\ (n+x)\} \end{aligned}$$

Figura 2.3: Definición del handler *add*.

Estos ejemplos muestran que los handlers no solo permiten dar una interpretación a las operaciones abstractas, sino que también representan una abstracción que hace posible desligar la construcción de programas con efectos de la implementación de estos últimos.

El entorno de ejecución deberá manejar ciertas operaciones, como es el caso de `Print`. De esta manera, todos los llamados a esta operación que no sean manejados por un handler serán invocadas por el entorno, efectivamente imprimiendo por pantalla el valor pasado.

Para mostrar otra característica de este enfoque, se tiene como ejemplo la computación c' definida en la Figura 2.4. Esta computación lee un valor y lo retorna si es mayor a cero. En caso contrario, lanza una excepción. Como la continuación de esta última operación requiere como entrada un valor del tipo vacío, esta continuación nunca será ejecutada.

$$
\begin{aligned}
c' = \;& \texttt{let}\ x \coloneqq \texttt{Read}()\ \texttt{in} \\
& \texttt{if}\ x > 0\ \texttt{then}\ (\texttt{val}\ x)\ \texttt{else}\ (\texttt{Throw}(0, \texttt{fun}\ y \mapsto \texttt{val}\ 0))
\end{aligned}
$$

Figura 2.4: Definición de la computación c'.

Para este ejemplo se definirán adicionalmente dos handlers, que se presentan en la Figura 2.5. El primero, *z_input*, maneja la operación de lectura haciendo que esta siempre lea la constante 0. El segundo, *exc_pr*, simplemente maneja las excepciones lanzadas imprimiendo el valor de sus argumentos. Como puede verse en el llamado a `Print` de este ejemplo, las continuaciones triviales pueden ser omitidas, como se explica en la Sección 3.1.

$$
\begin{aligned}
z_input = \;& \texttt{handler}\ \texttt{val}\ x \mapsto \texttt{val}\ x, \\
& \qquad\qquad \{\texttt{Read}\ x\ k \mapsto k\ 0\} \\
exc_pr = \;& \texttt{handler}\ \texttt{val}\ x \mapsto \texttt{val}\ x, \\
& \qquad\qquad \{\texttt{Throw}\ x\ k \mapsto \texttt{Print}(x)\}
\end{aligned}
$$

Figura 2.5: Definición de los handlers *z_input* y *exc_pr*.

Luego, se podrían aplicar ambos handlers a c' secuencialmente de manera trivial:

$$\texttt{with}\ exc_pr\ \texttt{handle}\ (\texttt{with}\ z_input\ \texttt{handle}\ c')$$

resultando en una computación que imprime 0.

De esta manera se expone otra ventaja de este modelo por sobre otros: los handlers permiten componer implementaciones de efectos de manera simple, sin necesidad de construcciones adicionales.

2.2. Fundamentos teóricos

Como se mencionó en el capítulo anterior, existen distintas teorías que buscan modelar los efectos computacionales matemáticamente. En esta sección se presentan brevemente los conceptos básicos de la teoría de efectos algebraicos. La presentación está basada en las notas introductorias al área de Andrej Bauer [1].

El modelo de efectos algebraicos se basa en considerar un álgebra cuyos elementos son computaciones de dos tipos:

- Computaciones puras de la forma `val` v.

- Computaciones con efectos de la forma $\texttt{Op}\ p\ k$.

Para definirla, se supone que se cuenta con un conjunto V de valores y una signatura $\Sigma = \{\texttt{Op}_i : P_i \to A_i\}_i$ que representa una colección de símbolos de operación, cada uno asociado a un conjunto P_i, llamado parámetro, y otro conjunto A_i, llamado aridad. Por ejemplo, la signatura con los efectos de lectura, escritura y excepciones es:

$$
\begin{aligned}
\Sigma = \{&\texttt{Read} : \texttt{unit} \to \texttt{nat}, \\
&\texttt{Print} : \texttt{nat} \to \texttt{unit}, \\
&\texttt{Throw} : \texttt{nat} \to \texttt{empty}\}
\end{aligned}
$$

Términos

Para comenzar se definen los *términos* del álgebra, que son descriptos por árboles pertenecientes al conjunto $V\langle\Sigma\rangle$, donde:

- Si $v \in V$, luego $\texttt{val}\ v \in V\langle\Sigma\rangle$.
- Si $p \in P_i$, $k : A_i \to V\langle\Sigma\rangle$ y $\texttt{Op}_i : P_i \to A_i \in \Sigma$, luego $\texttt{Op}_i(p, k) \in V\langle\Sigma\rangle$ corresponde al árbol con raíz $\texttt{Op}_i$ y cuyos sub-árboles son descriptos por k.

Por ejemplo, la Figura 2.6 muestra el árbol que describe la computación c' de la sección anterior.

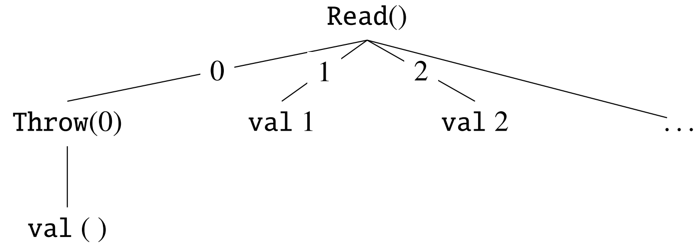


Figura 2.6: Árbol que describe c'.

Ecuaciones

En base a los términos se pueden definir ecuaciones que capturan el comportamiento de las operaciones. Estas tienen la forma $(t_1 = t_2)$, donde t_1 y t_2 son términos. Por ejemplo, tomemos la signatura del efecto Estado:

$$\Sigma = \{\texttt{Get} : \texttt{unit} \to \texttt{nat},\ \texttt{Set} : \texttt{nat} \to \texttt{unit}\}$$

Las ecuaciones que describen el comportamiento esperado de estas operaciones, originalmente presentadas por Plotkin y Power [26], son:

$$\texttt{Get}((\,),\ \texttt{fun}\ s \mapsto \texttt{Get}((\,),\ \texttt{fun}\ t \mapsto k\ s\ t)) = \texttt{Get}((\,),\ \texttt{fun}\ s \mapsto k\ s\ s) \quad (1)$$

Indicando que dos llamadas consecutivas a `Get` deben recibir el mismo valor.

$$\texttt{Get}((\,),\ \texttt{fun}\ s \mapsto \texttt{Set}(s,\ k)) = k() \quad (2)$$

Que permite cancelar un `Get` seguido de un `Set` cuando en la segunda se usa el valor obtenido en la primera.

$$\texttt{Set}(s,\ \texttt{fun}\ _ \mapsto \texttt{Get}((\,),\ k)) = \texttt{Set}(s, \texttt{fun}\ _ \mapsto k\ s) \quad (3)$$

Establece que si se llama a `Get` luego de hacer `Set`, el valor obtenido será el usado para `Set`.

$$\texttt{Set}(s,\ \texttt{fun}\ _ \mapsto \texttt{Set}(t,\ k)) = \texttt{Set}(t,\ k) \quad (4)$$

Que indica que si se hacen dos `Set` seguidos, el primero es irrelevante.

Estas ecuaciones dan origen a una relación de equivalencia en $V\langle\Sigma\rangle$. Siguiendo con el ejemplo de Estado, se considera el término *statec*, definido en la Figura 2.7.

$$\begin{aligned} \textit{statec} ::= {} & \texttt{Get}((\,),\ \texttt{fun}\ a \mapsto \\ & \texttt{Get}((\,),\ \texttt{fun}\ b \mapsto \\ & \texttt{Set}(a+1,\ \texttt{fun}\ _ \mapsto \\ & \texttt{Set}(a+2,\ k)))) \end{aligned}$$

Figura 2.7: Definición de la computación *statec*.

El árbol que representa esta computación se introduce en la Figura 2.8. De acuerdo a las ecuaciones anteriores, en particular la (1) y la (4), el árbol anterior es equivalente al árbol presentado en la Figura 2.9.

Como muestra este ejemplo, las ecuaciones, además de describir el comportamiento de las operaciones, introducen una noción de equivalencia de programas que puede ser aprovechada para garantizar la admisibilidad de ciertas optimizaciones.

Sin embargo, ninguno de los sistemas de efectos en la práctica incorporan ecuaciones. Hacerlo no solo requeriría que el usuario provea demostraciones respecto a sus programas, sino que también restringiría el uso de implementaciones usuales que no las cumplen. Por esto, las ecuaciones no serán tenidas en cuenta en el desarrollo de esta tesina.

Teorías algebraicas y sus modelos

Una *teoría algebraica* T se define como un par (Σ, E), donde E es un conjunto de ecuaciones sobre las operaciones de Σ.

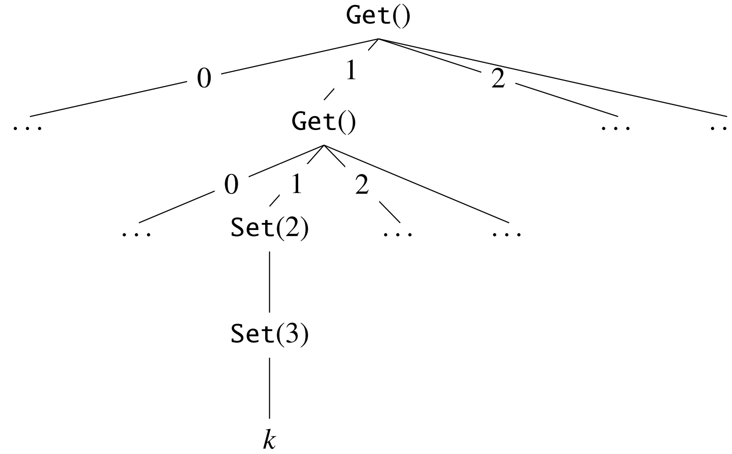


Figura 2.8: Árbol que representa la computación *statec*.

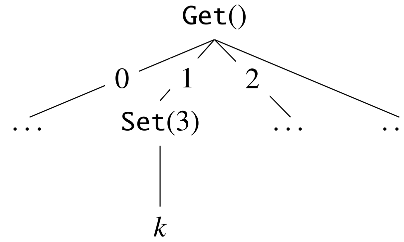


Figura 2.9: Árbol equivalente al del la Figura 2.8.

Dada una teoría T, un *T-modelo* M es definido por:

- Un conjunto portador $|M|$.
- Para cada operación $\mathtt{Op}_i : P_i \to A_i \in \Sigma$, un mapa $[\![\mathtt{Op}_i]\!]_M : P_i \times |M|^{A_i} \to |M|$ que da una interpretación de la operación sobre el conjunto portador $|M|$.

Estas últimas interpretaciones pueden extenderse a los términos de $V\langle\Sigma\rangle$, obteniendo para cada término un mapa $[\![t]\!]_M : |M|^V \to |M|$. Un modelo es válido si para toda ecuación $(t_1 = t_2) \in E$, vale que $[\![t_1]\!]_M = [\![t_2]\!]_M$.

Como se puede tener más de un modelo para una teoría T, suele resultar útil establecer relaciones entre los diferentes modelos. Para esto se introduce el concepto de *T-Homomorfismo*, que se define de la siguiente manera. Sean L y M dos T-modelos, un T-homomorfismo de L a M (notado como $L \twoheadrightarrow M$) es un mapa $h : |L| \to |M|$ que conmuta con las operaciones, es decir, tal que vale:

$$h \circ [\![\mathtt{Op}_i]\!]_L = [\![\mathtt{Op}_i]\!]_M \circ (id_{P_i} \times \underbrace{(h, h, \ldots, h)}_{A_i})$$

Ahora bien, si el objetivo de esta formalización es representar efectos computacionales, es natural preguntarse qué modelo es el más adecuado para describirlos. Para responder esto, se busca aquel que asuma lo menos posible acerca de la teoría en cuestión. De esta manera, se alcanza la definición del párrafo siguiente.

Dada una teoría T y un conjunto V, el *T-modelo libre* generado por V es un modelo M junto a un mapa $\eta : V \to |M|$ tal que para todo T-modelo L y todo mapa $f : V \to |L|$ existe un único T-homomorfismo $\overline{f} : M \twoheadrightarrow L$ que hace conmutar el diagrama:

$$\begin{array}{ccc} V & \xrightarrow{\eta} & |M| \\ & {\scriptstyle f}\searrow & \downarrow{\scriptstyle \overline{f}} \\ & & |L| \end{array}$$

Esta última propiedad se conoce como la *propiedad universal* del T-modelo libre sobre V, e indica que M es la forma más simple (que menos asume) de conseguir un T-modelo partiendo de V.

Como en este desarrollo se tienen en cuenta teorías sin ecuaciones, el modelo libre de $T = (\Sigma, \emptyset)$ generado por V no es más que el conjunto de árboles que describen sus términos, $V\langle\Sigma\rangle$, con $\eta = \texttt{val}$. En este caso, si $\texttt{Op}_i : P_i \to A_i \in \Sigma$, el mapa $[\![\texttt{Op}_i]\!]$ es el que, dados $p : P_i$ y $k : A_i \to V\langle\Sigma\rangle$, les asigna el árbol $\texttt{Op}_i(p, k)$. En adelante se hará abuso de notación, escribiendo $V\langle\Sigma\rangle$ para referirse a este modelo libre.

Resumiendo lo anterior, una computación que retorna valores de V y ejecuta operaciones en Σ es un elemento del T-modelo libre $V\langle\Sigma\rangle$. Sin embargo, falta formalizar una parte fundamental del cálculo presentado en la sección previa: los handlers.

Handlers

Como se mostró antes, un handler maneja computaciones cambiando tanto los valores que retornan como las operaciones que invocan. De esta manera, un handler puede verse como una transformación de un modelo $V\langle\Sigma\rangle$ a otro $V'\langle\Sigma'\rangle$.

Para que esta transformación preserve la estructura algebraica de las computaciones, se busca que un handler h sea un T-homomorfismo. En particular, como se quiere aprovechar la propiedad universal, h debe ser un T-homomorfismo con $T = (\Sigma, \emptyset)$. Esto se debe a que esta propiedad garantiza un homomorfismo saliendo del modelo libre.

En consecuencia, además de una inyección de los valores de V en $V'\langle\Sigma'\rangle$, se debe proveer un T-modelo en $V'\langle\Sigma'\rangle$. En concreto, un handler $h : V\langle\Sigma\rangle \twoheadrightarrow V'\langle\Sigma'\rangle$ se define mediante:

- Un mapa $f : V \to V'\langle\Sigma'\rangle$.
- Para cada $\texttt{Op}_i : P_i \to A_i \in \Sigma$, un mapa $h_i : P_i \times V'\langle\Sigma'\rangle^{A_i} \to V'\langle\Sigma'\rangle$ que forme un T-modelo en $V'\langle\Sigma'\rangle$.

Luego, por la propiedad universal y la definición de modelo, $h : V\langle\Sigma\rangle \twoheadrightarrow V'\langle\Sigma'\rangle$ es el único mapa que satisface:

$$h(\mathtt{val}\ v) = f(v)$$
$$h(\mathtt{Op}_i(p,k)) = h_i(p, h \circ k)$$

Recapitulando, el concepto de un handler corresponde con un homomorfismo entre modelos libres, es decir, una transformación de los árboles de computación que preserva su estructura algebraica.

Todo el desarrollo teórico de esta sección no solo sirve para modelar matemáticamente los efectos computacionales, sino que también son la base de la semántica denotacional de los tipos en lenguajes que implementan efectos algebraicos [2].

2.3. Sistemas de efectos

Como se introdujo en el Capítulo 1, muchos de los lenguajes que implementan estas nociones establecen un control estático sobre los efectos computacionales de sus programas mediante alguna variante de un sistema de efectos.

La teoría de efectos algebraicos presentada en la sección anterior provee una base sobre la cual diseñar un sistema de tipos y efectos. Para hacerlo, los conjuntos de valores se consideran *tipos valor*, mientras que los modelos libres sobre teorías sin ecuaciones corresponden a los *tipos computación*. En concreto, un término de tipo $\mathtt{nat}\langle\mathtt{Get},\ \mathtt{Set}\rangle$, por ejemplo, corresponde a una computación que puede ejecutar las operaciones del efecto Estado y retorna un número natural.

Se incluyen también dos constructores de tipos: uno para las funciones, que tienen como dominio un tipo valor y codominio un tipo computación, que se notará $A \rightarrow C\langle\Sigma\rangle$, y otro para handlers, para el que ambos son tipos computación, que usarán la notación de la sección anterior $A\langle\Sigma\rangle \twoheadrightarrow B\langle\Sigma'\rangle$.

La principal diferencia de estos sistemas respecto de la teoría se encuentra en que estos suelen incorporar algún mecanismo para extender los efectos asociados a un tipo. Esto permite implementar *polimorfismo de efectos*, que es fundamental para definir componentes reutilizables y así evitar duplicación de código.

Para ejemplificar la utilidad de esto último, se vuelve al handler *count* presentado en la Sección 2.1, cuya definición se encuentra en la Figura 2.2.

El tipo que le corresponde a este handler es $\mathtt{unit}\langle\mathtt{Print}\rangle \twoheadrightarrow \mathtt{nat}\langle\rangle$, donde el tipo de la computación resultante indica que esta no puede invocar ninguna operación. El problema radica en que el tipo de la computación entrada de *count* es demasiado restrictivo, requiere que esta a lo sumo pueda invocar `Print`. Sin embargo, en la práctica se podría querer aplicar este handler sobre computaciones con más efectos, ignorando todas las operaciones distintas de `Print`. Para poder hacer esto, los sistemas de efectos implementan polimorfismo de efectos, que permiten extender la información de los efectos presentes en una computación.

Dos técnicas utilizadas para alcanzar este objetivo son el *subtipado de efectos* y el *polimorfismo de fila*, siendo una variante de este último la alternativa elegida

para *Alef*. Esta consiste en considerar tipos computación de la forma $V\langle\mathcal{E}\rangle$, donde $\mathcal{E}$ es una fila conformada por un conjunto de operaciones Δ y una variable de efectos μ. Para extender este tipo con operaciones adicionales, basta con instanciar μ con una nueva fila que las agregue. Al producirse esta instanciación, se introduce a la fila una nueva variable de efectos que puede ser instanciada subsecuentemente. De esta forma, alcanza con tener una sola variable de efectos por fila. En el Capítulo 3 se describe la aplicación de polimorfismo de fila sobre los tipos computación con mayor detalle.

Volviendo al ejemplo, el tipo de *count* en un sistema con polimorfismo de efectos de fila pasa a ser $\texttt{unit}\langle\texttt{Print} \mid \mu\rangle \twoheadrightarrow \texttt{nat}\langle\mu\rangle$. Si ahora se quiere manejar una computación que además cuenta con la operación $\texttt{Op}$, basta instanciar μ a $\langle\texttt{Op} \mid \mu'\rangle$, haciendo que el tipo de *count* sea $\texttt{unit}\langle\texttt{Print}, \texttt{Op} \mid \mu'\rangle \twoheadrightarrow \texttt{nat}\langle\texttt{Op} \mid \mu'\rangle$. Como se mencionó anteriormente, se busca que las operaciones no manejadas por un handler sean ignoradas. Por esto el dominio y codominio del tipo de un handler tienen la misma variable de efectos, haciendo que el tipo de la computación resultante de *count* incluya la operación $\texttt{Op}$, ya que esta es "dejada pasar" por el handler.

La estrecha relación entre la teoría y este sistema de efectos tiene como consecuencia que los fundamentos matemáticos pueden utilizarse como semántica denotacional del lenguaje. Esto significa que los tipos definidos de esta manera facilitan el razonamiento sobre los programas permitiendo hacer uso de los resultados teóricos.

Capítulo 3

Formalismo

En este capítulo se presenta la formalización de *Alef*, el lenguaje desarrollado para esta tesina. En primer lugar se define la sintaxis de términos —basada en la de *Eff*, de Bauer y Pretnar [3]— y de tipos, junto a una noción de orden y equivalencia para estos últimos.

Seguidamente se introduce un sistema de tipos declarativo para un subconjunto del lenguaje, obviando ciertas construcciones para reducir el tamaño de las pruebas. Sobre este se define una semántica denotacional, una semántica operacional *small-step* y se prueba la seguridad de esta última respecto al tipado. Por último, se presenta un sistema de tipos implementable, basado en tipado bidireccional, que se demuestra consistente y completo respecto al declarativo.

3.1. Términos

La sintaxis está dada por la siguiente gramática. Los términos se encuentran divididos entre expresiones y computaciones, que pueden ejecutar operaciones que provocan efectos computacionales antes de devolver un valor.

$$
\begin{array}{llcl}
\textbf{Expresiones} & e & ::= & x \mid \texttt{true} \mid \texttt{false} \mid 0 \mid \texttt{succ}\ e \mid (\,) \mid \texttt{fun}\ x \mapsto c \mid \\
 & & & (e : A) \mid h \\
\textbf{Handler} & h & ::= & \texttt{handler}\ \texttt{val}\ x \mapsto c_v,\ \{\texttt{Op}_i\ x\ k \mapsto c_i\}_{i=1}^{n} \\
\textbf{Computaciones} & c & ::= & \texttt{val}\ e \mid \texttt{Op}\ e\ (y.c) \mid \texttt{with}\ e\ \texttt{handle}\ c \mid e_1\ e_2 \mid \\
 & & & \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{let}\ x \coloneqq c_1\ \texttt{in}\ c_2 \mid \\
 & & & (\texttt{match}\ e\ \texttt{with}\ 0 \mapsto c_1 \mid \texttt{succ}\ x \mapsto c_2)
\end{array}
$$

Las letras x, y y z representan variables. Un handler se compone de una computación c_v con la que se manejan los valores y de computaciones c_i para manejar cada operación. En la llamada a una operación, además de un valor argumento, se proporciona una continuación que captura el resto del programa.

Para simplificar la invocación de operaciones, se introduce azúcar sintáctico que permite utilizar *efectos genéricos* [23]:

$$\texttt{Op}(e) ::= \texttt{Op}\ e\ (z.\texttt{val}\ z)$$

Las expresiones pueden contar con anotaciones de tipo, que son necesarias para el sistema de tipado bidireccional. La sintaxis de estos tipos se da en la siguiente sección.

3.2. Tipos

Tipo valor	$A,\ B,\ C$	$::=$	$\texttt{bool} \mid \texttt{nat} \mid \texttt{unit} \mid \texttt{empty} \mid A \to \underline{C} \mid \underline{B} \twoheadrightarrow \underline{C}$
Tipo computación	$\underline{C}$	$::=$	$C\langle\mathcal{E}\rangle$
Fila de efectos	$\mathcal{E}$	$::=$	$\langle \texttt{Op} \mid \mathcal{E}\rangle \mid \mu$
Esquema de tipo	$\mathcal{S}$	$::=$	$\forall\overline{\mu}.A$

Los tipos, análogamente a los términos, se dividen en dos clases. La primera es la de tipos valor, que está compuesta por los tipos base, funciones y handlers. La segunda es la de tipos computación: un tipo de la forma $C\langle\mathcal{E}\rangle$ representa la computaciones que pueden invocar las operaciones en $\mathcal{E}$ y al terminar devuelven un valor de tipo C.

En el tipo $C\langle\mathcal{E}\rangle$, $\langle\mathcal{E}\rangle$ es una fila de efectos de la forma $\langle \texttt{Op}_1 \mid \langle \texttt{Op}_2 \mid \langle \ldots \mid \mu\rangle\rangle\rangle$, que usualmente se notará $\langle \texttt{Op}_1,\ \texttt{Op}_2, \ldots \mid \mu\rangle$. En esta fila de efectos, μ es una variable de efectos que representa los efectos de ambiente. Para ilustrar la expresividad que brindan estas variables, se consideran las siguientes funciones de alto orden:

$$\begin{aligned} \textit{ignore} &::= \texttt{fun}\ f \mapsto \texttt{val}\ (\,) \\ \textit{apply} &::= \texttt{fun}\ f \mapsto f\ (\,) \end{aligned}$$

A la primera se le puede asignar el tipo $(\texttt{unit} \to \texttt{unit}\langle\mu_1\rangle) \to \texttt{unit}\langle\mu_2\rangle$, en el que los efectos de la función de entrada son independientes a los que pueden ser invocados por *ignore*. Sin embargo, como *apply* aplica la función tomada, los efectos que puede tener son los mismos que los que puede llamar f. En consecuencia, el tipo correspondiente a la segunda es $(\texttt{unit} \to \texttt{unit}\langle\mu_1\rangle) \to \texttt{unit}\langle\mu_1\rangle$.

Toda signatura de tipo implícitamente cuantifica universalmente sus variables de efectos. Por esta razón, aun si dos tipos anotados para distintos sub-términos de un programa usan el mismo nombre de variable de efectos, estas se consideran distintas. Los esquemas de tipo, que representan tipos polimórficos en efectos, permiten los let polimórficos de *Alef*. El siguiente ejemplo ayuda a entender la relevancia de estos.

$\texttt{let}\ pure \coloneqq\ \texttt{val}\ (\texttt{fun}\ m \mapsto \texttt{val}\ (\,) : \texttt{nat} \to \texttt{unit}\langle\mu\rangle)\ \texttt{in}$

$\texttt{let}\ impure \coloneqq\ \texttt{val}\ (\texttt{fun}\ m \mapsto \texttt{Print}(m) : \texttt{nat} \to \texttt{unit}\langle\texttt{Print} \mid \mu\rangle)\ \texttt{in}$

$\texttt{let}\ f \coloneqq\ \texttt{if}\ \texttt{true}\ \texttt{then}\ (\texttt{val}\ pure)\ \texttt{else}\ (\texttt{val}\ impure)\ \texttt{in}$

$\texttt{val}\ pure$

Este ejemplo fue utilizado por Bauer y Pretnar para explicar el problema de *poisoning* [2], que Eff soluciona mediante subtipado. Este problema surge al promover el tipo de *pure*, para que coincida con el de *impure* al tipar el condicional. Si este tipo más grueso es asignado a *pure* en el resto del programa, el tipo del programa completo sería $(\texttt{nat} \to \texttt{unit}\langle\texttt{Print} \mid \mu\rangle)\langle\mu'\rangle$, perdiendo la información valiosa de que este retorna una función pura. Sin embargo, como los let de *Alef* son polimórficos en efectos, a *pure* se le asigna el esquema de tipo $\forall\mu.\texttt{nat} \to \texttt{unit}\langle\mu\rangle$. Este esquema será instanciado en cada uso de esta función, posibilitando que la primera instancia se promueva al tipo de *impure*, mientras que la segunda permanece pura. Entonces, es posible darle al programa completo el tipo deseado de $(\texttt{nat} \to \texttt{unit}\langle\mu\rangle)\langle\mu'\rangle$.

En las filas no importa el orden de las operaciones y no se permite su repetición. Teniendo en cuenta estas consideraciones, una fila puede representarse con la forma $\langle\Delta \mid \mu\rangle$, donde $\Delta = \{\texttt{Op}_i\}_i$ es el conjunto de operaciones de la fila.

En *Alef* todas las filas son abiertas, es decir, los tipos describen efectos que pueden ocurrir, pero no prohíben efectos. De esta manera, un tipo computación informa sobre los efectos que una computación puede agregar a los de ambiente.

El tipo de un handler debe tener la forma $C\langle\Delta_1 \mid \mu\rangle \twoheadrightarrow D\langle\Delta_2 \mid \mu\rangle$, donde Δ_1 debe contener al conjunto de las operaciones manejadas por este handler. La interpretación de este tipo es que corresponde a un handler que transforma computaciones que devuelven valores en C a computaciones que devuelven valores en D, manejando algunas operaciones de Δ_1 y posiblemente agregando otras en Δ_2. Toda operación de Δ_1 no manejada por el handler debe estar en Δ_2. La variable de efectos μ indica que si el tipo de la computación a la que se aplica el handler tiene operaciones no contenidas en Δ_1, estas también estarán en el tipo de la computación resultante.

Se supone que existe una signatura de efectos $\Sigma = \{\texttt{Op}_i : A_i \to B_i\}_i$ que le asigna a cada operación un tipo de entrada y otro de resultado, ambos tipos básicos.

Sustituciones

Se formaliza el concepto de sustitución, que será útil para representar instanciaciones de variables de efectos. Una sustitución es:

$$\sigma ::= [\,] \mid [\mu \mapsto \langle\Delta \mid \mu'\rangle, \sigma]$$

Se nota $\mathcal{T}[\sigma]$ a la aplicación usual de la sustitución σ sobre $\mathcal{T}$ y $\sigma_1\sigma_2$ a la composición de dos sustituciones $(\mathcal{T}[\sigma_1\sigma_2] = (\mathcal{T}[\sigma_1])[\sigma_2])$.

Además, se dice que una sustitución σ es un *renombre inyectivo*, y se nota $ri(\sigma)$, si tiene la forma $\sigma = [\mu_i \mapsto \mu'_i]$ y vale que:

$$\forall \mu_i, \mu_j \in \text{dom}(\sigma) : \mu_i \neq \mu_j \implies \mu'_i \neq \mu'_j$$

Este predicado representa una condición suficiente para asegurar que al aplicar la sustitución a un tipo, el nuevo tipo es equivalente al original, de acuerdo a la relación de equivalencia presentada anteriormente.

Orden y equivalencia de tipos

A continuación se define una noción de orden sobre los tipos, que será de utilidad en las siguientes secciones. Para ello se utilizan sustituciones de variables de efecto por filas de efectos, usualmente denotadas con las letras σ, η y γ. Dados dos tipos $\mathcal{T}_1$ y $\mathcal{T}_2$, se define la relación de orden como:

$$\mathcal{T}_1 \sqsubseteq \mathcal{T}_2 ::= \exists \sigma \, (\mathcal{T}_2 = \mathcal{T}_1[\sigma])$$

La interpretación de esta relación es que si $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$, $\mathcal{T}_1$ es menos concreto que $\mathcal{T}_2$. En base a este orden, se define la siguiente noción de equivalencia.

$$\mathcal{T}_1 \equiv \mathcal{T}_2 ::= \mathcal{T}_1 \sqsubseteq \mathcal{T}_2 \wedge \mathcal{T}_2 \sqsubseteq \mathcal{T}_1$$

Ambas relaciones pueden extenderse de manera trivial a los esquemas de tipo.

Contextos de tipado

Los contextos de tipado, notados Γ, distinguen entre variables monomórficas y polimórficas en efectos, de manera similar a los de Frank [19]:

$$\Gamma ::= \cdot \mid \Gamma, \; x : A \mid \Gamma, \; f : \mathcal{S}$$

No se permiten repeticiones de nombres de variables en el contexto. Adicionalmente, las relaciones de orden y equivalencia de los tipos puede extenderse a los contextos de la siguiente manera:

$$\Gamma_1 \sqsubseteq \Gamma_2 ::= \text{dom}(\Gamma_1) = \text{dom}(\Gamma_2) \wedge \exists \sigma \, (\Gamma_2 = \Gamma_1[\sigma])$$
$$\Gamma_1 \equiv \Gamma_2 ::= \Gamma_1 \sqsubseteq \Gamma_2 \wedge \Gamma_2 \sqsubseteq \Gamma_1$$

Esto quiere decir que dos contextos son equivalentes si tienen las mismas variables y todos los tipos que los componen son equivalentes simultáneamente.

3.3. Sistema declarativo

A continuación se introduce un sistema de tipos y efectos declarativo junto a su semántica denotacional y operacional para un subconjunto de los términos. Este sistema no está pensado teniendo en cuenta su implementación, sino que representa una abstracción útil para simplificar el razonamiento sobre el tipado. En particular, sobre este sistema declarativo se enuncia y se prueba la seguridad del lenguaje.

En esta sección se ignoran ciertas construcciones del lenguaje que no aportan diferencias fundamentales en el tipado y, por lo tanto, permiten reducir el tamaño de las pruebas sin perder generalidad. Estas son las contantes de tipo `bool` y `nat` junto a sus destructores.

Reglas de tipado

En esta sección se desarrollará el juicio de tipado $\Gamma \vdash t : \mathcal{T}$, que afirma que el término t tiene tipo $\mathcal{T}$ en el contexto Γ. Este juicio es, más formalmente, una generalización de dos juicios de tipado, uno para tipos valor y otro para tipos computación, que comparten la notación. En adelante, se usará $\mathcal{T}$ para notar tanto tipos valor como tipos computación.

A continuación, se describen las reglas de tipado definidas en la Figura 3.1:

Var$^{\mathbf{D}}$ Una variable monomórifica tiene el tipo que le asigna el entorno.

PVar$^{\mathbf{D}}$ Un esquema de tipo puede ser instanciado con una sustitución arbitraria θ de sus variables de efecto universalmente cuantificadas.

Val$^{\mathbf{D}}$ Las computaciones base pueden ser tipadas con cualquier fila de efectos. Esto permite agregar los efectos necesarios para hacer compatibles los tipos en las otras reglas.

Unit$^{\mathbf{D}}$ El valor ( ) tiene tipo `unit`.

App$^{\mathbf{D}}$ Para tipar la aplicación de e_1 a e_2, e_1 debe ser una función y e_2 debe ser del tipo del dominio de e_1.

Fun$^{\mathbf{D}}$ Una abstracción tiene el tipo $A \to \underline{C}$ si el cuerpo de la función puede ser tipado con $\underline{C}$ al extender el entorno con la variable abstraída.

Op$^{\mathbf{D}}$ Para tipar la invocación a una operación, primero se verifica que la expresión argumento verifique el tipo de entrada de dicha operación. En ese caso, esta invocación tiene el tipo de la continuación, asegurándose que este último cuente con dicha operación en su fila de efectos.

Hand$^{\mathbf{D}}$ Se requieren dos condiciones a las hipótesis. La primera es que las operaciones manejadas por un handler sean distintas entre sí, garantizando que su aplicación sea determinista. La segunda es que todas las operaciones presentes en el tipo de la computación de entrada que no son manejadas por el handler, también formen parte del tipo de salida.

With$^{\mathbf{D}}$ Para tipar la aplicación de un handler h a una computación c, se verifica que h tenga tipo handler y que c sea del tipo del dominio de h.

Anno$^{\mathbf{D}}$ El tipo de un término anotado es el de su anotación.

Let$^{\text{D}}$ En el tipado de una expresión `let` se pide que ambas computaciones tengan los mismos efectos en sus filas. Además, en el tipado de la segunda computación, se extiende el entorno reflejando que la variable declarada en esta construcción es polimórfica. Para ello, se cuantifican universalmente todas las variables de efectos de A que no ocurran en Γ, evitando capturar variables ya ligadas.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}^{\text{D}} \qquad \frac{f : \forall\overline{\mu}.A \in \Gamma}{\Gamma \vdash f : A[\theta]}\ \textsc{PVar}^{\text{D}} \qquad \frac{}{\Gamma \vdash (\,) : \texttt{unit}}\ \textsc{Unit}^{\text{D}}$$

$$\frac{\Gamma \vdash e_1 : A \to \underline{C} \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : \underline{C}}\ \textsc{App}^{\text{D}} \qquad \frac{\Gamma,\ x : A \vdash c : \underline{C}}{\Gamma \vdash \texttt{fun}\ x \mapsto c : A \to \underline{C}}\ \textsc{Fun}^{\text{D}}$$

$$\frac{\texttt{Op} : A_{op} \to B_{op} \in \Sigma \qquad \Gamma \vdash e : A_{op} \qquad \Gamma,\ y : B_{op} \vdash c : A\langle\mathcal{E}\rangle \qquad \texttt{Op} \in \mathcal{E}}{\Gamma \vdash \texttt{Op}\ e\ (y.c) : A\langle\mathcal{E}\rangle}\ \textsc{Op}^{\text{D}}$$

$$\frac{\begin{array}{lll}\{\texttt{Op}_i \neq \texttt{Op}_j\}_{i \neq j} & \Delta \setminus \{\texttt{Op}_i\}_{i=1}^{n} \subseteq \Delta' & \Gamma,\ x : A \vdash c_v : B\langle\Delta' \mid \mu\rangle \\ \{\texttt{Op}_i : A_i \to B_i \in \Sigma & \Gamma,\ x : A_i,\ k : B_i \to B\langle\Delta' \mid \mu\rangle \vdash c_i : B\langle\Delta' \mid \mu\rangle\}_{i=1}^{n} & \end{array}}{\Gamma \vdash \texttt{handler}\ \texttt{val}\ x \mapsto c_v,\ \{\texttt{Op}_i\ x\ k \mapsto c_i\}_{i=1}^{n} : A\langle\Delta \mid \mu\rangle \twoheadrightarrow B\langle\Delta' \mid \mu\rangle}\ \textsc{Hand}^{\text{D}}$$

$$\frac{\Gamma \vdash h : A\langle\mathcal{E}\rangle \twoheadrightarrow B\langle\mathcal{E}'\rangle \qquad \Gamma \vdash c : A\langle\mathcal{E}\rangle}{\Gamma \vdash \texttt{with}\ h\ \texttt{handle}\ c : B\langle\mathcal{E}'\rangle}\ \textsc{With}^{\text{D}} \qquad \frac{\Gamma \vdash t : \mathcal{T}}{\Gamma \vdash (t : \mathcal{T}) : \mathcal{T}}\ \textsc{Anno}^{\text{D}}$$

$$\frac{\Gamma \vdash c_1 : A\langle\mathcal{E}\rangle \qquad \Gamma,\ x : \forall\overline{\mu}.A \vdash c_2 : B\langle\mathcal{E}\rangle}{\Gamma \vdash \texttt{let}\ x \coloneqq c_1\ \texttt{in}\ c_2 : B\langle\mathcal{E}\rangle}\ \textsc{Let}^{\text{D}} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \texttt{val}\ e : A\langle\mathcal{E}\rangle}\ \textsc{Val}^{\text{D}}$$

Figura 3.1: Reglas de tipado declarativas.

Semántica denotacional

Para mostrar la estrecha relación entre *Alef* y la teoría de efectos algebraicos se presenta una semántica denotacional que modela este lenguaje haciendo uso de los conceptos matemáticos introducidos en la Sección 2.2.

Denotación de tipos

Puesto que *Alef* no cuenta con recursión ni funciones parciales, los tipos de *Alef* se modelan mediante conjuntos. Dado un tipo $\mathcal{T}$, su denotación se escribe $[\![\mathcal{T}]\!]$ y sigue la siguiente definición:

$$\begin{aligned} [\![\texttt{unit}]\!] &::= \{(\,)\} \\ [\![\texttt{empty}]\!] &::= \emptyset \\ [\![A \to \underline{C}]\!] &::= [\![A]\!] \to [\![\underline{C}]\!] \end{aligned}$$

Es decir, la denotación de un tipo función es el conjunto de funciones entre las denotaciones de su dominio y codominio.

$$[\![\underline{B} \twoheadrightarrow \underline{C}]\!] \ ::= [\![\underline{B}]\!] \twoheadrightarrow [\![\underline{C}]\!]$$

Para el tipo handler, su denotación son los T-homomorfismos entre las denotaciones de su dominio y codominio, siguiendo la presentación del modelado de handlers de la Sección 2.2.

$$[\![A\langle\Delta \mid \mu\rangle]\!] ::= [\![A]\!]\langle\Delta\rangle$$

Por último, la denotación de un tipo computación corresponde al modelo libre sobre la teoría sin ecuaciones $(\Delta, \emptyset)$ generado por $[\![A]\!]$. Esto quiere decir que $[\![A]\!]\langle\Delta\rangle$ es el conjunto de árboles de computación cuyos elementos tienen la forma:

- **val** e para algún $e \in [\![A]\!]$.
- **Op**$(e,\ \kappa)$ para algún $e \in [\![A_{op}]\!]$ y $\kappa : [\![B_{op}]\!] \rightarrow [\![A]\!]\langle\Delta\rangle$.

Se utiliza notación en negrita para diferenciar estos últimos de las construcciones de *Alef*.

Adicionalmente, es necesario introducir la denotación para los esquemas de tipo. Esta corresponde a la intersección de las denotaciones de las posibles instanciaciones de sus variables de efectos cuantificadas:

$$[\![\forall\overline{\mu}.A]\!] ::= \bigcap_{A \sqsubseteq A'} [\![A']\!]$$

Denotación de términos

Como en este modelo se interpretan términos tipados, es necesario contar con la denotación de los contextos de tipado:

$$[\![\Gamma]\!] ::= \prod_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]$$

En palabras, un contexto se denota mediante el producto cartesiano de las denotaciones de los tipos que lo forman. Entonces, un elemento de $[\![\Gamma]\!]$ es una función ρ tal que para cada $x \in \mathrm{dom}(\Gamma)$, $\rho(x) \in [\![\Gamma(x)]\!]$.

Con esto, los juicios de tipado $\Gamma \vdash t : \mathcal{T}$ reciben la interpretación:

$$[\![\Gamma \vdash t : \mathcal{T}]\!] ::= [\![\Gamma]\!] \rightarrow [\![\mathcal{T}]\!]$$

Es decir, un término tipado es denotado por una función que dado un contexto ρ devuelve un elemento de la denotación del tipo correspondiente. Estas interpretaciones de términos tipados se definen como:

$$[\![\Gamma \vdash x : A]\!](\rho) \qquad ::= \rho(x) \qquad (\text{Si } x : A \in \Gamma)$$

$$
\begin{array}{lll}
[\![\Gamma \vdash f : A[\theta]]\!](\rho) & ::= \rho(f) & (\text{Si } f : \forall\overline{\mu}.A \in \Gamma) \\
[\![\Gamma \vdash \texttt{val}\ e : A\langle\mathcal{E}\rangle]\!](\rho) & ::= \textbf{val}\ ([\![\Gamma \vdash e : A]\!]\rho) & \\
[\![\Gamma \vdash (\,) : \texttt{unit}]\!](\rho) & ::= (\,) & \\
[\![\Gamma \vdash e_1\ e_2 : \underline{C}]\!](\rho) & ::= ([\![\Gamma \vdash e_1 : A \to \underline{C}]\!]\rho)\ ([\![\Gamma \vdash e_2 : A]\!]\rho) & \\
[\![\Gamma \vdash (t : \mathcal{T}) : \mathcal{T}]\!](\rho) & ::= [\![\Gamma \vdash t : \mathcal{T}]\!]\rho & \\
[\![\Gamma \vdash \texttt{fun}\ x \mapsto a : A \to \underline{C}]\!](\rho) & ::= \lambda a \in [\![A]\!].\ [\![\Gamma,\ x : A \vdash c : \underline{C}]\!](\rho \oplus x \mapsto a) &
\end{array}
$$

Donde $(\rho \oplus x \mapsto a)$ corresponde a la extensión funcional de ρ que le asigna a a x.

$$
[\![\Gamma \vdash \texttt{Op}\ e\ (y.c) : A\langle\mathcal{E}\rangle]\!](\rho) \quad ::=
$$

$$
\textbf{Op}([\![\Gamma \vdash e : A_{op}]\!]\rho,\ \lambda b \in [\![B_{op}]\!].\ [\![\Gamma,\ y : B_{op} \vdash c : A\langle\mathcal{E}\rangle]\!](\rho \oplus y \mapsto b))
$$

Para la siguiente denotación, se nota con h a `handler` `val` $x \mapsto c_v,\ \{\texttt{Op}_i\ x\ k \mapsto c_i\}_{i=1}^{n}$.

$$
[\![\Gamma \vdash h : A\langle\mathcal{E}\rangle \twoheadrightarrow B\langle\mathcal{E}'\rangle]\!](\rho) \quad ::= H
$$

Donde $H : [\![\Gamma]\!] \to [\![A\langle\mathcal{E}\rangle]\!] \twoheadrightarrow [\![B\langle\mathcal{E}'\rangle]\!]$ se define recursivamente como:

$$
\begin{array}{l}
H(\rho)(\textbf{val}\ e) ::= [\![\Gamma,\ x : A \vdash c_v : B\langle\mathcal{E}'\rangle]\!](\rho \oplus x \mapsto e) \\
H(\rho)(\textbf{Op}(e,\ \kappa)) ::= [\![\Gamma,\ x : A_i,\ k : B_i \to B\langle\mathcal{E}'\rangle \vdash c_i : B\langle\mathcal{E}'\rangle]\!](\rho \oplus x \mapsto e \oplus k \mapsto H(\rho) \circ \kappa)
\end{array}
$$

$$
[\![\Gamma \vdash \texttt{with}\ h\ \texttt{handle}\ c : B\langle\mathcal{E}'\rangle]\!](\rho) ::= ([\![\Gamma \vdash h : A\langle\mathcal{E}\rangle \twoheadrightarrow B\langle\mathcal{E}'\rangle]\!]\rho)\ ([\![\Gamma \vdash c : A\langle\mathcal{E}\rangle]\!]\rho)
$$

Previo a la denotación del `let`, se define el *lifting* de una función $f : [\![A]\!] \to [\![B\langle\mathcal{E}\rangle]\!]$ como la función $f^\dagger : [\![A\langle\mathcal{E}\rangle]\!] \to [\![B\langle\mathcal{E}\rangle]\!]$ dada recursivamente por:

$$
\begin{array}{ll}
f^\dagger(\textbf{val}\ e) & ::= f(e) \\
f^\dagger(\textbf{Op}(e,\ \kappa)) & ::= \textbf{Op}(e,\ f^\dagger \circ \kappa)
\end{array}
$$

$$
[\![\Gamma \vdash \texttt{let}\ x \coloneqq\ c_1\ \texttt{in}\ c_2 : B\langle\mathcal{E}\rangle]\!](\rho) ::=
$$

$$
(\lambda a \in [\![A]\!].\ [\![\Gamma,\ x : \forall\overline{\mu}.A \vdash c_2 : B\langle\mathcal{E}\rangle]\!](\rho \oplus x \mapsto a))^\dagger\ ([\![\Gamma \vdash c_1 : A\langle\mathcal{E}\rangle]\!]\rho)
$$

Semántica operacional

La semántica operacional small-step se define en base a la relación $c \rightsquigarrow c'$, que significa que la computación c evalúa en un paso a la computación c'. Esta se define en la Figura 3.2.

Las reglas de mayor interés son aquellas que corresponden a la aplicación de un handler. La primera de estas, E-With-Val, se ocupa del caso en el que la computación manejada es una expresión pura, de la forma `val` e. En este caso, esta computación

es manejada con la cláusula correspondiente a valores, instanciando la variable declarada a la expresión pasada.

La segunda, E-WITH-HANDLE, corresponde al caso de la invocación de una de las operaciones manejadas por el handler. Esta computación evalúa a la cláusula que dicha operación tiene asociada en la definición del handler, instanciando por un lado la variable del argumento, y por otro lado la variable de la continuación, que pasa a ser la aplicación del mismo handler sobre la continuación provista por el llamado a la operación.

El último caso, además de la regla de congruencia, corresponde a la regla E-WITH-IGN que se ocupa de manejar el llamado a una operación no descripta en el handler. En este caso, se evalúa llamando a esta operación, pero aplicando el mismo handler sobre su continuación.

Estas últimas dos reglas muestran que los handlers utilizados en *Alef* corresponden a *handlers profundos*, ya que sus aplicaciones se propagan dentro de los árboles de computación.

Las últimas reglas —aquellas cuyos nombres terminan en A— corresponden a los casos en los que ciertas anotaciones de tipo deben ser propagadas en la evaluación. Esto es necesario cuando se sustituye una variable por una expresión de la cual se conoce el tipo por una anotación existente. Sin embargo, solo se anotan funciones y handlers, puesto que no tiene sentido anotar el resto de las expresiones del lenguaje (constantes y variables). Para tener en cuenta esto último, se define la siguiente notación:

$$\underset{\sim}{e : A} ::= \begin{cases} e : A & \text{si } e \text{ es una función o handler,} \\ e & \text{en otro caso.} \end{cases}$$

Seguridad

La seguridad de la relación de evaluación respecto al sistema de tipos se enuncia en base a dos lemas. El primero, Preservación, establece que si un término tipado puede dar un paso de evaluación, al resultado de esta última también se le puede asignar el mismo tipo.

Lema 1 (Preservación)**.** *Para todo* Γ*,* $\mathcal{T}$*,* c *y* c' *tales que* $\Gamma \vdash t : \mathcal{T}$ *y* $t \rightsquigarrow t'$*, vale que* $\Gamma \vdash t' : \mathcal{T}$*.*

El segundo lema corresponde al Progreso, y garantiza que todo término tipable es un terminal o puede ser evaluado.

Lema 2 (Progreso)**.** *Para toda computación cerrada* c *y todo tipo* $C\langle\mathcal{E}\rangle$ *tales que* $\cdot \vdash c : C\langle\mathcal{E}\rangle$*, vale una de las siguientes:*

- $\exists v\, (c = \texttt{val}\ v \wedge \cdot \vdash v : C)$.
- $\exists \texttt{Op}_j, e, k\, (c = \texttt{Op}_j\ e\ k \wedge \texttt{Op}_j \in \mathcal{E})$.
- $\exists c'\, (c \rightsquigarrow c')$

En estas reglas se nota con h la construcción $\texttt{handler val } x \mapsto c_v,\ \{\texttt{Op}_i\ x\ k \mapsto c_i\}_{i=1}^n$ y con H al tipo $\underline{C} \twoheadrightarrow \underline{D}$.

$$(\texttt{fun}\ x \mapsto c)\ e \rightsquigarrow c[x \mapsto e] \qquad \text{(E-APP)}$$

$$\texttt{with}\ h\ \texttt{handle}\ (\texttt{val}\ e) \rightsquigarrow c_v[x \mapsto e] \qquad \text{(E-WITH-VAL)}$$

$$\frac{\texttt{Op}_j \in \{\texttt{Op}_i\}_{i=1}^n}{\texttt{with}\ h\ \texttt{handle}\ (\texttt{Op}_j\ e\ y.c) \rightsquigarrow c_j[x \mapsto e,\ k \mapsto (\texttt{fun}\ y \mapsto \texttt{with}\ h\ \texttt{handle}\ c)]} \qquad \text{(E-WITH-HANDLE)}$$

$$\frac{\texttt{Op}_j \notin \{\texttt{Op}_i\}_{i=1}^n}{\texttt{with}\ h\ \texttt{handle}\ (\texttt{Op}_j\ e\ y.c) \rightsquigarrow \texttt{Op}_j\ e\ y.(\texttt{with}\ h\ \texttt{handle}\ c)} \qquad \text{(E-WITH-IGN)}$$

$$\frac{c \rightsquigarrow c'}{\texttt{with}\ e\ \texttt{handle}\ c \rightsquigarrow \texttt{with}\ e\ \texttt{handle}\ c'} \qquad \text{(E-WITH-CONGR)}$$

$$\texttt{let}\ x \coloneqq (\texttt{val}\ e)\ \texttt{in}\ c_2 \rightsquigarrow c_2[x \mapsto e] \qquad \text{(E-LET-VAL)}$$

$$\texttt{let}\ x \coloneqq \texttt{Op}\ e\ y.c_1\ \texttt{in}\ c_2 \rightsquigarrow \texttt{Op}\ e\ y.(\texttt{let}\ x \coloneqq c_1\ \texttt{in}\ c_2) \qquad \text{(E-LET-OP)}$$

$$\frac{c_1 \rightsquigarrow c_1'}{\texttt{let}\ x \coloneqq c_1\ \texttt{in}\ c_2 \rightsquigarrow \texttt{let}\ x \coloneqq c_1'\ \texttt{in}\ c_2} \qquad \text{(E-LET-CONGR)}$$

$$(\texttt{fun}\ x \mapsto c : A \to \underline{C})\ e \rightsquigarrow c[x \mapsto \utilde{e : A}] \qquad \text{(E-APP-A)}$$

$$\texttt{with}\ h : H\ \texttt{handle}\ (\texttt{val}\ e) \rightsquigarrow c_v[x \mapsto \utilde{e : C}] \qquad \text{(E-WITH-VAL-A)}$$

$$\frac{\texttt{Op}_j \in \{\texttt{Op}_i\}_{i=1}^n}{\texttt{with}\ h : H\ \texttt{handle}\ (\texttt{Op}_j\ e\ y.c) \rightsquigarrow c_j[x \mapsto \utilde{e : C},\ k \mapsto (\texttt{fun}\ y \mapsto \texttt{with}\ h : H\ \texttt{handle}\ c)]} \qquad \text{(E-WITH-HANDLE-A)}$$

$$\frac{\texttt{Op}_j \notin \{\texttt{Op}_i\}_{i=1}^n}{\texttt{with}\ h : H\ \texttt{handle}\ (\texttt{Op}_j\ e\ y.c) \rightsquigarrow \texttt{Op}_j\ e\ y.(\texttt{with}\ h : H\ \texttt{handle}\ c)} \qquad \text{(E-WITH-IGN-A)}$$

$$\texttt{let}\ x \coloneqq (\texttt{val}\ e) : \underline{A}\ \texttt{in}\ c_2 \rightsquigarrow c_2[x \mapsto \utilde{e : A}] \qquad \text{(E-LET-VAL-A)}$$

Figura 3.2: Relación de evaluación.

Las pruebas de los Lemas 1 y 2 se encuentran en el Capítulo 6. Juntando estos dos lemas, se alcanza el siguiente teorema de Seguridad.

Teorema 1 (Seguridad)**.** *Toda computación cerrada del tipo $C\langle\mathcal{E}\rangle$ es un valor de tipo C o la invocación de una operación contenida en $\mathcal{E}$ o puede ser evaluada a otra computación del tipo $C\langle\mathcal{E}\rangle$.*

Ejemplos

Como primer ejemplo se muestra el tipado de la aplicación de una función de alto orden. En particular, el término a tipar es *apply crash*, donde *apply* ::= $\texttt{fun}\ f \mapsto f\ (\,)$ y *crash* es una función que puede fallar.

Se divide el tipado en dos derivaciones, con el fin de simplificar su lectura. En primer lugar, se presenta en la Figura 3.3 el tipado de *apply*, para el cual se define el contexto Γ_1 como aquel con la asignación $f : \texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash f : \texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle}\ \textsc{Var}^{\mathrm{D}} \quad \dfrac{}{\Gamma_1 \vdash (\,) : \texttt{unit}}\ \textsc{Unit}^{\mathrm{D}}}{\Gamma_1 \vdash f\ (\,) : \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle}\ \textsc{App}^{\mathrm{D}}}{\cdot \vdash \texttt{fun}\ f \mapsto f\ (\,) : (\texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle) \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle}\ \textsc{Fun}^{\mathrm{D}}$$

Figura 3.3: Tipado de *apply*.

Luego, con Γ ::= *apply* : $(\texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle) \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle$, *crash* : $\texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle$, el tipado del término inicial, *apply crash*, se presenta el la figura 3.4.

El siguiente ejemplo consiste en la aplicación de un handler que cuenta el número de llamados a `Print` de una computación. Se supone que la signatura correspondiente a `Print` es `nat` → `unit`. El término *withcount*, que se tipará y evaluará, se define en la Figura 3.5,

Para el resto del ejemplo, se abreviará con *h* al handler del programa anterior. Similarmente al primer ejemplo, se separa la sub-derivación de *h* de la derivación de tipado principal, ambas presentes en la Figura 3.6.

Para concluir el ejemplo, se muestran en la Figura 3.7 los pasos que tomaría la evaluación de este programa.

$$\dfrac{\dfrac{}{\Gamma \vdash \mathit{apply} : (\texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle) \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle}\ \textsc{Var}^{\mathrm{D}} \quad \dfrac{}{\Gamma \vdash \mathit{crash} : \texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle}\ \textsc{Var}^{\mathrm{D}}}{\Gamma \vdash \mathit{apply\ crash} : \texttt{unit}\langle\texttt{Throw} \mid \mu\rangle}\ \textsc{App}^{\mathrm{D}}$$

Figura 3.4: Tipado del término *apply crash*.

$$
\begin{aligned}
\textit{withcount} ::= \ & \texttt{with}\ (\texttt{handler} \quad \texttt{val}\ x \mapsto \texttt{val}\ 0, \\
& \qquad\qquad\qquad \{\texttt{Print}\ x\ k \mapsto \texttt{let}\ y \coloneqq\ k\ (\,)\ \texttt{in}\ \texttt{val}\ (\texttt{succ}\ y)\}) \\
& \texttt{handle}\ \texttt{Print}(7)
\end{aligned}
$$

Figura 3.5: Definición del término *withcount*.

Sea T ::=

$$
\dfrac{\dfrac{\vdots}{x : \texttt{unit} \vdash \texttt{val}\ 0 : \texttt{nat}\langle\mu\rangle}\ \textsc{Val}^{\mathrm{D}} \qquad \dfrac{\dfrac{\vdots}{\ldots \vdash k\ (\,) : \texttt{nat}\langle\mu\rangle}\ \textsc{App}^{\mathrm{D}} \quad \dfrac{\vdots}{\ldots \vdash \texttt{val}\ (\texttt{succ}\ y) : \texttt{nat}\langle\mu\rangle}\ \textsc{Val}^{\mathrm{D}}}{x : \texttt{unit},\ k : \texttt{nat} \to \texttt{nat}\langle\mu\rangle \vdash \texttt{let}\ y \coloneqq\ k\ (\,)\ \texttt{in}\ \texttt{val}\ (\texttt{succ}\ y) : \texttt{nat}\langle\mu\rangle}\ \textsc{Let}^{\mathrm{D}}}{\cdot \vdash h : \texttt{unit}\langle \texttt{Print} \mid \mu\rangle \twoheadrightarrow \texttt{nat}\langle\mu\rangle}\ \textsc{Hand}^{\mathrm{D}}
$$

en

$$
\dfrac{\dfrac{T}{\cdot \vdash h : \texttt{unit}\langle \texttt{Print} \mid \mu\rangle \twoheadrightarrow \texttt{nat}\langle\mu\rangle}\ \textsc{Hand}^{\mathrm{D}} \qquad \dfrac{\dfrac{\vdots}{\cdot \vdash 7 : \texttt{nat}} \quad \dfrac{\vdots}{z : \texttt{unit} \vdash \texttt{val}\ z : \texttt{unit}\langle \texttt{Print} \mid \mu\rangle}\ \textsc{Val}^{\mathrm{D}}}{\cdot \vdash \texttt{Print}\ 7\ z.(\texttt{val}\ z) : \texttt{unit}\langle \texttt{Print} \mid \mu\rangle}\ \textsc{Op}^{\mathrm{D}}}{\cdot \vdash \texttt{with}\ h\ \texttt{handle}\ \texttt{Print}(7) : \texttt{nat}\langle\mu\rangle}\ \textsc{With}^{\mathrm{D}}
$$

Figura 3.6: Tipado de *withcount*.

$$
\begin{aligned}
& \texttt{with}\ h\ \texttt{handle}\ \texttt{Print}(7) \\
\rightsquigarrow\ & \texttt{let}\ y \coloneqq\ (\texttt{fun}\ z \mapsto \texttt{with}\ h\ \texttt{handle}\ (\texttt{val}\ z))\ (\,)\ \texttt{in}\ \texttt{val}\ (\texttt{succ}\ y) \\
\rightsquigarrow\ & \texttt{let}\ y \coloneqq\ \texttt{with}\ h\ \texttt{handle}\ \texttt{val}\ (\,)\ \texttt{in}\ \texttt{val}\ (\texttt{succ}\ y) \\
\rightsquigarrow\ & \texttt{let}\ y \coloneqq\ \texttt{val}\ 0\ \texttt{in}\ \texttt{val}\ (\texttt{succ}\ y) \\
\rightsquigarrow\ & \texttt{val}\ (\texttt{succ}\ 0)
\end{aligned}
$$

Figura 3.7: Evaluación del término *withcount*.

3.4. Sistema bidireccional

Si bien la claridad conceptual del sistema declarativo presentado en las secciones anteriores simplifica el razonamiento sobre el mismo, el nivel de abstracción en el que se define dificulta la tarea de implementación. En particular, la dificultad radica en que las reglas de este sistema declarativo asumen que los tipos en el contexto ya cuentan con todos los efectos necesarios para ser compatibles, sin que exista forma de promoverlos. Esto es posible gracias a la regla $\textsc{Val}^{\mathrm{D}}$, en la que los efectos asociados

a la computación son independientes de las hipótesis. Similarmente, las instancias de variables polimórficas generadas por $\text{PV}\textsc{ar}^{\text{D}}$ no determinan la sustitución aplicada. Estas particularidades hacen que el sistema declarativo no sea dirigido por sintaxis, lo que es un problema para su implementación.

Para expresar esto de una manera dirigida por sintaxis, en esta sección se presenta un sistema de tipado bidireccional con polimorfismo de efectos basado en la instanciación de variables de efectos.

Juicios

Al desarrollarse un sistema de tipado bidireccional, es necesario definir dos juicios de tipado, uno para síntesis y otro para chequeo. Estos son:

- $\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}$: Dados Γ y t, se puede sintetizar el tipo $\mathcal{T}$ para t en el contexto Γ, produciendo las instanciaciones de variables de efectos η.
- $\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T}$: Dados Γ, t y $\mathcal{T}$, t chequea el tipo $\mathcal{T}$ en Γ con la instanciación σ de sus variables de efectos. Esta instanciación es una sustitución de variables de efectos por filas, y representa la captura de los efectos de ambiente.

En base a este último juicio se puede definir un juicio de chequeo más convencional, que determina si un término tiene exactamente un tipo dado:

$$\Gamma \vdash t \Leftarrow \mathcal{T} ::= (\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T}) \wedge \mathcal{T} \equiv \mathcal{T}[\sigma]$$

De esta manera, $\Gamma \vdash t \Leftarrow \mathcal{T}$ vale si t chequea el tipo $\mathcal{T}$ en el contexto Γ con una instanciación que, una vez aplicada al tipo $\mathcal{T}$, produce un tipo α-equivalente a este. Esto garantiza que la sustitución correspondiente a esta instanciación no aporte información adicional sobre el tipo $\mathcal{T}$.

Instanciación de variables de efectos

En el tipado bidireccional suele ser necesario comparar un tipo que está siendo chequeado con uno sintetizado para el mismo término. Para estas comparaciones se dan dos relaciones de instanciación de variables de efectos, una para filas de efectos y otra para tipos.

La primera —$\mathcal{E}_1 \overset{\sigma}{\bowtie}_R \mathcal{E}_2$, definida en la Figura 3.8— toma dos filas de efectos ($\mathcal{E}_1$ y $\mathcal{E}_2$) y devuelve la sustitución σ que captura las instanciaciones necesarias para unificar estas dos filas en una nueva, que contiene todas las operaciones de ambas. Para esta definición se nota con $\Delta_1 \setminus \Delta_2$ a la diferencia de conjuntos.

Similarmente, la segunda relación, $\mathcal{T}_1 \overset{\sigma}{\bowtie}_T \mathcal{T}_2$, toma dos tipos ($\mathcal{T}_1$ y $\mathcal{T}_2$) y devuelve la sustitución σ que contiene las instanciaciones de variables de efectos necesarias para unificar las filas de efectos presentes en estos dos tipos. Para su definición, presente en la Figura 3.9, se simboliza con BT a los tipos base, es decir, `bool`, `unit`, `nat` y `empty`.

$$\frac{\mathcal{E}_1 = \mathcal{E}_2}{\mathcal{E}_1 \overset{[]}{\bowtie_R} \mathcal{E}_2} \qquad \frac{\langle \Delta_1 \mid \mu_1 \rangle \neq \langle \Delta_2 \mid \mu_2 \rangle}{\langle \Delta_1 \mid \mu_1 \rangle \overset{[\mu_1 \mapsto \Delta_2 \backslash \Delta_1 \mid \mu',\ \mu_2 \mapsto \Delta_1 \backslash \Delta_2 \mid \mu']}{\bowtie_R} \langle \Delta_2 \mid \mu_2 \rangle}$$

Figura 3.8: Relación de instanciación de variables de efectos para filas.

$$\frac{A \overset{\sigma_1}{\bowtie_T} A' \qquad C[\sigma_1] \overset{\sigma_2}{\bowtie_T} C'[\sigma_1] \qquad \mathcal{E}[\sigma_1\sigma_2] \overset{\sigma_3}{\bowtie_R} \mathcal{E}'[\sigma_1\sigma_2]}{A \to C\langle \mathcal{E} \rangle \overset{\sigma_1\sigma_2\sigma_3}{\bowtie_T} A' \to C'\langle \mathcal{E}' \rangle}$$

$$\frac{}{BT \overset{[]}{\bowtie_T} BT} \qquad \frac{C \overset{\sigma_1}{\bowtie_T} C' \qquad \mathcal{E}[\sigma_1] \overset{\sigma_2}{\bowtie_R} \mathcal{E}'[\sigma_1]}{C\langle \mathcal{E} \rangle \overset{\sigma_1\sigma_2}{\bowtie_T} C'\langle \mathcal{E}' \rangle}$$

$$\frac{\begin{array}{ll} C \overset{\sigma_1}{\bowtie_T} C' & D[\sigma_1] \overset{\sigma_2}{\bowtie_T} D'[\sigma_1] \\ \mathcal{E}_1[\sigma_1\sigma_2] \overset{\sigma_3}{\bowtie_R} \mathcal{E}'_1[\sigma_1\sigma_2] & \mathcal{E}_2[\sigma_1\sigma_2\sigma_3] \overset{\sigma_4}{\bowtie_R} \mathcal{E}'_2[\sigma_1\sigma_2\sigma_3] \end{array}}{C\langle \mathcal{E}_1 \rangle \twoheadrightarrow D\langle \mathcal{E}_2 \rangle \overset{\sigma_1\sigma_2\sigma_3\sigma_4}{\bowtie_T} C'\langle \mathcal{E}'_1 \rangle \twoheadrightarrow D'\langle \mathcal{E}'_2 \rangle}$$

Figura 3.9: Relación de instanciación de variables de efectos para tipos.

De esta forma, la segunda relación permite propagar las instanciaciones de variables de efecto sobre los tipos, chequeando además que ambos coincidan estructuralmente.

Reglas de tipado

Para auxiliar la definición de las reglas de tipado, se introduce una función $add(\mathcal{E}, \texttt{Op}) = \gamma$. Esta toma una fila $\mathcal{E}$ y una operación $\texttt{Op}$ y devuelve la sustitución necesaria (γ) para agregar $\texttt{Op}$ a $\mathcal{E}$. Su definición es:

$$add(\langle \Delta \mid \mu \rangle, \texttt{Op}) = \begin{cases} [] & \texttt{Op} \in \Delta \\ [\mu \mapsto \langle \texttt{Op} \mid \mu' \rangle] & \texttt{Op} \notin \Delta \end{cases}$$

Adicionalmente, la función *ren* toma un conjunto de variables de efecto y retorna una instanciación que sustituye cada una de ellas por una nueva variable fresca.

En la Figura 3.10 se definen las reglas de tipado. Las reglas que corresponden al tipado de valores de tipos base son triviales. A continuación se presentan algunas aclaraciones sobre las restantes:

Var Para sintetizar el tipo de una variable monomórfica en efectos perteneciente al contexto se retorna el tipo correspondiente.

PVar Un esquema de tipo se instancia renombrando sus variables de efectos cuantificadas con variables frescas.

Val Una vez sintetizado el tipo de e, se retorna un tipo computación con una variable de efectos fresca.

CS Permite chequear un tipo que puede ser sintetizado, módulo instanciaciones. Esta regla se aplica únicamente a los términos que no cuentan con otra regla de chequeo, por lo que no se rompe la dirección por sintaxis del sistema.

Anno La anotación de tipos es necesaria para algunas construcciones cuyos tipos solo pueden ser chequeados. Se suponen frescas las variables de tipos de las anotaciones.

App Al sintetizarse el tipo de una aplicación de función, se debe aplicar la instanciación de variables de efectos capturadas en el chequeo. Las instanciaciones η se propagan al contexto para el chequeo.

Fun El tipo de una abstracción solamente puede ser chequeado.

If, Match Se chequea que el tipo de la expresión sea correcto. Luego, se sintetiza el tipo correspondiente a la primera rama y se lo utiliza para chequear el tipo de la segunda, propagando en el contexto las instanciaciones de η. El tipo que se sintetiza es el de la primera rama luego de aplicar las instanciaciones resultantes del chequeo.

Op Para la invocación de una operación, se chequea el tipo del parámetro y luego se sintetiza el de la continuación. Por último, se computa la instanciación necesaria para que la operación esté presente en la fila de efectos. El tipo de esta computación es el resultante de aplicar ésta al tipo sintetizado para la continuación.

Hand Las hipótesis de esta regla se dividen en cuatro partes. En primer lugar se verifica que no haya más de una cláusula para cada operación, para que la aplicación de un handler sea determinista. Luego se comprueba que las operaciones adicionales de Δ también se encuentren en Δ'. En tercer lugar se chequea el tipo de la computación correspondiente al caso de `val` , capturando instanciaciones σ_0. Finalmente, para cada $\mathtt{Op}_i$ manejada por el handler, se chequea el tipo de c_i, capturando σ_i. En estos chequeos se aplican las instanciaciones $\theta_i ::= \sigma_0\sigma_1 \dots \sigma_i$.

With Es similar a App. En el chequeo de c se capturan las operaciones no manejadas ni declaradas por el handler en σ, para luego añadirlas en el tipo resultante por medio de la aplicación de esta instanciación.

Let En primer lugar se sintetizan el tipos de c_1. Posteriormente se hace lo mismo con c_2, propagando las instanciaciones de la primera en el contexto. Luego,

se unifica la fila del primero con la del segundo, generando una instanciación σ. Por último, se devuelve el tipo generado para c_2 luego de aplicar dicha instanciación.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \overset{[]}{\Rightarrow} A}\ \textsc{Var} \qquad \frac{f : \forall\overline{\mu}.A \in \Gamma \qquad ren(\overline{\mu}) = \rho}{\Gamma \vdash f \overset{[]}{\Rightarrow} A[\rho]}\ \textsc{PVar} \qquad \frac{\Gamma \vdash e \overset{\eta}{\Rightarrow} \mathtt{nat}}{\Gamma \vdash \mathtt{succ}\ e \overset{\eta}{\Rightarrow} \mathtt{nat}}\ \textsc{Succ}$$

$$\frac{}{\Gamma \vdash \mathtt{true} \overset{[]}{\Rightarrow} \mathtt{bool}}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \mathtt{false} \overset{[]}{\Rightarrow} \mathtt{bool}}\ \textsc{False} \qquad \frac{}{\Gamma \vdash 0 \overset{[]}{\Rightarrow} \mathtt{nat}}\ \textsc{Zero}$$

$$\frac{\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}' \qquad \mathcal{T}' \overset{\sigma}{\bowtie}_T \mathcal{T}[\eta]}{\Gamma \vdash t \overset{\eta\sigma}{\Leftarrow} \mathcal{T}}\ \text{CS} \qquad \frac{}{\Gamma \vdash (\,) \overset{[]}{\Rightarrow} \mathtt{unit}}\ \textsc{Unit} \qquad \frac{\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \qquad \mathcal{T} \equiv \mathcal{T}[\sigma]}{\Gamma \vdash (t : \mathcal{T}) \overset{\sigma}{\Rightarrow} \mathcal{T}[\sigma]}\ \textsc{Anno}$$

$$\frac{\Gamma \vdash e_1 \overset{\eta}{\Rightarrow} A \to C\langle\mathcal{E}\rangle \qquad \Gamma[\eta] \vdash e_2 \overset{\sigma}{\Leftarrow} A}{\Gamma \vdash e_1\ e_2 \overset{\eta\sigma}{\Rightarrow} C\langle\mathcal{E}\rangle[\sigma]}\ \textsc{App} \qquad \frac{\Gamma,\ x : A \vdash c \overset{\sigma}{\Leftarrow} C\langle\mathcal{E}\rangle}{\Gamma \vdash \mathtt{fun}\ x \mapsto c \overset{\sigma}{\Leftarrow} A \to C\langle\mathcal{E}\rangle}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathtt{bool} \qquad \Gamma \vdash c_1 \overset{\eta}{\Rightarrow} C\langle\mathcal{E}\rangle \qquad \Gamma[\eta] \vdash c_2 \overset{\sigma}{\Leftarrow} C\langle\mathcal{E}\rangle}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \overset{\eta\sigma}{\Rightarrow} C\langle\mathcal{E}\rangle[\sigma]}\ \textsc{If}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathtt{nat} \qquad \Gamma \vdash c_1 \overset{\eta}{\Rightarrow} C\langle\mathcal{E}\rangle \qquad \Gamma[\eta],\ x : \mathtt{nat} \vdash c_2 \overset{\sigma}{\Leftarrow} C\langle\mathcal{E}\rangle}{\Gamma \vdash (\mathtt{match}\ e\ \mathtt{with}\ 0 \mapsto c_1 \mid \mathtt{succ}\ x \mapsto c_2) \overset{\eta\sigma}{\Rightarrow} C\langle\mathcal{E}\rangle[\sigma]}\ \textsc{Match}$$

$$\frac{\mathtt{Op} : A_{op} \to B_{op} \in \Sigma \qquad \Gamma \vdash e \Leftarrow A_{op} \qquad (\Gamma,\ y : B_{op}) \vdash c \overset{\eta}{\Rightarrow} A\langle\mathcal{E}\rangle \qquad add(\mathcal{E}, \mathtt{Op}) = \gamma}{\Gamma \vdash \mathtt{Op}\ e\ (y.c) \overset{\eta\gamma}{\Rightarrow} A\langle\mathcal{E}\rangle[\gamma]}\ \textsc{Op}$$

$$\frac{\begin{array}{ccc}\{\mathtt{Op}_i \neq \mathtt{Op}_j\}_{i \neq j} & \Delta \setminus \{\mathtt{Op}_i\}_{i=1}^{n} \subseteq \Delta' & \Gamma,\ x : A \vdash c_v \overset{\sigma_0}{\Leftarrow} B\langle\Delta' \mid \mu\rangle \\ \multicolumn{3}{c}{\left\{\mathtt{Op}_i : A_i \to B_i \in \Sigma \quad (\Gamma,\ x : A_i,\ k : B_i \to B\langle\Delta' \mid \mu\rangle)[\theta_i] \vdash c_i \overset{\sigma_i}{\Leftarrow} B\langle\Delta' \mid \mu\rangle[\theta_i]\right\}_{i=1}^{n}}\end{array}}{\Gamma \vdash \mathtt{handler}\ \mathtt{val}\ x \mapsto c_v,\ \{\mathtt{Op}_i\ x\ k \mapsto c_i\}_{i=1}^{n} \overset{\theta_k}{\Leftarrow} A\langle\Delta \mid \mu\rangle \twoheadrightarrow B\langle\Delta' \mid \mu\rangle}\ \textsc{Hand}$$

$$\frac{\Gamma \vdash h \overset{\eta}{\Rightarrow} A\langle\mathcal{E}\rangle \twoheadrightarrow B\langle\mathcal{E}'\rangle \qquad \Gamma[\eta] \vdash c \overset{\sigma}{\Leftarrow} A\langle\mathcal{E}\rangle}{\Gamma \vdash \mathtt{with}\ h\ \mathtt{handle}\ c \overset{\eta\sigma}{\Rightarrow} B\langle\mathcal{E}'\rangle[\sigma]}\ \textsc{With} \qquad \frac{\Gamma \vdash e \overset{\eta}{\Rightarrow} A}{\Gamma \vdash \mathtt{val}\ e \overset{\eta}{\Rightarrow} A\langle\mu\rangle}\ \textsc{Val}$$

$$\frac{\Gamma \vdash c_1 \overset{\eta_1}{\Rightarrow} A\langle\mathcal{E}_1\rangle \qquad \Gamma[\eta_1],\ x : \forall\overline{\mu}.A \vdash c_2 \overset{\eta_2}{\Rightarrow} B\langle\mathcal{E}_2\rangle \qquad \mathcal{E}_1[\eta_2] \overset{\sigma}{\bowtie}_R \mathcal{E}_2}{\Gamma \vdash \mathtt{let}\ x \coloneqq\ c_1\ \mathtt{in}\ c_2 \overset{\eta_1\eta_2\sigma}{\Rightarrow} B\langle\mathcal{E}_2\rangle[\sigma]}\ \textsc{Let}$$

Figura 3.10: Reglas de tipado bidireccional.

Ejemplos

Se mostrará el tipado de los mismos ejemplos usados para el sistema declarativo. Esto permite apreciar los puntos en común entre ambos y entender los mecanismos que garantizan la implementabilidad del bidireccional.

Nuevamente se divide el primer ejemplo en dos partes: el tipado de la función *apply* y el de la aplicación *apply crash*. Para el primero, se nota con Γ_1 al contexto con la asignación $f : \texttt{unit} \to \texttt{unit}\langle\mu\rangle$. Esta primera derivación se muestra en la Figura 3.11 y corresponde a chequeo, ya que se trata de la definición de una función.

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash f \overset{[]}{\Rightarrow} \texttt{unit} \to \texttt{unit}\langle\mu\rangle}\textsc{Var} \quad \dfrac{\vdots}{\Gamma_1 \vdash () \overset{[]}{\Leftarrow} \texttt{unit}}}{\Gamma_1 \vdash f\;() \overset{[]}{\Rightarrow} \texttt{unit}\langle\mu\rangle}\textsc{App} \quad \texttt{unit}\langle\mu\rangle \overset{[]}{\bowtie_T} \texttt{unit}\langle\mu\rangle}{\dfrac{f : \texttt{unit} \to \texttt{unit}\langle\mu\rangle \vdash f\;() \overset{[]}{\Leftarrow} \texttt{unit}\langle\mu\rangle}{\cdot \vdash \texttt{fun}\; f \mapsto f\;() \overset{[]}{\Leftarrow} (\texttt{unit} \to \texttt{unit}\langle\mu\rangle) \to \texttt{unit}\langle\mu\rangle}\textsc{Fun}}\text{CS}$$

Figura 3.11: Tipado de la función *apply*.

Además, como no se producen instanciaciones, se tiene que

$$\cdot \vdash apply \Leftarrow (\texttt{unit} \to \texttt{unit}\langle\mu\rangle) \to \texttt{unit}\langle\mu\rangle$$

La segunda parte, presente en la Figura 3.12, muestra la síntesis del tipo de *apply crash*.

$$\begin{aligned}
\Gamma &::= \; apply : (\texttt{unit} \to \texttt{unit}\langle\mu\rangle) \to \texttt{unit}\langle\mu\rangle, \\
&\quad\;\; crash : \texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu'\rangle \\
\mathcal{T}_1 &::= \; \texttt{unit} \to \texttt{unit}\langle\mu\rangle \\
\mathcal{T}_2 &::= \; \texttt{unit} \to \texttt{unit}\langle\texttt{Throw} \mid \mu'\rangle \\
\sigma &::= [\mu \mapsto \langle\texttt{Throw} \mid \mu_1\rangle,\; \mu' \mapsto \langle\mu_1\rangle]
\end{aligned}$$

$$\dfrac{\dfrac{}{\Gamma \vdash apply \overset{[]}{\Rightarrow} (\texttt{unit} \to \texttt{unit}\langle\mu\rangle) \to \texttt{unit}\langle\mu\rangle}\textsc{Var}^{\text{D}} \quad \dfrac{\dfrac{}{\Gamma \vdash crash \overset{[]}{\Rightarrow} \mathcal{T}_2}\textsc{Var} \quad \mathcal{T}_2 \overset{\sigma}{\bowtie_T} \mathcal{T}_1}{\Gamma \vdash crash \overset{\sigma}{\Leftarrow} \mathcal{T}_1}\text{CS}}{\Gamma \vdash apply\; crash \overset{\sigma}{\Rightarrow} \texttt{unit}\langle\texttt{Throw} \mid \mu_1\rangle}\textsc{App}$$

Figura 3.12: Tipado del término *apply crash*.

El siguiente ejemplo corresponde al tipado de la aplicación de un handler que cuenta el número de llamados a `Print` de un programa. Su derivación se encuentra en la Figura 3.13.

Sean
$\Gamma_1 ::= x : \texttt{unit},\ k : \texttt{nat} \to \texttt{nat}\langle\mu_1\rangle$
$\rho_1 ::= [\mu \mapsto \langle\mu_1\rangle]$
$\sigma_1 ::= [\mu_1 \mapsto \langle\mu_3\rangle,\ \mu_2 \mapsto \langle\mu_3\rangle]$
$\mathcal{T} ::= \texttt{unit}\langle\texttt{Print} \mid \mu\rangle \twoheadrightarrow \texttt{nat}\langle\mu\rangle$
$T ::=$

$$\dfrac{\dfrac{\vdots}{x : \texttt{unit} \vdash \texttt{val}\ 0 \overset{\rho_1}{\Leftarrow} \texttt{nat}\langle\mu\rangle}\textsc{Val} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma_1 \vdash k\ (\,) \overset{[]}{\Rightarrow} \texttt{nat}\langle\mu_1\rangle}\textsc{App} \quad \dfrac{\vdots}{\Gamma_2 \vdash \texttt{val}\ (\texttt{succ}\ y) \overset{[]}{\Rightarrow} \texttt{nat}\langle\mu_2\rangle}\textsc{Val} \quad \langle\mu_1\rangle \overset{\sigma_1}{\bowtie_R} \langle\mu_2\rangle}{\Gamma_1 \vdash \texttt{let}\ y := k\ (\,)\ \texttt{in}\ \texttt{val}\ (\texttt{succ}\ y) \overset{\sigma_1}{\Rightarrow} \texttt{nat}\langle\mu_3\rangle}\textsc{Let} \quad \texttt{nat}\langle\mu_3\rangle \overset{[]}{\bowtie_T} \texttt{nat}\langle\mu_3\rangle}{\Gamma_1 \vdash \texttt{let}\ y := k\ (\,)\ \texttt{in}\ \texttt{val}\ (\texttt{succ}\ y) \overset{\sigma_1}{\Leftarrow} \texttt{nat}\langle\mu_1\rangle}\textsc{CS}}{\dfrac{\cdot \vdash h \overset{\rho_1\sigma_1}{\Leftarrow} \texttt{unit}\langle\texttt{Print} \mid \mu\rangle \twoheadrightarrow \texttt{nat}\langle\mu\rangle}{\cdot \vdash h : \mathcal{T} \overset{\rho_1\sigma_1}{\Rightarrow} \texttt{unit}\langle\texttt{Print} \mid \mu_3\rangle \twoheadrightarrow \texttt{nat}\langle\mu_3\rangle}\textsc{Anno}}\textsc{Hand}$$

$\eta ::= \rho_1\sigma_1 = [\mu \mapsto \langle\mu_3\rangle,\ \mu_1 \mapsto \langle\mu_3\rangle,\ \mu_2 \mapsto \langle\mu_3\rangle]$
$\gamma_1 ::= [\mu_4 \mapsto \langle\texttt{Print} \mid \mu_5\rangle]$
$\sigma_2 ::= [\mu_5 \mapsto \langle\mu_6\rangle,\ \mu_3 \mapsto \langle\mu_6\rangle]$
en:

$$\dfrac{\dfrac{T}{\cdot \vdash h : \mathcal{T} \overset{\eta}{\Rightarrow} \texttt{unit}\langle\texttt{Print} \mid \mu_3\rangle \twoheadrightarrow \texttt{nat}\langle\mu_3\rangle} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\cdot \vdash 7 \Leftarrow \texttt{nat}} \quad \dfrac{\vdots}{z : \texttt{unit} \vdash \texttt{val}\ z \Rightarrow \texttt{unit}\langle\mu_4\rangle}\textsc{Val} \quad add(\langle\mu_4\rangle, \texttt{Print}) = \gamma_1}{\cdot \vdash \texttt{Print}\ 7\ z.(\texttt{val}\ z) \overset{\gamma_1}{\Rightarrow} \texttt{unit}\langle\texttt{Print} \mid \mu_5\rangle}\textsc{Op} \quad \texttt{unit}\langle\texttt{Print} \mid \mu_5\rangle \overset{\sigma_2}{\bowtie_T} \texttt{unit}\langle\texttt{Print} \mid \mu_3\rangle}{\cdot \vdash \texttt{Print}\ 7\ z.(\texttt{val}\ z) \overset{\gamma_1\sigma_2}{\Leftarrow} \texttt{unit}\langle\texttt{Print} \mid \mu_3\rangle}\textsc{CS}}{\cdot \vdash \texttt{with}\ (h : \mathcal{T})\ \texttt{handle}\ \texttt{Print}(7) \overset{\eta\gamma_1\sigma_2}{\Rightarrow} \texttt{nat}\langle\mu_6\rangle}\textsc{With}$$

Figura 3.13: Tipado bidireccional de un handler.

3.5. Correspondencia de ambos sistemas

Hasta ahora se tienen dos sistemas de tipos para *Alef*: uno declarativo sobre el cual se enuncia la seguridad y otro bidireccional que corresponde a la implementación concreta. Para garantizar que lo demostrado sobre el primero vale también en el segundo, en esta sección se establece una correspondencia entre ambos.

Consistencia y completitud

La correspondencia mencionada anteriormente se divide en dos teoremas: uno que lleva del sistema bidireccional al declarativo (Consistencia) y el otro en el sentido inverso (Completitud). Con estos dos teoremas es simple extender el resultado de seguridad al sistema bidireccional.

Teorema 2 (Consistencia). $\forall\, \Gamma, t, \mathcal{T}, \sigma, \eta$
$(\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \longrightarrow \Gamma[\sigma] \vdash t[\sigma] : \mathcal{T}[\sigma]) \wedge$
$(\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T} \longrightarrow \Gamma[\eta] \vdash t[\eta] : \mathcal{T})$

Corolario 1. $\forall\, \Gamma, t, \mathcal{T}$
$\Gamma \vdash t \Leftarrow \mathcal{T} \longrightarrow \Gamma \vdash t : \mathcal{T}$

Se introduce la notación $\Gamma_1, \mathcal{T}_1 \sqsubseteq \Gamma_2, \mathcal{T}_2$ para abreviar $\Gamma_1, z : \mathcal{T}_1 \sqsubseteq \Gamma_2, z : \mathcal{T}_2$ donde z es una variable que no pertenece a Γ_1.

Adicionalmente se dice que un término t está *suficientemente anotado*, y se nota SA(t), si todos sus sub-términos de la forma $\texttt{fun}\ x \mapsto c$ o $\texttt{handler val}\ x \mapsto c_v,\ \{\texttt{Op}_i\ x\ k \mapsto c_i\}_{i=1}^{n}$ que no sean argumento de una función se encuentran anotados.

Teorema 3 (Completitud). $\forall\, \Gamma, t, \mathcal{T}$

$$\Gamma \vdash t : \mathcal{T} \wedge SA(t) \longrightarrow \quad \exists \eta, \mathcal{T}'\ (\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}' \wedge \Gamma[\eta], \mathcal{T}' \sqsubseteq \Gamma, \mathcal{T}) \vee \tag{5}$$

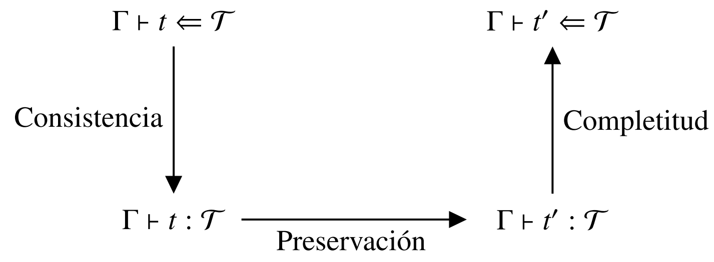
$$\exists \sigma\ (\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \wedge \Gamma, \mathcal{T} \equiv (\Gamma, \mathcal{T})[\sigma]) \tag{6}$$

Corolario 2. $\forall\, \Gamma, t, \mathcal{T}$
$\Gamma \vdash t : \mathcal{T} \wedge SA(t) \longrightarrow \Gamma \vdash t \Leftarrow \mathcal{T}$

El teorema de Preservación se extiende al sistema implementable de la siguiente manera. La Preservación del sistema bidireccional se presenta en base al juicio de chequeo final, ya que este es el que se usa en el top-level y su formulación es la más simple.

Sea t un término tal que exista un t' para el que $t \rightsquigarrow t'$ y $\Gamma, \mathcal{T}$ tales que $\Gamma \vdash t \Leftarrow \mathcal{T}$. Como t evalúa a otro término, t es una computación, por lo que cuenta con regla de síntesis. Luego, por Lema 10, se tiene que SA(t) y por Lema 11, SA(t').

Luego, se alcanza la Preservación en el sistema bidireccional de la siguiente manera:



Para teorema de Progreso se procede de una manera similar, efectivamente extendiendo el resultado de seguridad del sistema declarativo sobre el bidireccional.

Teorema 4. *El sistema de tipos bidireccional es seguro.*

Capítulo 4

Comparación con otros sistemas

A continuación se presentará una comparación de *Alef* con tres de los principales lenguajes con sistemas de efectos nombrados en el Capítulo 1: *Eff*, *Frank* y *Koka*. Se eligieron estos tres porque, además de ser los lenguajes sobre los cuales más trabajo se ha realizado sobre efectos algebraicos, cada uno de ellos ha influenciado el diseño de *Alef* en cierta medida.

¿Por qué no inferencia?

Antes de comenzar las comparaciones, se hace un breve comentario acerca de la distinción entre inferencia y chequeo de tipos, y la razón por la cual se eligió la segunda en *Alef*.

Dos de los sistemas con los que se compara, Koka y Eff, cuentan con sistemas de inferencia de tipos y efectos (en el caso de Eff, no forma parte de la implementación). Estos sistemas evitan por completo la necesidad de anotaciones de tipo por parte del programador. Sin embargo, para esta tesina se optó utilizar la técnica de chequeo bidireccional principalmente para concentrar la atención en control de los efectos, ya que la inferencia de tipos valor es un problema independiente, con soluciones establecidas.

De todas maneras, la versión de tipado bidireccional desarrollado para *Alef* es capaz de sintetizar la información de efectos, por lo que extenderlo para implementar inferencia no debería traer mayores complicaciones.

4.1. Eff

En primer lugar se establecerá una comparación con Eff, ya que es el lenguaje sobre el cual se basa *Alef*. En particular, el lenguaje de esta tesina deriva directamente de Core Eff [3].

Si bien la definición de los tipos de *Alef* sigue la de los de este sub-lenguaje de Eff, estos se diferencian en cómo se establece el polimorfismo de efectos. Eff utiliza subtipado de efectos, mientras que en *Alef* se usan filas de efectos. Esta distinción

tiene consecuencias semánticas, ya que el enfoque usado en esta tesina resulta en tipos más expresivos.

Para ver esto en concreto, se vuelve a los ejemplos de la Sección 3.2 en el cual se introducen las funciones de alto orden:

$$\begin{aligned}
&\textit{ignore} : (\texttt{unit} \to \texttt{unit}\langle\mu_1\rangle) \to \texttt{unit}\langle\mu_2\rangle \\
&\textit{ignore} ::= \texttt{fun}\ f \mapsto \texttt{val}\ (\,) \\
\\
&\textit{apply}\ : (\texttt{unit} \to \texttt{unit}\langle\mu_1\rangle) \to \texttt{unit}\langle\mu_1\rangle \\
&\textit{apply}\ ::= \texttt{fun}\ f \mapsto f\ (\,)
\end{aligned}$$

Figura 4.1: Ejemplo de funciones de alto orden.

Como se mostró antes, la posibilidad de usar distintas variables de efectos permite diferenciar a nivel de tipos estas funciones. De esta manera, los tipos permiten expresar que *apply* tendrá los mismos efectos que la función que toma como argumento, mientras que los efectos *ignore* son independientes.

En contraste, en Eff ambas funciones tienen el tipo

$$(\texttt{unit} \to \texttt{unit!}\emptyset) \to \texttt{unit!}\emptyset$$

haciendo imposible esta distinción.

Otra diferencia entre los tipos de estos lenguajes radica en que el sistema declarativo de Eff describe instancias de efectos. Se decidió dejar de lado estas para el desarrollo de *Alef*, ya que el manejo de su generación dinámica complica significativamente el tipado de efectos (por lo cual tampoco se incluyen en Core Eff) y las instancias estáticas no aportan semánticamente nada, simplemente pueden considerarse renombres de las operaciones.

Teniendo en cuenta estas diferencias, las reglas del sistema declarativo de *Alef* tienen una estructura similar a las de Eff, en particular a su presentación en el tutorial de Pretnar [27]. En ambos, la única forma de promover la información de efectos de un término es por medio de la regla para computaciones puras, evitando la necesidad de subtipado y, por ende, resultando en un sistema más simple.

Adicionalmente, Pretnar [28] describe un sistema de inferencia de tipos y efectos que consiste en una versión implementable del anterior. Sin embargo, este no forma parte de la implementación actual de Eff. Este sistema de inferencia está basado en la recolección de restricciones que, luego del tipado, son resueltas por una función de unificación. Además, se extienden los tipos con *parámetros de tipos*, *parámetros de región* (que capturan información de las instancias de efectos), y un *dirt parameter*, que cumple la función de la variable de efectos en un sistema con polimorfismo de filas, pero al ser única no añade la expresividad mencionada anteriormente.

Estas extensiones, especialmente las restricciones, complejizan significativamente los tipos del sistema. Esto no solo hace más difícil razonar sobre este sistema, sino que también dificultan su interacción con el usuario. Si bien se presentan mecanismos de simplificación de restricciones, resulta necesario descartar algunas de estas para poder mostrar los tipos inferidos, comunicando tipos menos precisos. Este sistema de efectos de Eff no incorpora una construcción para anotar el tipo de un término, lo cual, a pesar de no ser necesario por contar con inferencia, puede ser útil en la práctica. La complejidad de los tipos sería una barrera para añadir anotaciones de tipo al lenguaje, ya que los usuarios deberían escribir los tipos con todas las restricciones.

En cuanto a la implementación concreta de estos sistemas, no es posible hacer una comparación puesto que la versión de Eff disponible[1] no cuenta con un sistema de tipos. Esto se debe a que ninguno de los sistemas implementables de Eff logró incluir el comportamiento de las instancias dinámicas, que son parte del lenguaje.

4.2. Frank

Como se introdujo en la Sección 1.1, Frank [19] es un lenguaje funcional aplicativo que introduce una variante de los handlers descriptos originalmente por Plotkin y Pretnar [24]. Esta consiste en generalizar las funciones y los handlers a una única abstracción, llamada *operador*, que puede manejar múltiples computaciones impuras a la vez, convirtiéndose en *multihandlers*.

En contraste a *Alef*, que cuenta con handlers profundos, Frank usa *handlers superficiales*. La diferencia de estos respecto a los primeros es que al ser aplicados, solo manejan un paso de la computación. En concreto, en la semántica del manejo de una operación por un handler superficial no se vuelve a aplicar el handler sobre la continuación que captura el resto del programa. Otra manera de ver la diferencia entre estos tipos de handlers consiste en que los profundos representan un *fold* sobre el árbol de la computación que maneja, mientras que los superficiales son análogos a hacer análisis por casos sobre su raíz.

A pesar de que el uso de handlers superficiales simplifica la escritura de ciertos programas, los profundos se comportan de manera más regular y su semántica de *fold* hace que sea más fácil razonar sobre ellos. Adicionalmente, las implementaciones de handlers profundos suelen ser más eficientes [13]. Estas variaciones que introduce Frank también se ven reflejadas en sus tipos. Para incorporar estas variaciones, en particular los multihandlers, los tipos de Frank se alejan de la formulación original de Plotkin y Power. Como consecuencia, se pierde en cierta medida la semántica denotacional de tipos basada en la teoría presentada en la Sección 2.2.

Para contar con polimorfismo de efectos, Frank usa una variante de polimorfismo de fila similar a la usada en *Alef*, pero en Frank se busca evitar escribir la variable de efectos en el código. Para esto, se introduce azúcar sintáctico que incorpora variables frescas a los tipos. Sin embargo, esto no se puede evitar del todo, ya que esta elabo-

[1] Disponible en https://github.com/matijapretnar/eff/.

ración automática no puede distinguir la diferencia semántica vista en la Figura 4.1 de la sección anterior.

Los sistemas de tipado tanto de Frank como de *Alef* se basan en la técnica bidireccional, pero no de la misma manera. La principal diferencia radica en que en Frank la información de efectos siempre fluye "hacia adentro", es decir, siempre es chequeada, por lo que debe ser llevada de manera adicional en ambos juicios de tipado. En cambio, en *Alef* toda la información de efectos se encuentra en los tipos y, en consecuencia, es chequeada en el juicio de chequeo y sintetizada en el juicio de síntesis. Si bien la primera alternativa puede ser más simple de formular, requiere de tipos más gruesos y la segunda obtiene mejores mensajes de error, ya que permite mostrar el tipo con efectos sintetizado junto al que se chequea.

Para ver puntualmente cómo la alternativa usada en Frank requiere tipos más gruesos, se vuelve al ejemplo de las funciones de alto orden de la sección anterior. Si se quisiera tipar la aplicación *ignore* ($\lambda_.\texttt{Op}()$), el tipo con el que se debería declarar *ignore* tendría que ser análogo a

$$(\texttt{unit} \to \texttt{unit}\langle \texttt{Op} \mid \mu_1 \rangle) \to \texttt{unit}\langle \mu_2 \rangle$$

Esto se debe a que en Frank no es posible instanciar μ_1 a $\langle \texttt{Op} | \mu_1' \rangle$ en el tipado y propagar esta instanciación "hacia afuera". Esto se ve claramente en la regla de cambio de modo, en la que para chequear con un tipo $\mathcal{T}$ un término para el cual se sintetiza el tipo $\mathcal{T}'$, se requiere que $\mathcal{T}$ y $\mathcal{T}'$ sean exactamente iguales.

Sin embargo, la implementación de Frank[2] soluciona estos inconvenientes introduciendo un mecanismo de unificación que reemplaza esa igualdad de tipos. De esta manera, la implementación permite propagar información de tipado "hacia afuera", diferenciándose de su planteo teórico. Si bien esto logra simplificar las anotaciones, y en particular hace que la implementación de Frank acepte el ejemplo de *ignore* con el mismo tipo que en *Alef*, este mecanismo no se incluye dentro de lo formulación teórica del sistema de efectos de Frank, como sí se hace con la relación de instanciación $\bowtie_T$ en *Alef*.

4.3. Koka

El lenguaje Koka [17, 18], a diferencia de los dos anteriores y *Alef*, no se diseñó originalmente en base a la teoría de efectos algebraicos, sino que los conceptos derivados de esta se introdujeron posteriormente. Por esta razón se pueden notar significativas diferencias entre Koka y el resto de estos lenguajes, en particular en sus tipos.

En primer lugar, la mayor distinción entre los tipos de Koka y *Alef* es que el primero no cuenta con tipos computación. En cambio, la información de efectos de Koka siempre se encuentra en el tipo de las funciones, ya que la aplicación de estas son el único punto en el cual se pueden invocar operaciones en este lenguaje.

[2]Disponible en https://github.com/frank-lang/frank.

Esta distinción cobró importancia una vez que Koka incorporó partes de la teoría de efectos algebraicos. Su versión inicial [17] solo consideraba efectos intrínsecos del lenguaje, no efectos definidos por el usuario. Por esto, no contaba con handlers generales, sino que cada efecto tenía un destructor específico. Una versión posterior [18] extiende Koka añadiendo un mecanismo para manejar computaciones, junto a azúcar sintáctico que permite escribir handlers mediante funciones. Sin embargo, los tipos de Koka no se modificaron al incorporar esto y los handlers, en vez de tener un constructor de tipos que asocia un par de tipos computación (de la forma $\underline{C} \twoheadrightarrow \underline{D}$), se codifican mediante tipos función análogos a $(\texttt{unit} \to \underline{C}) \to (\texttt{unit} \to \underline{D})$.

Todo esto tiene como consecuencia que Koka abandona la semántica denotacional categórica de tipos basada en el desarrollo introducido en la Sección 2.2, perdiendo la claridad conceptual que esta provee para el razonamiento sobre programas.

Koka implementa polimorfismo de efectos incorporando filas de efectos en sus tipos. Si bien estas fueron la base de las usadas en *Alef* (por lo cual los tipos de ambos son igual de expresivos), se diferencian en que las de Koka permiten la duplicación de un efecto en una fila. Permitir esto, además de simplificar la implementación de la unificación, es necesario en este lenguaje ya que sus construcciones son exclusivamente “eliminadores de efectos”. Esto significa que sus funciones y handlers solo pueden remover efectos del tipo y no agregar efectos. Entonces, por ejemplo, si se tuviera un handler de excepciones que a su vez pueda lanzar estas mismas, el tipo de la computación de entrada debería tener duplicado este efecto, para que pueda estar en el tipo de la computación salida.

La duplicación de efectos también se utiliza en Koka para la construcción *inject*, que permite encapsular efectos, asumiendo otro significado distinto. A pesar de la utilidad práctica de permitir efectos duplicados, estos no cuentan con una semántica clara, reforzando la diferencia con *Alef* respecto a la semántica de sus tipos.

Capítulo 5

Implementación

A continuación se introduce la implementación de *Alef*, la cual se nota `Alef`. Esta fue desarrollada en el lenguaje de programación *Haskell*, siguiendo la formalización presentada en el capítulo anterior.

Este capítulo se organiza de la siguiente manera: en primer lugar se describe la estructura del código fuente, luego se introduce el azúcar sintáctico utilizado para simplificar la escritura de programas, seguidamente se presentan ejemplos concretos y para concluir se muestran los casos de prueba generados.

El código fuente de esta implementación se encuentra disponible en el repositorio: `https://github.com/antoniolocascio/Alef/`.

5.1. Estructura del código

Para comenzar se presenta en el Cuadro 5.1 una síntesis de la estructura general del código fuente. Después de esta, se provee una descripción más detallada de los componentes principales del mismo.

Términos

Los tipos de datos que describen la sintaxis abstracta de *Alef* se definen en el módulo `AST`, y se presentan en el Código 5.2. Siguiendo la gramática formulada en el Capítulo 3, los términos (`Term`) se dividen en dos categorías: expresiones (`Exp`) y computaciones (`Comp`).

Para cada construcción sintáctica se provee el constructor del tipo que corresponde. Las cláusulas de un handler se describen mediante una lista cuyos elementos tienen una operación, dos variables (x y k) y una computación c_i. La llamada a una operación, además de esta última, requiere una expresión para su parámetro y una variable y una computación para la continuación. El resto de los constructores son triviales.

	Módulo	**Descripción**
Utilidades	`Error`	Define la clase `Fallible` que describe computaciones que pueden fallar.
	`Set`	Define una interfaz para manipular conjuntos.
	`Unique`	Define la clase `UniqueGen`, que provee una operación que devuelve un valor nuevo cada vez que se la llama.
	`Substitution`	Define las sustituciones de manera generalizada. El tipo `Substitution k v` corresponde a las sustituciones de valores de tipo `k` por valores de tipo `v`.
Sintaxis	`AST`	Define el tipo de datos `Term` que se utiliza para representar los términos de `Alef`.
	`Types`	Define los tipos de `Alef`, representados por el tipo de datos `Type`.
	`EffectRow`	Describe la representación y manejo de filas de efectos.
	`FreshVars`	Introduce la clase `FreshGen`, que incluye una operación que devuelve una variable de efectos fresca.
Tipado	`RowUnification`	Implementa la instanciación de variables de efecto.
	`TypeChecker`	Concretiza las reglas del sistema bidireccional, contando con una función para síntesis y una para chequeo.
Evaluador	`Eval`	Implementa la relación de evaluación en base a la semántica operacional.
	`TopLevelOps`	Proporciona el manejo de operaciones por el entorno.
Parser	`Lexer`	Define el analizador léxico de `Alef`.
	`Parser`	Define el parser del `Alef`.
Printer	`PPAST`	Función de impresión de términos.
	`PPTypes`	Función de impresión de tipos.
	`PPSubstitution`	Función de impresión de sustituciones.
Azúcar	`SugarTypes`	Define tipos usados en el módulo `Sugar`.
Sintáctico	`Sugar`	Desarrolla la elaboración de la Sección 5.2.

Cuadro 5.1: Estructura de la implementación de `Alef`.

```
data Term = E Exp | C Comp

data Exp where
    EVar   ::Var -> Exp
    ETrue  ::Exp
    EFalse::Exp
    EZero  ::Exp
    ESucc  ::Exp -> Exp
    EUnit  ::Exp
    EFunc  ::Var -> Comp -> Exp
    EHand  ::Var   -- val x
            -> Comp -- cv
            -> [(Operation -- op
                , Var -- x
                , Var -- k
                , Comp -- ci
                )]
            -> Exp
    EAnno  ::Exp -> VType -> Exp

data Comp where
    CVal   ::Exp -> Comp
    COp    ::Operation -> Exp -> Var -> Comp -> Comp
    CWith  ::Exp -> Comp -> Comp
    CApp   ::Exp -> Exp -> Comp
    CIf    ::Exp -> Comp -> Comp -> Comp
    CLet   ::Var -> Comp -> Comp -> Comp
    CMatch::Exp -> Comp -> Var -> Comp -> Comp
    CAnno  ::Comp -> CType -> Comp
```

Código 5.2: Tipos de datos que definen el AST.

Tipos

En primer lugar es necesario tener una representación de las filas de efecto, que estarán presentes en el tipo de las computaciones. Para ello se hace uso de la observación planteada en el Capítulo 3, que indica que una fila en *Alef* puede verse como un conjunto de operaciones junto a una variable de efectos. El Código 5.3 muestra en concreto esta representación.

```
type Delta = Set.Set Operation
type EffRow = (Delta, EffVar)
```

Código 5.3: Representación de las filas de efectos.

Los tipos tienen un tratamiento similar a los términos. Estos, descriptos por el tipo `Type` están compuestos por tipos valor (`VType`) y tipos computación (`CType`).

```
data Type = VT VType | CT CType

data VType where
    TBool  ::VType
    TNat   ::VType
    TUnit  ::VType
    TEmpty::VType
    TFunc  ::VType -> CType -> VType
    THand  ::CType -> CType -> VType

data CType where
    TComp  ::VType -> EffRow -> CType
```

Código 5.4: Representación de tipos.

Sistema de tipado

El módulo `TypeEnv` define una clase con el mismo nombre para representar el entorno de tipado y la signatura de efectos. En ella se definen las operaciones de búsqueda del tipo de una variable en el entorno, de búsqueda del tipo de una operación en la signatura y de extensión del entorno.

Para poder definir las reglas de tipado, se debe primero implementar las operaciones de instanciación de variables de efectos $\bowtie_R$ y $\bowtie_T$. Los tipos de estas se presentan en el Código 5.5 y las funciones se encuentran en el módulo `RowUnification`. La única diferencia con la definición conceptual es que `instTypes` además toma un término que se utiliza para mejorar los mensajes de error.

```
instRows
    :: (FreshGen m, Monad m)
    => EffRow
    -> EffRow
    -> m (Substitution EffVar EffRow)

instTypes
    :: (FreshGen m, Monad m, Fallible m)
    => Type
    -> Type
    -> Term
    -> m (Substitution EffVar EffRow)
```

Código 5.5: Tipos de las operaciones de instanciación.

Con todo esto es posible definir las funciones que implementan los juicios del sistema de tipado bidireccional. Como se mencionó previamente, se introduce una función para cada juicio. Todas estas se definen en el módulo `TypeChecker`, en el que además se define una clase con este mismo nombre que tan solo agrega las restricciones de `Fallible`, `Monad` y `FreshGen`.

```
synthType
    :: (TypeChecker m, TypeEnv e)
    => e
    -> Term
    -> m (Type, Substitution EffVar EffRow)

checkTypeS
    :: (TypeChecker m, TypeEnv e)
    => e
    -> Term
    -> Type
    -> m (Substitution EffVar EffRow)
```

Código 5.6: Tipos para las funciones de síntesis y chequeo.

La función `synthType` corresponde al juicio de síntesis $\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}$. Como puede apreciarse en el Código 5.6, su tipo refleja que en este caso Γ y t son entradas, mientras que $\mathcal{T}$ y η son potenciales salidas.

En el caso de `checkTypeS`, que implementa $\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T}$, se tiene que $\mathcal{T}$ también es una entrada, mientras que la sustitución σ es la única salida generada, de no fallar.

Una ventaja del sistema de tipado bidireccional es que las definiciones de estas funciones son una traducción directa de las reglas. Por ejemplo, se toma la regla App:

$$\frac{\Gamma \vdash e_1 \overset{\eta}{\Rightarrow} A \rightarrow C\langle\mathcal{E}\rangle \qquad \Gamma[\eta] \vdash e_2 \overset{\sigma}{\Leftarrow} A}{\eta \vdash e_1\ e_2 \overset{\eta\sigma}{\Rightarrow} C\langle\mathcal{E}\rangle[\sigma]}\ \textsc{App}$$

Su implementación se presenta en el Código 5.7. Esta primero sintetiza el tipo de e_1. Si este corresponde a una función, chequea el de e_2 y devuelve el nuevo tipo sintetizado junto con la composición de las instanciaciones.

En este punto se introduce la única diferencia significativa de la implementación respecto del formalismo presentado en el capítulo anterior. Para simplificar el mecanismo de chequeo de tipos, la implementación mantiene un invariante más fuerte acerca de las instanciaciones. Esto se refleja en el juicio de chequeo final, cuya definición en este caso hace uso del predicado de renombre inyectivo (*ri*) definido en la Sección 3.4:

$$\Gamma \vdash t \Leftarrow \mathcal{T} ::= (\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T}) \wedge ri(\sigma \lhd \mathrm{FV}(\mathcal{T}))$$

donde $\lhd$ simboliza la restricción de dominio.

```
synthType tenv ctx@(C (CApp e1 e2)) = do
    (te1, eta) <- synthType tenv (E e1)
    case te1 of
        (VT (TFunc a c)) -> do
            s <- checkTypeS (apply eta tenv) (E e2) (VT a)
            return (CT (apply s c), eta ∘ s)
        _ -> throw $ notAFunction (E e1) ctx te1
```

Código 5.7: Definición de la regla App.

Esta verificación sobre la instanciación σ garantiza que $\mathcal{T} \equiv \mathcal{T}[\sigma]$.

Teniendo en cuenta esta aclaración, la función que implementa el juicio $\Gamma \vdash t \Leftarrow \mathcal{T}$ también se incluye en este módulo con el nombre `checkType`. Su definición es una traducción de la formulación anterior y se presenta en el Código 5.8. Nuevamente, esta función devuelve la sustitución producida simplemente para mejorar los mensajes de error.

```
checkType
    :: (TypeChecker m, TypeEnv e)
    => e
    -> Term
    -> Type
    -> m (Substitution EffVar EffRow)
checkType tenv t tau = do
    s <- checkTypeS tenv t tau
    if ri (dres s (fv tau)) renaming
        then return s
        else throw $ notAlphaEq tau s t
```

Código 5.8: Definición del juicio de chequeo.

Evaluador

Por último, el módulo `Eval` implementa la semántica operacional small-step. Esto se hace definiendo una función `step`, que describe un paso de evaluación. Su tipo se muestra en el Código 5.9. Esta además lleva el entorno de tipado, ya que hay casos en los que es necesario anotar sub-términos al evaluar, y para esto se debe contar con el tipo de las operaciones.

Adicionalmente, este módulo también define la función `steps`, que evalúa una computación hasta alcanzar un valor o un llamado a una operación que no es manejado por ningún handler. Si se cuenta con el tipo de la computación y se alcanza un llamado a una operación, se verifica que esta esté presente en el tipo.

```
step
    :: (Fallible m, TypeEnv e)
    => e
    -> Comp
    -> m Comp

steps
    :: (Fallible m, TypeEnv e)
    => e
    -> (Comp, Maybe CType)
    -> m Comp
```

Código 5.9: Tipos de la funciones de evaluación.

5.2. Azúcar sintáctico

Con el fin de simplificar la escritura de programas, se introduce azúcar sintáctico para `Alef`. Este permite definir la signatura de efectos del programa y dividir este último en múltiples declaraciones de expresiones, cada una acompañada por el tipo correspondiente. Las declaraciones de expresiones son secuenciales, por lo que el nombre que se le da a una solo puede aparecer en las definiciones subsecuentes.

Adicionalmente, en esta elaboración se renombran las variables de tipo de cada signatura y anotación en el código, ya que estas están universalmente cuantificadas en cada anotación por separado de manera implícita.

Un ejemplo de su uso puede verse en el Código 5.10, en donde se declara primero un handler y posteriormente el `main`, que debe estar presente en todos los programas `Alef` al final de las declaraciones. Como puede verse en este ejemplo, la signatura de efectos es opcional. De estar presente se debe encontrar al inicio del programa, como muestra el Código 5.13. Todos los programas `Alef` cuentan con la siguiente signatura por defecto, cuyas operaciones son manejadas por el entorno:

```
Throw : nat  → empty
Print : nat  → unit
Read  : unit → nat
```

```
count : unit⟨Print | μ⟩ ↠ nat⟨μ⟩
count = handler
    val x → val 0,
    {Print x k → let a = k () in val (succ a)};;

main : nat⟨μ⟩
main = with count handle
    (let x = Print(1) in
     let y = Print(2) in
     if false then Print(3) else val ())
```

Código 5.10: `count.alef`

Formalmente, esta elaboración de `Alef` acepta programas de acuerdo a la siguiente gramática:

$$
\begin{array}{lll}
\mathit{Program} & ::= & \mathit{Signature} \mid \mathit{DecList} \\
\mathit{Signature} & ::= & \texttt{signature}\{\mathit{OpList}\}\ \mathit{DecList} \\
\mathit{DecList} & ::= & \mathit{Dec}\ \mathit{DecList} \mid \mathit{Main} \\
\mathit{OpList} & ::= & \mathit{OpDef},\ \mathit{OpList} \mid \mathit{OpDef} \\
\mathit{OpDef} & ::= & \mathbf{Op} : \mathit{BT} \to \mathit{BT} \\
\mathit{Dec} & ::= & \mathbf{id} : A \\
 & & \mathbf{id} = e;; \\
\mathit{Main} & ::= & \texttt{main} : \underline{C} \\
 & & \texttt{main} = c
\end{array}
$$

Donde BT son tipos base y $\underline{C}$ tipos computación de la sintaxis de tipos de la Sección 3.2, e son expresiones y c computaciones de la sintaxis de términos de la Sección 3.1, **Op** son identificadores de operaciones que inician con una letra mayúscula e **id** son identificadores de expresiones que inician con una letra minúscula.

5.3. Ejemplos

A continuación se muestran algunos programas *Alef*, junto a su evaluación. Para comenzar se tienen los ejemplos presentados en el Capítulo 3: un handler que cuenta el número de llamadas a `Print` de un programa y una función de alto orden que aplica la función tomada como parámetro.

Contando Prints

En el primero de estos, cuya implementación corresponde al Código 5.10, se define el handler `count`, que hace lo siguiente: si se encuentra con un valor, devuelve

el número cero, si en cambio debe manejar un llamado a `Print`, simplemente ejecuta la continuación y devuelve el número que esta retorna adicionando uno.

El `main` de este programa maneja con `count` una secuencia de computaciones. Al ejecutarlo, este retorna el valor 2, que es el número de llamadas a `Print` del programa.

Funciones de alto orden

El segundo ejemplo, Código 5.11, define la función `apply`, que a su vez toma una función `f` y la aplica. Para utilizarla, se define `g`, que simplemente ignora su entrada y llama a `Print`.

```
apply : (unit → unit⟨μ⟩) → unit⟨μ⟩
apply = fun f  →  f ();;

g : unit → unit⟨Print | μ⟩
g = fun u → Print(1);;

main : unit⟨Print | μ⟩
main = apply g
```

Código 5.11: `apply.alef`

Al ejecutarse este programa se imprime por pantalla el número 1 y se retorna el valor *unit*.

Como se explicó en el capítulo anterior, este ejemplo muestra la importancia del nombre de las variables de efecto en un tipo. Si en el tipo de `apply` se usaran dos variables distintas, por ejemplo μ_1 y μ_2, el chequeador de tipos produciría un error indicando que estas deben ser iguales. En concreto, si el programador le diera a `apply` la signatura:

```
apply : (unit → unit⟨μ0⟩) → unit⟨μ1⟩
```

Al ejecutar el programa, se produciría el siguiente error:

```
Error:
Type cannot be checked, types not alpha-equivalent:
Expected type: ((unit -> unit<_mu1>) -> unit<_mu2>)
Actual type: ((unit -> unit<_mu5>) -> unit<_mu5>)
For Term: apply
```

Que muestra que las variables de efectos del tipo de `apply` deben ser iguales.

Sin embargo, una función a la que se le puede asignar el tipo anterior se encuentra definida en el Código 5.12. Este es muy similar al programa previo, pero ahora se define una función `ignore` que ignora la función tomada como argumento. Por esta

razón los efectos de `f` son independientes de los de `ignore`, lo que se refleja en los nombres de variables.

```
ignore : (unit → unit⟨μ0⟩) → unit⟨μ1⟩
ignore = fun f → val ();;

g : unit → unit⟨Print | μ⟩
g = fun u → Print(1);;

main : unit⟨μ⟩
main = ignore g
```

Código 5.12: `ignore.alef`

Estado

En el siguiente ejemplo se muestra el modelado de estado en *Alef*. En este caso, el estado corresponde a un número entero, y se definen las dos operaciones usuales para modificarlo: `Get` y `Set`.

El handler que maneja estas operaciones, `st`, toma como entrada una computación de tipo `nat` que puede invocar estos efectos y devuelve una función de naturales en naturales, removiendo ambas operaciones del tipo. Esta función toma el estado inicial y produce el resultado final de la computación.

Para hacer esto, `st` se define en el Código 5.13 de la siguiente manera. Los valores se manejan con una función que toma el estado de la computación y devuelve el valor tomado. La operación `Get` se maneja con una función que pasa el estado actual a la continuación, capturando el resultado de esta última en una función f, la cual finalmente aplica. En cuanto a `Set`, esta se maneja con una función similar a la anterior, pero que ahora al llamar la función `f` resultante de invocar la continuación, se pasa el valor tomado por la operación.

Este programa maneja con `st` una secuencia de computaciones en las que primero se pone en 1 el estado, que luego es recuperado y retornado en la variable `y`. Al hacerlo, se guarda en `comp` la función resultante, que finalmente es invocada con el estado inicial 0. Como es de esperar, al ejecutar el programa se obtiene el valor 1.

No determinismo

Por último, se toma como ejemplo el modelado de la elección no determinista. Para introducir este efecto, es necesario declarar la operación `Choose`, que devuelve un booleano que representa la elección no determinista.

En el Código 5.14 se presenta un programa que hace uso de esta operación. Este la invoca dos veces y retorna un número natural que corresponde a la cantidad de veces que esta devolvió `true`. Adicionalmente, se define un handler sencillo `alwaysTrue`, que interpreta la elección haciendo que la operación siempre devuelva `true`, con el que se maneja el programa anterior. Al ejecutarlo, entonces, el valor retornado es 2.

```
signature {
    Get: unit → nat
  , Set: nat → unit
}

st : nat⟨Get, Set | μ⟩ ↠ (nat  →  nat⟨μ⟩)⟨μ⟩
st = handler
        val x    → val (fun s → val x),
        {Get u k → val (fun s → let f = k s in f s),
         Set x k → val (fun s → let f = k () in f x)};;

main : nat⟨μ⟩
main =  let comp = with st handle
                    (let x = Set(1) in
                     let y = Get() in
                     val y)
        in comp 0
```

Código 5.13: `state.alef`

Además de interpretar la operación de elección con una constante, se podría querer implementar un intercalamiento de decisiones, haciendo que el valor booleano retornado cambie cada vez que esta es ejecutada.

Para lograr esto, se hará uso del efecto Estado de la sección anterior, pero en este caso se llevará un estado booleano. Este efecto será lo que se conoce como *efecto intermediario*, ya que cada invocación a `Choose` se transformará en una computación que llama a `Get` y `Set`. La idea será que, en cada invocación en el programa de la forma:

```
...
let b = Choose() in
...
```

Se convierta a:

```
...
let b  = Get() in
let nb = not b in
let u  = Set(nb) in
...
```

De esta manera, el estado lleva el valor que será devuelto en la próxima invocación a `Choose`, y cada vez que se la llama se guarda en el estado el valor opuesto.

Esta transformación es capturada en el handler `choiceToSt`, presente en el Código 5.15. Para implementar la interpretación buscada, entonces, se componen el handler anterior y el que maneja las operaciones de estado, consiguiendo la computación

```
signature {
    Choose: unit → bool
}

alwaysTrue : nat⟨Choose | μ⟩ ↠ nat⟨μ⟩
alwaysTrue = handler
                val x → val x,
                {Choose x k → k true};;

main : nat⟨μ⟩
main = with alwaysTrue handle
        (let c1 = Choose() in
         let c2 = Choose() in
         if c1
            then (if c2 then val 2 else val 1)
            else (if c2 then val 1 else val 0)
        )
```

Código 5.14: `choice1.alef`

final parametrizada por el valor del estado inicial. En este caso, al intercalarse los valores booleanos, para cualquier estado inicial el programa retorna el valor 1.

5.4. Testing

La implementación de *Alef* se validó mediante dos técnicas de testing. Por un lado, se escribieron múltiples programas completos como ejemplo de distintas características y limitaciones de este lenguaje, en particular del sistema de tipos. Estos se encuentran en el directorio `examples`, donde se dividen en los casos que deben ser aceptados y los que deben ser rechazados. El programa `Unit` del directorio `test` se encarga de ejecutarlos y verificar que el resultado sea el esperado.

Por otro lado, el sistema también se validó de una manera sistematizada mediante la herramienta *QuickCheck* [7]. Para ello, en el módulo `QCInstances` se definen instancias que permiten generar términos *Alef* aleatorios con sus respectivos tipos. Esto se hace generando primero un tipo al azar, y en base a este se genera un término que lo habite.

Si bien esto es útil para validar el parser (`Parsing`) y el evaluador (`Stepper`), el principal propósito por el cual se decidió utilizar esta herramienta es validar las propiedades de seguridad planteadas en el Capítulo 3 (`Safety`). Esta es la razón por la cual solo se considera la generación de términos tipados, ya que esta condición es parte de la hipótesis de ambas propiedades.

El uso de ambas técnicas sirvió tanto para encontrar errores propios de la implementación como para detectar problemas de las primeras versiones del sistema de

```
signature {
    Choose : unit → bool
  , Get    : unit → bool
  , Set    : bool → unit
}

st : nat⟨Get, Set | μ⟩ ↠ (bool  →  nat⟨μ⟩)⟨μ⟩
st = handler
        val x     → val (fun s → val x),
        {Get u k  → val (fun s → let f = k s  in f s),
         Set x k  → val (fun s → let f = k () in f x)};;

not : bool → bool⟨μ⟩
not = fun b → if b then val false else val true;;

choiceToSt : nat⟨Choose | μ⟩ ↠ nat⟨Get, Set | μ⟩
choiceToSt = handler
                val x → val x,
                {Choose x k → let b = Get()   in
                              let nb = not b  in
                              let u = Set(nb) in
                              k b};;

main : nat⟨μ⟩
main =  let run = with st handle (with choiceToSt handle
           (let c1 = Choose() in
            let c2 = Choose() in
            if c1
                then (if c2 then val 2 else val 1)
                else (if c2 then val 1 else val 0)
            ))
        in run false
```

Código 5.15: choice2.alef

tipos. Es decir, el testing se utilizó como un paso previo a la demostración formal de seguridad.

Capítulo 6

Demostraciones

A continuación se desarrollan las demostraciones de los resultados presentados en el Capítulo 3.

6.1. Seguridad

En primer lugar se elabora la prueba de la seguridad del sistema de tipos declarativo. Esta depende de dos lemas principales (Preservación y Progreso) que a su vez dependen de lemas auxiliares.

Preservación

Para la preservación se introducen los tres lemas usuales: Exchange, Weakening y el de Sustitución. En estos se notará con las letras α y β tanto a los tipos como los esquemas de tipo.

El lema de Exchange, o *intercambio*, simplemente establece que es posible cambiar el orden de las variables en el contexto de tipado sin alterar el tipo de un término.

Lema 3 (Exchange)**.** $\forall\ \Gamma,\ x,\ y,\ \alpha,\ \beta,\ t,\ \mathcal{T}$
$\Gamma,\ x : \alpha,\ y : \beta \vdash t : \mathcal{T} \longrightarrow \Gamma,\ y : \beta,\ x : \alpha \vdash t : \mathcal{T}$

Demostración. Simple inducción estructural sobre la derivación. □

Weakening, o *debilitamiento*, determina que todo término tipado en un cierto contexto también puede ser tipado extendiendo el contexto con una variable nueva.

Lema 4 (Weakening)**.** $\forall \Gamma,\ t,\ \mathcal{T},\ x,\ \alpha\ \ (x \notin dom(\Gamma) \land \Gamma \vdash t : \mathcal{T} \longrightarrow \Gamma,\ x : \alpha \vdash t : \mathcal{T})$

Demostración. Simple inducción estructural sobre la derivación. □

El Lema de sustitución, por su parte, establece que al reemplazarse en un término una variable por una expresión del tipo adecuado, su tipo sigue siendo el mismo.

Lema 5 (Sustitución)**.** $\forall\, \Gamma,\ x,\ A,\ t,\ \mathcal{T}, e$

$$\begin{aligned} \Gamma,\ x : A \vdash t : \mathcal{T} \wedge \Gamma \vdash e : A \quad &\longrightarrow \Gamma \vdash t[x \mapsto e] : \mathcal{T} \wedge && (1)\\ \Gamma,\ x : \forall\overline{\mu}.A \vdash t : \mathcal{T} \wedge \Gamma \vdash e : A &\longrightarrow \Gamma \vdash t[x \mapsto e] : \mathcal{T} && (2) \end{aligned}$$

Demostración. Por inducción en la derivación de tipado de t. Sea la última regla aplicada:

Unit$^{\mathbf{D}}$. Trivial.
Var$^{\mathbf{D}}$. Se supone $\Gamma,\ x : A \vdash y : B$ (con $y : B \in \Gamma,\ x : A$) y $\Gamma \vdash e : A$.
Si $x = y$, luego también se tiene $B = A \wedge y[x \mapsto e] = e$. Aplicando estas igualdades sobre la última hipótesis, se alcanza que $\Gamma \vdash y[x \mapsto e] : B$.
Si $x \neq y$, se tiene en cambio que $y[x \mapsto e] = y \wedge y : B \in \Gamma$. Luego, por Var$^{\text{D}}$, $\Gamma \vdash y[x \mapsto e] : B$.
El caso (2) es trivial, puesto que las variables deben ser distintas.

PVar$^{\mathbf{D}}$. Se supone $\Gamma,\ x : \forall\overline{\mu}.A \vdash y : B$ (con $y : \forall\overline{\mu}.B' \in \Gamma,\ x : \forall\overline{\mu}.A$) y $\Gamma \vdash e : A$.
Si $x = y$, luego también se tiene $B' = A$, $B = A[\theta]$ para algún θ y $y[x \mapsto e] = e$. Como θ solo se define sobre las variable cuantificadas universalmente en el esquema de tipo, se tiene que $\Gamma[\theta] = \Gamma$ y $e[\theta] = e$.
Luego, por Lema 6, $\Gamma \vdash y[x \mapsto e] : A[\theta]$
Si $x \neq y$, el caso es análogo a la regla anterior.
El caso (1) es trivial, puesto que las variables deben ser distintas.

En los casos siguientes se prueba (1), puesto que el razonamiento para (2) es análogo.
Val$^{\mathbf{D}}$. Se supone $\Gamma,\ x : A \vdash \texttt{val}\ e_1 : B\langle\mathcal{E}\rangle$ y $\Gamma \vdash e : A$.
Por hipótesis inductiva, se tiene que $\Gamma \vdash e_1[x \mapsto e] : B$. Aplicando a esto Val$^{\text{D}}$, se alcanza finalmente que $\Gamma \vdash (\texttt{val}\ e_1)[x \mapsto e] : B\langle\mathcal{E}\rangle$.

Fun$^{\mathbf{D}}$. Se tiene como hipótesis que la derivación de t tiene la forma:

$$\frac{\Gamma,\ x : A,\ y : B \vdash c : \underline{C}}{\Gamma,\ x : A \vdash \texttt{fun}\, y \mapsto c : B \to \underline{C}}\ \text{Fun}^{\text{D}}$$

y que $\Gamma \vdash e : A$. Además, se supone que $x \neq y \wedge y \notin \text{dom}(\Gamma)$.
Aplicando el Lema 3 sobre la hipótesis de la derivación, se alcanza

$$\Gamma,\ y : B,\ x : A \vdash c : \underline{C}$$

Por hipótesis y Lema 4, se tiene $\Gamma,\ y : B \vdash e : A$. Finalmente, aplicando la hipótesis inductiva sobre estas últimas dos, se obtiene $\Gamma,\ y : B \vdash c[x \mapsto e] : \underline{C}$ y, por Fun$^{\text{D}}$, vale $\Gamma \vdash (\texttt{fun}\, y \mapsto c)[x \mapsto e] : B \to \underline{C}$.

App$^{\mathbf{D}}$. Se tiene

$$\frac{\Gamma,\ x : A \vdash e_1 : B \to \underline{C} \qquad \Gamma,\ x : A \vdash e_2 : B}{\Gamma,\ x : A \vdash e_1\ e_2 : \underline{C}}\ \text{App}^{\text{D}}$$

y $\Gamma \vdash e : A$.
Aplicando la hipótesis inductiva sobre las sub-derivaciones, se tiene:

$$\Gamma \vdash e_1[x \mapsto e] : B \to \underline{C} \wedge \Gamma \vdash e_2[x \mapsto e] : B$$

Usando estas dos como hipótesis de App$^{\mathrm{D}}$, se alcanza $\Gamma \vdash (e_1\ e_2)[x \mapsto e] : \underline{C}$.

Op$^{\mathrm{D}}$. Se tiene que la derivación de t tiene la forma

$$\frac{\mathtt{Op} : A_{op} \to B_{op} \in \Sigma \qquad \Gamma,\ x : A \vdash e_1 : A_{op} \qquad \Gamma,\ x : A,\ y : B_{op} \vdash c : C\langle\mathcal{E}\rangle \qquad \mathtt{Op} \in \mathcal{E}}{\Gamma,\ x : A \vdash \mathtt{Op}\ e_1\ (y.c) : C\langle\mathcal{E}\rangle}\ \textsc{Op}^{\mathrm{D}}$$

y que $\Gamma \vdash e : A$. Además, se supone que $x \neq y \wedge y \notin \mathrm{dom}(\Gamma)$.
Aplicando la hipótesis inductiva sobre la primera sub-derivación, se obtiene
$\Gamma \vdash e_1[x \mapsto e] : A_{op}$. Con un razonamiento similar al de Fun$^{\mathrm{D}}$, se puede aplicar *Exchange* a la otra sub-derivación y *Weakening* al tipado de e para poder utilizar la hipótesis inductiva y alcanzar $\Gamma,\ y : B_{op} \vdash c[x \mapsto e] : C\langle\mathcal{E}\rangle$.
Con estos resultados, se puede finalmente aplicar Op$^{\mathrm{D}}$ y obtener

$$\Gamma \vdash (\mathtt{Op}\ e_1\ (y.c))[x \mapsto e] : C\langle\mathcal{E}\rangle$$

Anno$^{\mathrm{D}}$. Se tiene que la derivación tiene la forma

$$\frac{\Gamma,\ x : A \vdash t : \mathcal{T}}{\Gamma,\ x : A \vdash (t : \mathcal{T}) : \mathcal{T}}\ \textsc{Anno}^{\mathrm{D}}$$

y que $\Gamma \vdash e : A$.
Aplicando la hipótesis inductiva sobre la sub-derivación obtiene que:

$$\Gamma \vdash t[x \mapsto e] : \mathcal{T}$$

Luego, aplicando Anno$^{\mathrm{D}}$, se tiene que

$$\Gamma \vdash (t : \mathcal{T})[x \mapsto e] : \mathcal{T}$$

With$^{\mathrm{D}}$. Razonamiento análogo al caso App$^{\mathrm{D}}$.

Hand$^{\mathrm{D}}$. Razonamiento análogo al caso Fun$^{\mathrm{D}}$.

Let$^{\mathrm{D}}$. Razonamiento análogo al caso Op$^{\mathrm{D}}$.

□

En base a estos lemas, se prueba la Preservación de tipos.

Lema 1 (Preservación)**.** *$\forall\, \Gamma, \mathcal{T}, t$ y t' tales que $\Gamma \vdash t : \mathcal{T}$ y $t \rightsquigarrow t'$, vale que $\Gamma \vdash t' : \mathcal{T}$.*

Demostración. Por inducción en $t \rightsquigarrow t'$. Si el último paso de evaluación es:

- $(\texttt{fun}\ x \mapsto c)\ e \rightsquigarrow c[x \mapsto e]$. Por hipótesis, deben existir tipos A y $\underline{C}$ tales que valga la siguiente derivación de tipado:

$$\dfrac{\dfrac{\overset{\textcircled{1}}{\Gamma,\ x : A \vdash c : \underline{C}}}{\Gamma \vdash \texttt{fun}\ x \mapsto c : A \to \underline{C}}\ \textsc{Fun}^{\mathrm{D}} \qquad \overset{\textcircled{2}}{\Gamma \vdash e : A}}{\Gamma \vdash (\texttt{fun}\ x \mapsto c)\ e : \underline{C}}\ \textsc{App}^{\mathrm{D}}$$

  Por ①, ② y el Lema de Sustitución, se tiene que

$$\Gamma \vdash c[x \mapsto e] : \underline{C}$$

- $\texttt{with}\ h\ \texttt{handle}\ (\texttt{val}\ e) \rightsquigarrow c_v[x \mapsto e]$. Similarmente al caso anterior, deben existir los tipos $\underline{A}$ y $\underline{B}$ para los que vale:

$$\dfrac{\dfrac{\overset{\textcircled{1}}{\Gamma,\ x : A \vdash c_v : \underline{B}} \quad \dots}{\Gamma \vdash h : \underline{A} \twoheadrightarrow \underline{B}}\ \textsc{Hand}^{\mathrm{D}} \qquad \dfrac{\overset{\textcircled{2}}{\Gamma \vdash e : A}}{\Gamma \vdash \texttt{val}\ e : \underline{A}}\ \textsc{Val}^{\mathrm{D}}}{\Gamma \vdash \texttt{with}\ h\ \texttt{handle}\ (\texttt{val}\ e) : \underline{B}}\ \textsc{With}^{\mathrm{D}}$$

  Por ①, ② y el Lema de Sustitución, se tiene que

$$\Gamma \vdash c_v[x \mapsto e] : \underline{B}$$

- $\texttt{with}\ h\ \texttt{handle}\ (\texttt{Op}_j\ e\ y.c) \rightsquigarrow c_j[x \mapsto e, k \mapsto \texttt{fun}\ y \mapsto \texttt{with}\ h\ \texttt{handle}\ c]$, con $\texttt{Op}_j \in \{\texttt{Op}_i\}_{i=1}^{k}$. Además, deben existir tipos $A\langle\mathcal{E}\rangle$ y $B\langle\mathcal{E}'\rangle$ para los que vale:

$$\dfrac{\dfrac{\overset{\textcircled{5}}{\Gamma,\ x : A_j,\ k : B_j \to B\langle\mathcal{E}'\rangle \vdash c_j : \underline{B}} \quad \dots}{\textcircled{2}\ \ \Gamma \vdash h : A\langle\mathcal{E}\rangle \twoheadrightarrow B\langle\mathcal{E}'\rangle}\ \textsc{Hand}^{\mathrm{D}} \qquad \dfrac{\overset{\textcircled{4}}{\Gamma \vdash e : A_j} \qquad \overset{\textcircled{1}}{\Gamma,\ y : B_j \vdash c : A\langle\mathcal{E}\rangle}}{\Gamma \vdash \texttt{Op}_j\ e\ y.c : A\langle\mathcal{E}\rangle}\ \textsc{Op}^{\mathrm{D}}}{\Gamma \vdash \texttt{with}\ h\ \texttt{handle}\ (\texttt{Op}_j\ e\ y.c) : B\langle\mathcal{E}'\rangle}\ \textsc{With}^{\mathrm{D}}$$

  Por ① se tiene $\Gamma,\ y : B_j \vdash c : A\langle\mathcal{E}\rangle$. Esto junto a ② y la regla $\textsc{With}^{\mathrm{D}}$ garantiza que valga

$$\Gamma,\ y : B_j \vdash \texttt{with}\ h\ \texttt{handle}\ c : B\langle\mathcal{E}'\rangle$$

  Aplicando $\textsc{Fun}^{\mathrm{D}}$, se alcanza

$$\textcircled{3}\ \ \Gamma \vdash \texttt{fun}\ y \mapsto \texttt{with}\ h\ \texttt{handle}\ c : B_j \to B\langle\mathcal{E}'\rangle$$

  Por último, aplicando el Lema de Sustitución sobre ⑤ dos veces, primero con ③ y luego con ④, se obtiene

$$\Gamma \vdash c_j[x \mapsto e, k \mapsto \texttt{fun}\ y \mapsto \texttt{with}\ h\ \texttt{handle}\ c] : B\langle\mathcal{E}'\rangle$$

- $\texttt{with}\ h\ \texttt{handle}\ (\texttt{Op}_j\ e\ y.c) \rightsquigarrow \texttt{Op}_j\ e\ (y.\texttt{with}\ h\ \texttt{handle}\ c)$, con $\texttt{Op}_j \notin \{\texttt{Op}_i\}_{i=1}^k$. Por hipótesis, deben existir tipos $A\langle\mathcal{E}\rangle$ y $B\langle\mathcal{E}'\rangle$, tales que valga la derivación:

$$\dfrac{\overset{②}{\Gamma \vdash h : A\langle\mathcal{E}\rangle \twoheadrightarrow B\langle\mathcal{E}'\rangle} \qquad \dfrac{\overset{①}{\Gamma \vdash e : A_j} \qquad \overset{③}{\Gamma,\ y : B_j \vdash c : A\langle\mathcal{E}\rangle}}{\Gamma \vdash \texttt{Op}_j\ e\ y.c : A\langle\mathcal{E}\rangle}\ \textsc{Op}^{\mathrm{D}}}{\Gamma \vdash \texttt{with}\ h\ \texttt{handle}\ (\texttt{Op}_j\ e\ y.c) : B\langle\mathcal{E}'\rangle}\ \textsc{With}^{\mathrm{D}}$$

Además, por $\textsc{Hand}^{\mathrm{D}}$, $\texttt{Op}_j \in \mathcal{E}'$. Luego, es posible construir la siguiente derivación:

$$\dfrac{\overset{①}{\Gamma \vdash e : A_j} \qquad \dfrac{\overset{② + \text{Weakening}}{\Gamma,\ y : B_j \vdash h : A\langle\mathcal{E}\rangle \twoheadrightarrow B\langle\mathcal{E}'\rangle} \qquad \overset{③}{\Gamma,\ y : B_j \vdash c : A\langle\mathcal{E}\rangle}}{\Gamma,\ y : B_j \vdash \texttt{with}\ h\ \texttt{handle}\ c : B\langle\mathcal{E}'\rangle}\ \textsc{With}^{\mathrm{D}}}{\Gamma \vdash \texttt{Op}_j\ e\ (y.\texttt{with}\ h\ \texttt{handle}\ c) : B\langle\mathcal{E}'\rangle}\ \textsc{Op}^{\mathrm{D}}$$

- $\texttt{with}\ h\ \texttt{handle}\ c \rightsquigarrow \texttt{with}\ h\ \texttt{handle}\ c'$ con $c \rightsquigarrow c'$. Por hipótesis, deben existir tipos $\underline{A}$ y $\underline{B}$ tales que valga:

$$\dfrac{\overset{①}{\Gamma \vdash h : \underline{A} \twoheadrightarrow \underline{B}} \qquad \overset{②}{\Gamma \vdash c : \underline{A}}}{\Gamma \vdash \texttt{with}\ h\ \texttt{handle}\ c : \underline{B}}\ \textsc{With}^{\mathrm{D}}$$

Aplicando la hipótesis inductiva sobre ②, se obtiene que $\Gamma \vdash c' : \underline{A}$. Esto último, junto a ①, indica por $\textsc{With}^{\mathrm{D}}$ que $\Gamma \vdash \texttt{with}\ h\ \texttt{handle}\ c' : \underline{B}$.

- $\texttt{let}\ x := (\texttt{val}\ e)\ \texttt{in}\ c_2 \rightsquigarrow c_2[x \mapsto e]$. Por hipótesis, se tiene una derivación de la forma:

$$\dfrac{\dfrac{\overset{①}{\Gamma \vdash e : A}}{\Gamma \vdash \texttt{val}\ e : A\langle\mathcal{E}\rangle}\ \textsc{Val}^{\mathrm{D}} \qquad \overset{②}{\Gamma,\ x : \forall\overline{\mu}.A \vdash c_2 : B\langle\mathcal{E}\rangle}}{\Gamma \vdash \texttt{let}\ x := \texttt{val}\ e\ \texttt{in}\ c_2 : B\langle\mathcal{E}\rangle}\ \textsc{Let}^{\mathrm{D}}$$

Aplicando el Lema de Sustitución sobre ① y ② se obtiene

$$\Gamma \vdash c_2[x \mapsto e] : B\langle\mathcal{E}\rangle$$

- $\texttt{let}\ x := \texttt{Op}_j\ e\ (y.c_1)\ \texttt{in}\ c_2 \rightsquigarrow \texttt{Op}_j\ e\ (y.\texttt{let}\ x := c_1\ \texttt{in}\ c_2)$. Por hipótesis, deben existir tipos $A\langle\mathcal{E}\rangle$ y $B\langle\mathcal{E}\rangle$ que hagan valer la derivación:

$$\dfrac{\dfrac{\overset{①}{\Gamma \vdash e : A_j} \qquad \overset{②}{\Gamma,\ y : B_j \vdash c_1 : A\langle\mathcal{E}\rangle} \qquad \texttt{Op}_j \in \mathcal{E}}{\Gamma \vdash \texttt{Op}_j\ e\ (y.c_1) : A\langle\mathcal{E}\rangle}\ \textsc{Op}^{\mathrm{D}} \qquad \overset{③}{\Gamma,\ x : \forall\overline{\mu}.A \vdash c_2 : B\langle\mathcal{E}\rangle}}{\Gamma \vdash \texttt{let}\ x := \texttt{Op}_j\ e\ (y.c_1)\ \texttt{in}\ c_2 : B\langle\mathcal{E}\rangle}\ \textsc{Let}^{\mathrm{D}}$$

Luego, se puede construir la siguiente derivación para el término resultante del paso de evaluación:

$$
\dfrac{\dfrac{①}{\Gamma \vdash e : A_j} \qquad \dfrac{\dfrac{②}{\Gamma,\ y : B_j \vdash c_1 : A\langle\mathcal{E}\rangle} \qquad \dfrac{③ + \text{Weak.} + \text{Exch.}}{\Gamma,\ y : B_j,\ x : \forall\overline{\mu}.A \vdash c_2 : B\langle\mathcal{E}\rangle}}{\Gamma,\ y : B_j \vdash \texttt{let}\ x \coloneqq\ c_1\ \texttt{in}\ c_2 : B\langle\mathcal{E}\rangle}\ \textsc{Let}^{\mathrm{D}} \qquad \texttt{Op}_j \in \mathcal{E}}{\Gamma \vdash \texttt{Op}_j\ e\ (y.\texttt{let}\ x \coloneqq\ c_1\ \texttt{in}\ c_2) : B\langle\mathcal{E}\rangle}\ \textsc{Op}^{\mathrm{D}}
$$

- El razonamiento para la regla de congruencia de `let` es análogo al de la de `with`.
- $(\texttt{fun}\ x \mapsto c : A \to \underline{C})\ e \rightsquigarrow c[x \mapsto \utilde{e : A}]$. Por hipótesis, debe existir una instanciación σ para la que valga la siguiente derivación de tipado:

$$
\dfrac{\dfrac{\dfrac{\dfrac{①}{\Gamma,\ x : A[\sigma] \vdash c : \underline{C}[\sigma]}}{\Gamma \vdash \texttt{fun}\ x \mapsto c : (A \to \underline{C})[\sigma]}\ \textsc{Fun}^{\mathrm{D}}}{\Gamma \vdash (\texttt{fun}\ x \mapsto c : A \to \underline{C}) : (A \to \underline{C})[\sigma]}\ \textsc{Anno}^{\mathrm{D}} \qquad \dfrac{②}{\Gamma \vdash e : A[\sigma]}}{\Gamma \vdash (\texttt{fun}\ x \mapsto c : A \to \underline{C})\ e : \underline{C}[\sigma]}\ \textsc{App}^{\mathrm{D}}
$$

Es simple ver que, por ②, se tiene que

$$\Gamma \vdash (\utilde{e : A}) : A[\sigma]$$

Por esto último, ① y el Lema de Sustitución, se tiene que

$$\Gamma \vdash c[x \mapsto \utilde{e : A}] : \underline{C}[\sigma]$$

El resto de las reglas que manejan anotaciones de tipos se extienden de sus correspondientes sin anotaciones de manera análoga a la anterior. □

Progreso

La segunda parte de la prueba de seguridad consiste en la demostración de Progreso, presentada a continuación.

Lema 2 (Progreso)**.** *Para toda computación cerrada c y todo tipo $C\langle\mathcal{E}\rangle$ tales que $\cdot \vdash c : C\langle\mathcal{E}\rangle$, vale una de las siguientes:*

- $\exists v\ (c = \texttt{val}\ v \land \cdot \vdash v : C)$.
- $\exists \texttt{Op}_j, e, k\ (c = \texttt{Op}_j\ e\ k \land \texttt{Op}_j \in \mathcal{E})$.
- $\exists c'\ (c \rightsquigarrow c')$

Demostración. Por inducción en la derivación de tipado de c. Sea la última regla aplicada:

VAL$^{\mathbf{D}}$. Se supone que la derivación de tipado tiene la forma

$$\frac{\cdot \vdash e : C}{\cdot \vdash \texttt{val}\ e : C\langle\mathcal{E}\rangle}\ \textsc{Val}^{\mathrm{D}}$$

Luego, $c = \texttt{val}\ e$ y $\cdot \vdash e : C$.

APP$^{\mathbf{D}}$. Se supone que la derivación de tipado tiene la forma

$$\frac{\cdot \vdash e_1 : A \to \underline{C} \qquad \cdot \vdash e_2 : A}{\cdot \vdash e_1\ e_2 : \underline{C}}\ \textsc{App}^{\mathrm{D}}$$

Por inspección de las reglas y al ser un término cerrado, es claro que e_1 tiene una de las dos siguientes formas:

- $\texttt{fun}\ x \mapsto c$ para algún x, c. Luego, $\exists c' = c[x \mapsto e_2]$ $((\texttt{fun}\ x \mapsto c)\ e_2 \rightsquigarrow c')$.
- $(\texttt{fun}\ x \mapsto c : A' \to \underline{C'})$ para algunos $x, c, A, \underline{C'}$ y θ tales que $(A' \to \underline{C'})[\theta] = A \to \underline{C}$. Luego, $\exists c' = c[x \mapsto \underset{\sim}{e_2 : A'}]$ $(e_1\ e_2 \rightsquigarrow c')$.

OP$^{\mathbf{D}}$. Se supone que la derivación de tipado tiene la forma

$$\frac{\texttt{Op} : A_{op} \to B_{op} \in \Sigma \qquad \cdot \vdash e : A_{op} \qquad \cdot, y : B_{op} \vdash c : C\langle\mathcal{E}\rangle \qquad \texttt{Op} \in \mathcal{E}}{\cdot \vdash \texttt{Op}\ e\ (y.c) : C\langle\mathcal{E}\rangle}\ \textsc{Op}^{\mathrm{D}}$$

Este caso es trivial, ya que vale la segunda cláusula.

WITH$^{\mathbf{D}}$. Se supone que la derivación de tipado tiene la forma

$$\frac{\cdot \vdash h : A\langle\mathcal{E}\rangle \twoheadrightarrow B\langle\mathcal{E}'\rangle \qquad \cdot \vdash c : A\langle\mathcal{E}\rangle}{\cdot \vdash \texttt{with}\ h\ \texttt{handle}\ c : B\langle\mathcal{E}'\rangle}\ \textsc{With}^{\mathrm{D}}$$

Por inspección de las reglas y al ser un término cerrado, es claro que h puede tener la forma $\texttt{handler}\ \texttt{val}\ x \mapsto c_v,\ \{\texttt{Op}_i\ x\ k \mapsto c_i\}_{i=1}^{n}$ o esta misma con una anotación de tipo. Se muestra el primera caso, ya que el segundo es análogo con las reglas de evaluación para aplicación de handlers anotados.
Aplicando la hipótesis inductiva sobre la sub-derivación de c, se obtiene que debe valer una de las siguientes:

- $\exists v\ (c = \texttt{val}\ v \wedge \cdot \vdash v : A)$. En este caso, se tiene que

$$\exists c' = c_v[x \mapsto v]\ (\texttt{with}\ h\ \texttt{handle}\ c \rightsquigarrow c')$$

- $\exists \mathtt{Op}_j, e, y, c_1\ (c = \mathtt{Op}_j\ e\ (y.c_1) \land \mathtt{Op}_j \in \mathcal{E})$.
  Si $\mathtt{Op}_j \in \{\mathtt{Op}_i\}_{i=1}^{k}$:

  $$\exists c' = c_j[x \mapsto e,\ k \mapsto \mathtt{fun}\, y \mapsto \mathtt{with}\, h\, \mathtt{handle}\, c_1]\ (\mathtt{with}\, h\, \mathtt{handle}\, c \rightsquigarrow c')$$

  Si $\mathtt{Op}_j \notin \{\mathtt{Op}_i\}_{i=1}^{k}$:

  $$\exists c' = \mathtt{Op}_j\ e\ (y.\mathtt{with}\ h\ \mathtt{handle}\ c_1)\ (\mathtt{with}\ h\ \mathtt{handle}\ c \rightsquigarrow c')$$

- $\exists c'\ (c \rightsquigarrow c')$. En este caso, se tiene que

  $$\mathtt{with}\ h\ \mathtt{handle}\ c \rightsquigarrow \mathtt{with}\ h\ \mathtt{handle}\ c'$$

Let$^{\mathrm{D}}$. Se supone que la derivación de tipado tiene la forma

$$\frac{\cdot \vdash c_1 : A\langle\mathcal{E}\rangle \qquad \cdot,\ x : \forall\overline{\mu}.A \vdash c_2 : B\langle\mathcal{E}\rangle}{\cdot \vdash \mathtt{let}\ x \coloneqq\ c_1\ \mathtt{in}\ c_2 : B\langle\mathcal{E}\rangle}\ \textsc{Let}^{\mathrm{D}}$$

Aplicando la hipótesis inductiva sobre la sub-derivación de c_1, se obtiene que debe valer una de las siguientes:

- $\exists v\ (c_1 = \mathtt{val}\ v \land \cdot \vdash v : A)$. En este caso, se tiene que

  $$\exists c' = c_2[x \mapsto v]\ (\mathtt{let}\ x \coloneqq\ c_1\ \mathtt{in}\ c_2 \rightsquigarrow c')$$

- $\exists \mathtt{Op}_j, e, y, c_3\ (c_1 = \mathtt{Op}_j\ e\ (y.c_3) \land \mathtt{Op}_j \in \mathcal{E})$.
  Luego,

  $$\exists c' = \mathtt{Op}_j\ e\ (y.\mathtt{let}\ x \coloneqq\ c_3\ \mathtt{in}\ c_2)\ (\mathtt{let}\ x \coloneqq\ c_1\ \mathtt{in}\ c_2 \rightsquigarrow c')$$

- $\exists c_1'\ (c_1 \rightsquigarrow c_1')$. En este caso, se tiene que

  $$\mathtt{let}\ x \coloneqq\ c_1\ \mathtt{in}\ c_2 \rightsquigarrow \mathtt{let}\ x \coloneqq\ c_1'\ \mathtt{in}\ c_2$$

□

6.2. Correspondencia

Como se estableció en el Capítulo 3, para extender los resultados sobre la seguridad del sistema declarativo al bidireccional, se necesita de una correspondencia entre estos. Esta relación se da en base a los dos teoremas que se prueban en esta sección.

Consistencia

El primero de estos, Consistencia, permite pasar del sistema bidireccional al declarativo. Su prueba depende del lema Sub, de *subsumption*, permite hacer más gruesos los tipos asignados por el sistema declarativo.

Lema 6 (Sub)**.** $\forall \Gamma,\ t,\ \mathcal{T},\ \omega\ \ (\Gamma \vdash t : \mathcal{T} \longrightarrow \Gamma[\omega] \vdash t[\omega] : \mathcal{T}[\omega])$

Demostración. Por inducción en la derivación de tipado. Sea la última regla aplicada:

Var$^{\mathbf{D}}$. Se supone $\Gamma \vdash x : A$ con $x : A \in \Gamma$.
Luego, $x : A[\omega] \in \Gamma[\omega]$ y, por lo tanto, $\Gamma[\omega] \vdash x : A[\omega]$.

PVar$^{\mathbf{D}}$. Se supone $\Gamma \vdash f : A[\theta]$ con $f : \forall \overline{\mu}.A \in \Gamma$.
Luego, $f : \forall \overline{\mu}.A[\omega] \in \Gamma[\omega]$, donde ω solo se aplica sobre las variables de efectos no cuantificadas. Por lo tanto, $\Gamma[\omega] \vdash f : A[\omega\theta']$. Como θ' y θ solo instancian $\overline{\mu}$ y son arbitrarias, se puede obtener una θ' tal que $\Gamma[\omega] \vdash f : A[\theta\omega]$.

Val$^{\mathbf{D}}$. Se supone $\Gamma \vdash \texttt{val}\ e : A\langle\mathcal{E}\rangle$.
Por hipótesis inductiva, se tiene que $\Gamma[\omega] \vdash e[\omega] : A[\omega]$. Luego, aplicando sobre esto último la regla Val$^{\mathrm{D}}$, se obtiene que $\Gamma[\omega] \vdash \texttt{val}\ e[\omega] : A\langle\mathcal{E}\rangle[\omega]$.

Fun$^{\mathbf{D}}$. Se supone

$$\frac{\Gamma,\ x : A \vdash c : \underline{C}}{\Gamma \vdash \texttt{fun}\ x \mapsto c : A \to \underline{C}}\ \textsc{Fun}^{\mathrm{D}}$$

Por hipótesis inductiva, $(\Gamma,\ x\ :\ A)[\omega]\ \vdash\ c\ :\ \underline{C}[\omega]$. Luego, aplicándole la regla Fun$^{\mathrm{D}}$, se obtiene $\Gamma[\omega] \vdash \texttt{fun}\ x \mapsto c : (A \to \underline{C})[\omega]$.

App$^{\mathbf{D}}$. Se supone

$$\frac{\Gamma \vdash e_1 : A \to \underline{C} \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : \underline{C}}\ \textsc{App}^{\mathrm{D}}$$

Aplicando la hipótesis inductiva sobre las sub-derivaciones, se obtiene:

- $\Gamma[\omega] \vdash e_1[\omega] : (A \to \underline{C})[\omega]$.
- $\Gamma[\omega] \vdash e_2[\omega] : A[\omega]$.

Luego, al aplicar App$^{\mathrm{D}}$ sobre estos dos resultados, se alcanza que

$$\Gamma[\omega] \vdash e_1\ e_2[\omega] : \underline{C}[\omega]$$

Anno$^{\mathbf{D}}$. Se supone

$$\frac{\Gamma \vdash t : \mathcal{T}}{\Gamma \vdash (t : \mathcal{T}) : \mathcal{T}}\ \textsc{Anno}^{\mathrm{D}}$$

Aplicando la hipótesis inductiva sobre la sub-derivación, se tiene que

$$\Gamma[\omega] \vdash t[\omega] : \mathcal{T}[\omega]$$

Luego, aplicando Anno^{D}, se alcanza:

$$\Gamma[\omega] \vdash (t : \mathcal{T})[\omega] : \mathcal{T}[\omega]$$

Let^{D}. Se supone

$$\frac{\Gamma \vdash c_1 : A\langle\mathcal{E}\rangle \qquad \Gamma,\ x : \forall\overline{\mu}.A \vdash c_2 : B\langle\mathcal{E}\rangle}{\Gamma \vdash \texttt{let}\ x := c_1\ \texttt{in}\ c_2 : B\langle\mathcal{E}\rangle}\ \text{Let}^{D}$$

Aplicando la hipótesis inductiva sobre las sub-derivaciones, se obtiene:

- $\Gamma[\omega] \vdash c_1[\omega] : A\langle\mathcal{E}\rangle[\omega]$.
- $(\Gamma,\ x : A)[\omega] \vdash c_2[\omega] : B\langle\mathcal{E}\rangle[\omega]$.

Luego, al aplicar Let^{D} sobre estos dos resultados, se alcanza que

$$\Gamma[\omega] \vdash (\texttt{let}\ x := c_1\ \texttt{in}\ c_2)[\omega] : B\langle\mathcal{E}\rangle[\omega]$$

El resto de los casos son análogos.

□

Teorema 2 (Consistencia)**.** $\forall\ \Gamma, t, \mathcal{T}, \sigma, \eta$
$(\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \longrightarrow \Gamma[\sigma] \vdash t[\sigma] : \mathcal{T}[\sigma]) \land$
$(\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T} \longrightarrow \Gamma[\eta] \vdash t[\eta] : \mathcal{T})$

Demostración. Por inducción simultánea sobre los juicios de tipado. Si la última regla aplicada es:

Var. Se tiene que

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \overset{[]}{\Rightarrow} A}\ \text{Var}$$

Luego, $x : A \in \Gamma$ y, por Var^{D}, $\Gamma \vdash x : A$.

Val. Se tiene que

$$\frac{\Gamma \vdash e \overset{\eta}{\Rightarrow} A}{\Gamma \vdash \texttt{val}\ e \overset{\eta}{\Rightarrow} A\langle\mu\rangle}\ \text{Val}$$

Aplicando la hipótesis inductiva, se obtiene que $\Gamma[\eta] \vdash e[\eta] : A$. Por Val^{D}, se alcanza que $\Gamma[\eta] \vdash \texttt{val}\ e[\eta] : A\langle\mu\rangle$.

App. Se tiene que

$$\dfrac{\Gamma \vdash e_1 \overset{\eta}{\Rightarrow} A \to C\langle\mathcal{E}\rangle \qquad \Gamma[\eta] \vdash e_2 \overset{\sigma}{\Leftarrow} A}{\eta \vdash e_1\ e_2 \overset{\eta\sigma}{\Rightarrow} C\langle\mathcal{E}\rangle[\sigma]}\ \textsc{App}$$

Como las variables de las anotaciones son frescas, $e_2 = e_2[\eta]$. Aplicando la hipótesis inductiva sobre ambas sub-derivaciones, se obtienen:

(1) $\Gamma[\eta] \vdash e_1[\eta] : A \to C\langle\mathcal{E}\rangle$

(2) $\Gamma[\eta\sigma] \vdash e_2[\eta\sigma] : A[\sigma]$

Por (1) y el Lema 6 (Sub), se tiene que

$$\Gamma[\eta\sigma] \vdash e_1[\eta\sigma] : (A \to C\langle\mathcal{E}\rangle)[\sigma]$$

Finalmente, por $\textsc{Fun}^{\mathrm{D}}$, se alcanza

$$\Gamma[\eta\sigma] \vdash (e_1\ e_2)[\eta\sigma] : C\langle\mathcal{E}\rangle[\sigma]$$

Fun. Se tiene que

$$\dfrac{\Gamma,\ x : A \vdash c \overset{\sigma}{\Leftarrow} C\langle\mathcal{E}\rangle}{\Gamma \vdash \texttt{fun}\ x \mapsto c \overset{\sigma}{\Leftarrow} A \to C\langle\mathcal{E}\rangle}\ \textsc{Fun}$$

Aplicando la hipótesis inductiva sobre la sub-derivación, se tiene

$$(\Gamma,\ x : A)[\sigma] \vdash c[\sigma] : C\langle\mathcal{E}\rangle[\sigma]$$

Por $\textsc{Fun}^{\mathrm{D}}$, se alcanza

$$\Gamma[\sigma] \vdash \texttt{fun}\ x \mapsto c[\sigma] : (A \to C\langle\mathcal{E}\rangle)[\sigma]$$

Op. Se tiene que

$$\dfrac{\texttt{Op} : A_{op} \to B_{op} \in \Sigma \qquad \Gamma \vdash e \Leftarrow A_{op} \qquad (\Gamma,\ y : B_{op}) \vdash c \overset{\eta}{\Rightarrow} A\langle\mathcal{E}\rangle \qquad add(\mathcal{E}, \texttt{Op}) = \gamma}{\Gamma \vdash \texttt{Op}\ e\ (y.c) \overset{\eta\gamma}{\Rightarrow} A\langle\mathcal{E}\rangle[\gamma]}\ \textsc{Op}$$

Aplicando las hipótesis inductivas en las sub-derivaciones se obtiene:

(1) $\Gamma \vdash e : A_{op}$

(2) $(\Gamma,\ y : B_{op})[\eta] \vdash c[\eta] : A\langle\mathcal{E}\rangle$

Aplicando el Lema 6 (Sub) a ambas se alcanza:

(1') $\Gamma[\eta\gamma] \vdash e : A_{op}$

(2') $(\Gamma,\ y : B_{op})[\eta\gamma] \vdash c[\eta\gamma] : A\langle\mathcal{E}\rangle[\gamma]$

El tipo A_{op} no se instancia ya que debe ser un tipo base. Además, se sabe que $\mathtt{Op} \in \mathcal{E}[\gamma]$ por la definición de esta misma. Luego, por Op^{D}, se tiene

$$\Gamma[\eta\gamma] \vdash \mathtt{Op}\ e\ (y.c)[\eta\gamma] : A\langle\mathcal{E}\rangle[\gamma]$$

CS. Se tiene que

$$\frac{\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}' \qquad \mathcal{T}' \overset{\sigma}{\bowtie}_T \mathcal{T}[\eta]}{\Gamma \vdash t \overset{\eta\sigma}{\Leftarrow} \mathcal{T}}\ \text{CS}$$

Como $\mathcal{T}' \overset{\sigma}{\bowtie}_T \mathcal{T}[\eta]$, vale que $\mathcal{T}'[\sigma] = \mathcal{T}[\eta\sigma]$. Además, por hipótesis inductiva, se sabe que

$$\Gamma[\eta] \vdash t[\eta] : \mathcal{T}'$$

Aplicando sobre esto último el Lema 6 (Sub) y usando la igualdad planteada, se alcanza

$$\Gamma[\eta\sigma] \vdash t[\eta\sigma] : \mathcal{T}[\eta\sigma]$$

Anno. Se tiene que

$$\frac{\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \qquad \mathcal{T} \equiv \mathcal{T}[\sigma]}{\Gamma \vdash (t : \mathcal{T}) \overset{\sigma}{\Rightarrow} \mathcal{T}[\sigma]}\ \text{Anno}$$

Por hipótesis inductiva se sabe que

$$\Gamma[\sigma] \vdash t[\sigma] : \mathcal{T}[\sigma]$$

Luego, por Anno^{D}, se tiene que

$$\Gamma[\sigma] \vdash (t : \mathcal{T})[\sigma] : \mathcal{T}[\sigma]$$

Let. Se tiene que

$$\frac{\Gamma \vdash c_1 \overset{\eta_1}{\Rightarrow} A\langle\mathcal{E}_1\rangle \qquad \Gamma[\eta_1],\ x : \forall\overline{\mu}.A \vdash c_2 \overset{\eta_2}{\Rightarrow} B\langle\mathcal{E}_2\rangle \qquad \mathcal{E}_1[\eta_2] \overset{\sigma}{\bowtie}_R \mathcal{E}_2}{\Gamma \vdash \mathtt{let}\ x \coloneqq\ c_1\ \mathtt{in}\ c_2 \overset{\eta_1\eta_2\sigma}{\Rightarrow} B\langle\mathcal{E}_2\rangle[\sigma]}\ \text{Let}$$

Puesto que $\mathcal{E}_1[\eta_2] \overset{\sigma}{\bowtie}_R \mathcal{E}_2$, se sabe que $\mathcal{E}_1[\eta_2\sigma] = \mathcal{E}_2[\sigma]$. Aplicando la hipótesis inductiva y el Lema 6 (Sub), se alcanza:

(1) $\Gamma[\eta_1\eta_2\sigma] \vdash c_1[\eta_1\eta_2\sigma] : A\langle\mathcal{E}_1\rangle[\eta_2\sigma]$

(2) $\Gamma[\eta_1\eta_2\sigma],\ x : \forall\overline{\mu}.A[\eta_2\sigma] \vdash c_2[\eta_1\eta_2\sigma] : B\langle\mathcal{E}_2\rangle[\sigma]$

Luego, por LET^{D} y la igualdad presentada, se obitene que:

$$\Gamma[\eta_1\eta_2\sigma] \vdash (\texttt{let}\ x := c_1\ \texttt{in}\ c_2)[\eta_1\eta_2\sigma] : B\langle\mathcal{E}_2\rangle[\sigma]$$

Los casos restantes siguen razonamientos análogos.

□

Completitud

Para la prueba de completitud, que permite pasar del sistema declarativo al bidireccional, primero es necesario introducir algunos lemas auxiliares.

Lema 7. $\forall\ \Gamma,\ t,\ \mathcal{T},\ \mathcal{T}',\ \eta$

$$\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}' \wedge \Gamma[\eta], \mathcal{T}' \sqsubseteq \Gamma, \mathcal{T} \longrightarrow \exists\sigma\,(\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \wedge \Gamma, \mathcal{T} \equiv (\Gamma, \mathcal{T})[\sigma])$$

Demostración. En primer lugar se nota que como η es la instanciación resultante de la síntesis $\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}'$, las variables de efecto de $\mathcal{T}$ sobre las que esta puede estar definida deben ocurrir en Γ. Esto se debe a que las variables de efectos de los tipos anotados a un término son siempre frescas. Con esto, junto a que $\Gamma \equiv \Gamma[\eta]$, se tiene que $(\Gamma, \mathcal{T})[\eta] \equiv \Gamma, \mathcal{T}$.

Sea σ tal que $\mathcal{T}' \overset{\sigma}{\bowtie}_T \mathcal{T}[\eta]$. Como $\mathcal{T}' \sqsubseteq \mathcal{T}[\eta]$, σ es a lo sumo un renombre de $\mathcal{T}[\eta]$. Luego se puede obtener la derivación:

$$\frac{\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}' \qquad \mathcal{T}' \overset{\sigma}{\bowtie}_T \mathcal{T}[\eta]}{\Gamma \vdash t \overset{\eta\sigma}{\Leftarrow} \mathcal{T}}$$

con $\Gamma, \mathcal{T} \equiv (\Gamma, \mathcal{T})[\eta] \equiv (\Gamma, \mathcal{T})[\eta\sigma]$. □

Lema 8. $\forall\ \Gamma,\ t,\ \mathcal{T},\ \mathcal{T}',\ \eta,\ \sigma$

$$\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \wedge \Gamma, \mathcal{T} \equiv (\Gamma, \mathcal{T})[\sigma] \wedge \Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}' \longrightarrow \Gamma[\eta], \mathcal{T}' \sqsubseteq \Gamma, \mathcal{T}$$

Demostración. Como $\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}'$, el chequeo de este término debe darse por la regla de cambio de modo, es decir, debe tener la forma:

$$\frac{\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}' \qquad \mathcal{T}' \overset{\omega}{\bowtie}_T \mathcal{T}[\eta]}{\Gamma \vdash t \overset{\eta\omega}{\Leftarrow} \mathcal{T}}$$

Para algún ω tal que $\sigma = \eta\omega$ y, por lo tanto, $\Gamma, \mathcal{T} \equiv (\Gamma, \mathcal{T})[\eta\omega]$.

Como $\mathcal{T}' \overset{\omega}{\bowtie}_T \mathcal{T}[\eta]$, se tiene que $\mathcal{T}'[\omega] = \mathcal{T}[\eta\omega]$. Aplicando esta igualdad sobre el resultado anterior, se alcanza $(\Gamma, \mathcal{T})[\eta\omega] = (\Gamma[\eta], \mathcal{T}')[\omega] \equiv \Gamma, \mathcal{T}$. Esto, por definición, equivale a $\Gamma[\eta], \mathcal{T}' \sqsubseteq \Gamma, \mathcal{T}$. □

Estos dos lemas permiten garantizar que si el término t cuenta con una regla de síntesis, se puede tomar la parte izquierda de la disyunción en la hipótesis inductiva en el Teorema 3 y que para cualquier término se puede tomar la parte derecha.

El siguiente lema establece que los juicios de síntesis y chequeo preservan la relación $\sqsubseteq$.

Lema 9. $\forall\, \Gamma, \Gamma', t, \mathcal{T}, \mathcal{T}', \eta, \sigma$

$$\Gamma', \mathcal{T}' \sqsubseteq \Gamma, \mathcal{T} \wedge \Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \longrightarrow \exists \sigma' \; (\Gamma' \vdash t \overset{\sigma'}{\Leftarrow} \mathcal{T}' \wedge (\Gamma', \mathcal{T}')[\sigma'] \sqsubseteq (\Gamma, \mathcal{T})[\sigma]) \quad (1)$$

$$\Gamma' \sqsubseteq \Gamma \wedge \Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T} \longrightarrow \exists \eta', \mathcal{T}' \; (\Gamma' \vdash t \overset{\eta'}{\Rightarrow} \mathcal{T}' \wedge \Gamma'[\eta'], \mathcal{T}' \sqsubseteq \Gamma[\eta], \mathcal{T}) \quad (2)$$

Demostración. Simple inducción mutua sobre los juicios de tipado. □

Habiendo presentado todo esto, se repite a continuación el enunciado de la Completitud:

Teorema 3 (Completitud)**.** $\forall\, \Gamma, t, \mathcal{T}$

$$\Gamma \vdash t : \mathcal{T} \wedge SA(t) \longrightarrow \quad \exists \eta, \mathcal{T}' \; (\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T}' \wedge \Gamma[\eta], \mathcal{T}' \sqsubseteq \Gamma, \mathcal{T}) \vee \quad (1)$$

$$\exists \sigma \; (\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \wedge \Gamma, \mathcal{T} \equiv (\Gamma, \mathcal{T})[\sigma]) \quad (2)$$

Demostración. Se hace inducción sobre la derivación de $\Gamma \vdash t : \mathcal{T}$. Si la última regla aplicada fue:

Val$^{\mathbf{D}}$. Se tiene que

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathtt{val}\, e : A\langle \mathcal{E} \rangle}$$

Como el término está suficientemente anotado, e tiene regla de síntesis. Luego, por hipótesis inductiva y Lema 8, existen η y A' tales que

$$\Gamma \vdash e \overset{\eta}{\Rightarrow} A' \wedge \Gamma[\eta], A' \sqsubseteq \Gamma, A$$

En consecuencia, por Val, $\Gamma \vdash \mathtt{val}\, e \overset{\eta}{\Rightarrow} A\langle \mu' \rangle$.
Al ser μ' fresca, $\Gamma[\eta], A'\langle \mu' \rangle \sqsubseteq \Gamma, A\langle \mathcal{E} \rangle$.

App$^{\mathbf{D}}$. Se tiene

$$\frac{\Gamma \vdash e_1 : A \to \underline{C} \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : \underline{C}}$$

Como el término está suficientemente anotado, e_1 tiene regla de síntesis. Luego, por hipótesis inductiva y Lema 8, existen η y $A' \to \underline{C'}$ tales que

$$\Gamma \vdash e_1 \overset{\eta}{\Rightarrow} A' \to \underline{C'} \wedge \Gamma[\eta], A' \to \underline{C'} \sqsubseteq \Gamma, A \to \underline{C}$$

Por Lema 7 e hipótesis inductiva, existe σ tal que

$$\Gamma \vdash e_2 \overset{\sigma}{\Leftarrow} A \wedge \Gamma, A \equiv (\Gamma, A)[\sigma]$$

Aplicando el Lema 9, se tiene que existe σ' tal que

$$\Gamma[\eta] \vdash e_2 \overset{\sigma'}{\Leftarrow} A' \wedge (\Gamma[\eta])[\sigma'] \sqsubseteq \Gamma, A$$

Luego,

$$\Gamma \vdash e_1\ e_2 \overset{\eta\sigma'}{\Rightarrow} \underline{C'}[\sigma'] \wedge \Gamma[\eta\sigma'], \underline{C'}[\sigma'] \sqsubseteq \Gamma, \underline{C}$$

Fun$^{\mathbf{D}}$. Se tiene

$$\frac{\Gamma,\ x : A \vdash c : \underline{C}}{\Gamma \vdash \mathtt{fun}\ x \mapsto c : A \to \underline{C}}$$

Por hipótesis inductiva y Lema 7, existe σ tal que

$$\Gamma,\ x : A \vdash c \overset{\sigma}{\Leftarrow} \underline{C} \wedge \Gamma,\ x : A, C \equiv (\Gamma,\ x : A, C)[\sigma]$$

Luego, por Fun, se tiene

$$\Gamma \vdash \mathtt{fun}\ x \mapsto c \overset{\sigma}{\Leftarrow} A \to \underline{C} \wedge \Gamma, A \to \underline{C} \equiv (\Gamma, A \to \underline{C})[\sigma]$$

Anno$^{\mathbf{D}}$. Se tiene

$$\frac{\Gamma \vdash t : \mathcal{T}}{\Gamma \vdash (t : \mathcal{T}) : \mathcal{T}}$$

Por hipótesis inductiva y Lema 7, existe σ tal que

$$\Gamma \vdash t \overset{\sigma}{\Leftarrow} \mathcal{T} \wedge (\Gamma, \mathcal{T})[\sigma] \equiv \Gamma, \mathcal{T}$$

Luego, por Anno, se alcanza

$$\Gamma \vdash t : \mathcal{T} \overset{\sigma}{\Rightarrow} \mathcal{T}[\sigma] \wedge (\Gamma, \mathcal{T})[\sigma] \sqsubseteq \Gamma, \mathcal{T}$$

Los casos restantes son triviales o siguen razonamientos análogos.

□

Finalmente, se enuncian dos lemas que establecen que todo término cuyo tipo es sintetizable está suficientemente anotado y que la relación de evaluación preserva esta propiedad.

Lema 10. $\forall \Gamma, t, \mathcal{T}, \eta\ (\Gamma \vdash t \overset{\eta}{\Rightarrow} \mathcal{T} \longrightarrow SA(t))$

Demostración. Simple inducción en la derivación de tipado. □

Lema 11. $\forall t, t'\ (SA(t) \wedge t \rightsquigarrow t' \longrightarrow SA(t'))$

Demostración. Simple inducción sobre la relación de evaluación. □

Capítulo 7

Conclusiones y trabajo futuro

Los modelos matemáticos de los efectos computacionales son fundamentales para comprender y razonar sobre programas. Sin embargo, para que estos sean adoptados masivamente, es necesario que sus presentaciones prácticas no introduzcan barreras que dificulten la construcción de programas a sus usuarios. Esto mismo es lo que se busca con los sistemas basados en la teoría de efectos algebraicos, y en *Alef* en particular.

Por esta razón, en esta tesina se presentó un lenguaje con efectos modulares que permiten expresar programas impuros de manera reutilizable pero controlada, junto a un sistema de efectos implementable descripto con un nivel de generalidad suficiente para permitir ser extendido sin mayores complicaciones. La extensibilidad del sistema es resultado de que la variación introducida del mecanismo de tipado bidireccional —que aumenta la expresividad de la técnica bidireccional— se expresa de manera independiente del lenguaje y sus tipos.

La principal novedad del sistema bidireccional propuesto en esta tesina se encuentra en las instanciaciones explícitas presentes en los juicios. Estas instanciaciones brindan mayor flexibilidad al tipado bidireccional y pueden ser usadas para llevar información de distinta clase, de acuerdo a lo que se busque conseguir con el sistema de tipado.

Haciendo uso de esta técnica se introdujo un sistema de efectos dirigido por sintaxis cercano conceptualmente a las ideas originales de la teoría categórica de efectos algebraicos. Por consiguiente, se logró describir un sistema de efectos preciso, implementable y con una semántica clara. En contraste con otros sistemas de efectos, esta presentación permite una implementación directa, evitando la necesidad de introducir mecanismos externos al formalismo. Este último hecho puede verse en la implementación Haskell provista, donde las funciones que concretizan el tipado no son más que una simple traducción de las reglas que lo definen.

Durante el diseño del sistema presentado, se formularon distintas variantes del mismo. Las versiones iniciales resultaron demasiado simples, por lo que no lograban capturar la información necesaria para expresar el flujo de los efectos.

Por todo lo anterior, se considera que se cumplieron en gran medida los objetivos

establecidos en la propuesta de esta tesina. A pesar de que la necesidad de sintetizar información de efectos obligó a complejizar el sistema un poco más que lo esperado, este preserva la simplicidad conceptual buscada, en especial cuando se lo compara con el resto de los sistemas existentes. Adicionalmente, fue este requerimiento lo que condujo a la formulación de la nueva variante del tipado bidireccional previamente mencionada, que es otro aporte de esta tesina.

7.1. Trabajo futuro

Como se mostró a lo largo de la tesina, la usabilidad de un sistema de efectos depende en gran manera de la precisión de sus tipos. Es por esto que las principales líneas de trabajo futuro conciernen maneras de aumentar esta última, generalmente resolviendo problemas que provocan imprecisiones en el tipado.

Composición de handlers

Una fuente de imprecisión en los tipos surge de la manera en que se componen los handlers en los cálculos basados en efectos algebraicos. Como se mostró en el Código 5.15, la noción de composición de handler corresponde a:

$$\texttt{with}\ (h_2 \circ h_1)\ \texttt{handle}\ c ::= \texttt{with}\ h_2\ \texttt{handle}\ (\texttt{with}\ h_1\ \texttt{handle}\ c)$$

El problema de esto es que, como los handlers son abiertos, esta forma de componerlos hace que se capturen operaciones de c de más. En concreto, si los tipos de estos últimos fueran:

$$h_1 : \texttt{nat}\langle \texttt{Op}_1 \mid \mu \rangle \twoheadrightarrow \texttt{nat}\langle \texttt{Op}_2 \mid \mu \rangle$$
$$h_2 : \texttt{nat}\langle \texttt{Op}_2 \mid \mu \rangle \twoheadrightarrow \texttt{nat}\langle \mu \rangle$$

Entonces las invocaciones a la operación $\texttt{Op}_2$ presentes en c serían capturadas e interpretadas de igual manera que las llamadas por h_1. En resumen, la composición de estos handlers, en vez de tener el tipo esperado:

$$h_2 \circ h_1 : \texttt{nat}\langle \texttt{Op}_1 \mid \mu \rangle \twoheadrightarrow \texttt{nat}\langle \mu \rangle$$

sería de tipo:

$$h_2 \circ h_1 : \texttt{nat}\langle \texttt{Op}_1, \texttt{Op}_2 \mid \mu \rangle \twoheadrightarrow \texttt{nat}\langle \mu \rangle$$

A este problema se lo conoce con el nombre *polución* [8] y cuenta con distintas soluciones propuestas. Sin embargo, todas estas [8, 16, 4] se basan en introducir nuevas construcciones que permiten encapsular efectos de tal manera que estos puedan "ignorar" el primer handler que los debería manejar. Si bien estas son efectivas en la práctica, ninguna aborda el problema desde sus fundamentos. Es por esto que se considera también como trabajo futuro la tarea de definir una noción de composición de handlers más precisa que permita evitar este problema.

Instancias dinámicas de efectos

Como se detalló en el Capítulo 4, la razón por la cual la implementación actual de Eff no cuenta con un sistema de efectos yace en la dificultad de capturar el comportamiento de las instancias dinámicas de efectos. Por consiguiente, otra línea de trabajo corresponde a intentar modelar estas últimas en *Alef*. De lograrlo, se mostraría que el sistema presentado en esta tesina puede ser extendido para el lenguaje Eff.

Extensiones

Puesto que *Alef* es un cálculo básico para experimentar con efectos algebraicos, un importante área de trabajo futuro corresponde a integrar extensiones al enfoque de efectos algebraicos, como *scoped operations* [22], al desarrollo de esta tesina.

Formalización

Otra tarea de trabajo futuro corresponde a la formalización del cálculo de *Alef* y los resultados teóricos de esta tesina en un asistente de pruebas.

Bibliografía

[1] Andrej Bauer. "What is algebraic about algebraic effects and handlers?" En: *arXiv preprint arXiv:1807.05923* (2018).

[2] Andrej Bauer y Matija Pretnar. "An Effect System for Algebraic Effects and Handlers". En: *Logical Methods in Computer Science* 10.4 (2014). DOI: 10.2168/LMCS-10(4:9)2014. URL: https://doi.org/10.2168/LMCS-10(4:9)2014.

[3] Andrej Bauer y Matija Pretnar. "Programming with algebraic effects and handlers". En: *J. Log. Algebr. Meth. Program.* 84.1 (2015), págs. 108-123. DOI: 10.1016/j.jlamp.2014.02.001. URL: https://doi.org/10.1016/j.jlamp.2014.02.001.

[4] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk y Filip Sieczkowski. "Handle with Care: Relational Interpretation of Algebraic Effects and Handlers". En: *Proc. ACM Program. Lang.* 2.POPL (dic. de 2017). DOI: 10.1145/3158096. URL: https://doi.org/10.1145/3158096.

[5] Jonathan Brachthäuser y Philipp Schuster. "Effekt: extensible algebraic effects in Scala (short paper)". En: oct. de 2017, págs. 67-72. DOI: 10.1145/3136000.3136007.

[6] Edwin Brady. "Programming and Reasoning with Algebraic Effects and Dependent Types". En: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, 133–144. ISBN: 9781450323260. DOI: 10.1145/2500365.2500581. URL: https://doi.org/10.1145/2500365.2500581.

[7] Koen Claessen y John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". En: *SIGPLAN Not.* 46.4 (mayo de 2011), 53–64. ISSN: 0362-1340. DOI: 10.1145/1988042.1988046. URL: https://doi.org/10.1145/1988042.1988046.

[8] Lukas Convent, Sam Lindley, Conor McBride y Craig McLaughlin. "Doo bee doo bee doo". En: *Journal of Functional Programming* 30 (mar. de 2020). DOI: 10.1017/S0956796820000039.

[9] Stephen Dolan, Leo White, K. C. Sivaramakrishnan, Jeremy Yallop y Anil Madhavapeddy. "Effective concurrency through algebraic effects". En: *Ocaml Workshop*. 2015.

[10] Nathan Faubion. *Purescript Run*. https://github.com/natefaubion/purescript-run.

[11] Daniel Hillerström y Sam Lindley. "Liberating Effects with Rows and Handlers". En: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: Association for Computing Machinery, 2016, 15–27. ISBN: 9781450344357. DOI: 10.1145/2976022.2976033. URL: https://doi.org/10.1145/2976022.2976033.

[12] Ohad Kammar, Sam Lindley y Nicolas Oury. "Handlers in Action". En: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, 145–158. ISBN: 9781450323260. DOI: 10.1145/2500365.2500590. URL: https://doi.org/10.1145/2500365.2500590.

[13] Ohad Kammar, Sam Lindley y Nicolas Oury. "Handlers in Action". En: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, págs. 145-158. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500590. URL: http://doi.acm.org/10.1145/2500365.2500590.

[14] Ohad Kammar y Gordon D. Plotkin. "Algebraic Foundations for Effect-Dependent Optimisations". En: *SIGPLAN Not.* 47.1 (ene. de 2012), 349–360. ISSN: 0362-1340. DOI: 10.1145/2103621.2103698. URL: https://doi.org/10.1145/2103621.2103698.

[15] Oleg Kiselyov, Amr Sabry y Cameron Swords. "Extensible Effects: An Alternative to Monad Transformers". En: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, 59–70. ISBN: 9781450323833. DOI: 10.1145/2503778.2503791. URL: https://doi.org/10.1145/2503778.2503791.

[16] Daan Leijen. *Algebraic Effect Handlers with Resources and Deep Finalization*. Inf. téc. MSR-TR-2018-10. 2018, pág. 35. URL: https://www.microsoft.com/en-us/research/publication/algebraic-effect-handlers-resources-deep-finalization/.

[17] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types". En: *Mathematically Structured Functional Programming 2014*. EPTCS, 2014. URL: https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types-2/.

[18] Daan Leijen. “Type Directed Compilation of Row-Typed Algebraic Effects”. En: *Proceedings of Principles of Programming Languages (POPL’17), Paris, France*. 2017. URL: https://www.microsoft.com/en-us/research/publication/type-directed-compilation-row-typed-algebraic-effects/.

[19] Sam Lindley, Conor McBride y Craig McLaughlin. “Do Be Do Be Do”. En: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, págs. 500-514. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009897. URL: http://doi.acm.org/10.1145/3009837.3009897.

[20] Eugenio Moggi. “Notions of Computation and Monads”. En: *Inf. Comput.* 93.1 (jul. de 1991), págs. 55-92. ISSN: 0890-5401. DOI: 10.1016/0890-5401(91)90052-4. URL: http://dx.doi.org/10.1016/0890-5401(91)90052-4.

[21] Benjamin C. Pierce y David N. Turner. “Local Type Inference”. En: *ACM Trans. Program. Lang. Syst.* 22.1 (ene. de 2000), págs. 1-44. ISSN: 0164-0925. DOI: 10.1145/345099.345100. URL: http://doi.acm.org/10.1145/345099.345100.

[22] Maciej Piróg, Tom Schrijvers, Nicolas Wu y Mauro Jaskelioff. “Syntax and Semantics for Operations with Scopes”. En: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’18. Oxford, United Kingdom: Association for Computing Machinery, 2018, 809–818. ISBN: 9781450355834. DOI: 10.1145/3209108.3209166. URL: https://doi.org/10.1145/3209108.3209166.

[23] Gordon Plotkin y John Power. “Algebraic Operations and Generic Effects”. En: *Applied Categorical Structures* 11.1 (2003), págs. 69-94. DOI: 10.1023/A:1023064908962. URL: https://doi.org/10.1023/A:1023064908962.

[24] Gordon Plotkin y Matija Pretnar. “Handlers of Algebraic Effects”. En: *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP ’09. York, UK: Springer-Verlag, 2009, págs. 80-94. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9_7. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_7.

[25] Gordon D. Plotkin y John Power. “Adequacy for Algebraic Effects”. En: *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*. FoSSaCS ’01. Berlin, Heidelberg: Springer-Verlag, 2001, págs. 1-24. ISBN: 3-540-41864-4. URL: http://dl.acm.org/citation.cfm?id=646793.704708.

[26] Gordon D. Plotkin y John Power. “Notions of Computation Determine Monads”. En: *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*. FoSSaCS ’02. London, UK, UK: Springer-Verlag, 2002, págs. 342-356. ISBN: 3-540-43366-X. URL: `http://dl.acm.org/citation.cfm?id=646794.704856`.

[27] Matija Pretnar. “An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper”. En: *Electron. Notes Theor. Comput. Sci.* 319.C (dic. de 2015), págs. 19-35. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2015.12.003`. URL: `http://dx.doi.org/10.1016/j.entcs.2015.12.003`.

[28] Matija Pretnar. “Inferring Algebraic Effects”. En: *Logical Methods in Computer Science* 10.3 (2014). Ed. por NeilEditor Jones. ISSN: 1860-5974. DOI: `10.2168/lmcs-10(3:21)2014`. URL: `http://dx.doi.org/10.2168/LMCS-10(3:21)2014`.

[29] Didier Rémy. “Type Inference for Records in Natural Extension of ML”. En: *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Cambridge, MA, USA: MIT Press, 1994, 67–95. ISBN: 026207155X.

[30] Amr Hany Saleh, Georgios Karachalias, Matija Pretnar y Tom Schrijvers. “Explicit Effect Subtyping”. En: *Programming Languages and Systems*. Ed. por Amal Ahmed. Cham: Springer International Publishing, 2018, págs. 327-354. ISBN: 978-3-319-89884-1.

[31] Tom Schrijvers, Maciej Piróg, Nicolas Wu y Mauro Jaskelioff. “Monad Transformers and Modular Algebraic Effects: What Binds Them Together”. En: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, 2019, 98–113. ISBN: 9781450368131. DOI: `10.1145/3331545.3342595`. URL: `https://doi.org/10.1145/3331545.3342595`.

[32] Philip Wadler. “Monads for functional programming”. En: *Program Design Calculi*. Ed. por Manfred Broy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, págs. 233-264. ISBN: 978-3-662-02880-3.