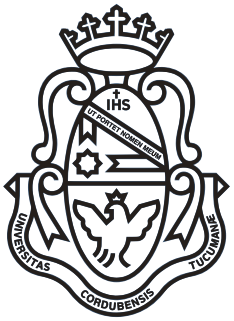
Compilación Certificada sobre Máquinas Abstractas de Evaluación Normal

Leonardo M. Rodríguez

En cumplimiento de los requisitos para adquirir el grado de Doctor en Ciencias de la Computación



FaMAF, 2017

Universidad Nacional de Córdoba

Resumen

El proceso de compilación consiste básicamente en una traducción que parte de un programa *fuente* escrito en algún lenguaje de programación y genera un programa *objeto* que frecuentemente está escrito en un lenguaje de bajo nivel, posiblemente cercano al lenguaje de máquina. Idealmente, el programa fuente tiene definida su semántica (ya sea, de manera operacional o denotacional), y se espera que el proceso de traducción que realiza el compilador preserve esa semántica, es decir, se espera que el programa objeto se comporte como lo indica la semántica del programa fuente.

Sin embargo, como sucede con cualquier tipo de software, este proceso de traducción es propenso a errores. En principio, no se tiene ninguna garantía de que el proceso de traducción no introduzca algún error que pueda resultar en cierto comportamiento inesperado por parte del programa objeto. El hecho de no contar con esa garantía debilita o resta impacto a cualquier intento de establecer propiedades sobre los programas basado en el análisis del código fuente.

El testing del programa objeto puede ayudar a encontrar errores de compilación, pero no puede garantizar la ausencia de los mismos. Las técnicas de testing resultan insuficientes sobre todo cuando se trabaja con sistemas críticos, que demandan mayores garantías de seguridad y confiabilidad.

Un enfoque más ambicioso consiste en demostrar la *corrección del compilador*, es decir, obtener una demostración de que el proceso de traducción preserva la semántica del programa fuente. Desde que aparecieron las primeras pruebas de corrección de compiladores [81, 84], han habido ya numerosas contribuciones en el área. De destacada importancia es el trabajo realizado por el proyecto CompCert [76], un compilador certificado para el lenguaje C. En el caso de los lenguajes funcionales, se pueden mencionar el trabajo de Chlipala [32], que presenta un compilador certificado para el cálculo lambda tipado, y a Benton [15] donde el lenguaje fuente es un lenguaje funcional con evaluación estricta, y el lenguaje objeto es una variante de la máquina abstracta SECD [73].

Para probar que un compilador es correcto es necesario establecer una relación entre la semántica del lenguaje fuente y la semántica del lenguaje objeto. En general, la semántica del lenguaje objeto se describe de manera operacional: se definen cuáles son las instrucciones disponibles en la máquina (que puede ser un microprocesador real o una máquina abstracta) y cómo esas instrucciones modifican el estado a medida que se ejecutan. Por otro lado, existen muchas maneras de describir la semántica del lenguaje fuente, y el método utilizado determina significativamente la estrategia de la prueba de corrección. Existen pruebas de corrección

basadas en semántica natural (big-step) [34, 78], semántica de reducción (small-step) [10] o semántica denotacional [15, 32], entre otras.

Otro aspecto importante del lenguaje fuente es su estrategia de evaluación. Existen lenguajes estrictos (como C) y lenguajes con evaluación normal (como Haskell). La enorme mayoría de los trabajos existentes en el área de corrección de compiladores han estudiado sólo lenguajes estrictos, dejando fuera del análisis a los lenguajes con otro tipo de estrategia de evaluación.

En la tesis se analiza cómo aplicar las diferentes técnicas de demostración sobre lenguajes con evaluación normal. Se consideran diferentes lenguajes de programación funcionales, y para cada uno se demuestra la corrección del compilador empleando un método diferente. Como lenguaje objeto se utilizan distintas variantes de la máquina de Krivine [71], extendidas apropiadamente con las instrucciones necesarias para implementar las funcionalidades que provee cada lenguaje fuente. En particular, la tesis presenta una prueba de corrección de un compilador basada en la semántica denotacional del lenguaje, utilizando técnicas como *step-indexing* [7, 11] y *biortogonalidad* [94] para definir relaciones lógicas que capturen la noción de corrección del compilador de manera composicional. Además, se desarrolla un enfoque basado en la noción de *realizabilidad* [66, 72] para demostrar la corrección del compilador en un lenguaje con evaluación lazy. Todas las pruebas de corrección presentadas en esta tesis están formalizadas el asistente de demostración Coq [3]. En la tesis se presenta el análisis y los comentarios pertinentes sobre cada formalización.

Agradecimientos

A mi familia, José, Mariana y Maxi, por su apoyo incondicional.
A Daniel Fridlender por su paciencia, por apoyarme en mis momentos de inseguridad, por compartir conmigo su enorme conocimiento y sabiduría durante toda mi carrera universitaria.
A Miguel Pagano, *World's Best Advisor*. Siempre presente para darme una mano cuando más me hizo falta. Siempre respondiendo con *categoría*.
A todos mis compañeros de doctorado, Alejandro Gadea, Emmanuel Gunther, y al grupo de Semántica de la Programación. Disfruté mucho el tiempo con ellos.
A Pedro D'Argenio por su apoyo para terminar la tesis mientras trabajé en su grupo de investigación, y por su labor como director de la carrera de doctorado.
A Raúl Monti, Matías "Chun" Lee y Carlos Budde, por el tiempo compartido trabajando para el DSG.
A Franco Luque, Beta Ziliani y Álvaro Tasistro por las correcciones y sugerencias sobre el contenido de la tesis.
A toda la comunidad de computación de FaMAF, excelente grupo humano.

Muchas gracias!

Índice general

Introducción

Un compilador es una herramienta que realiza esencialmente una traducción de un programa fuente a un programa objeto. El programa objeto usualmente está escrito en un lenguaje de bajo nivel, cercano al lenguaje máquina. En cambio, el programa fuente está escrito en un lenguaje de alto nivel, que es más comprensible para el programador y le permite estructurar su tarea de desarrollo modularmente y con abstracciones cercanas al dominio del problema.

Es usual que el programador asuma que el proceso de traducción que realiza el compilador se lleva a cabo sin errores, esto es, supone que el programa objeto se comportará exactamente como lo indica el programa fuente. Sin embargo, han existido casos de errores de traducción, incluso en compiladores de nivel industrial, que han dado lugar a código objeto incorrecto [52, 118].

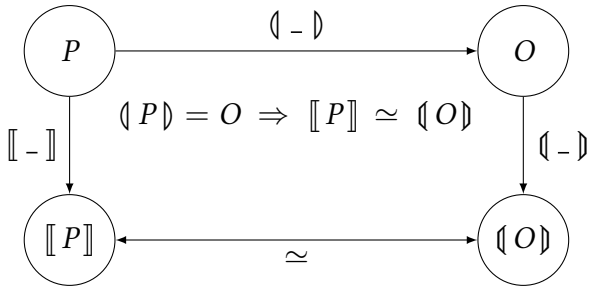
Los errores de traducción son importantes puesto que pueden invalidar todo tipo de análisis estático que se realice sobre el programa fuente. Es decir, si se utilizan métodos formales para demostrar que el programa fuente cumple con cierta propiedad (por ejemplo, que su comportamiento coincide con el de su especificación) entonces un error de traducción puede invalidar esa propiedad haciendo que el código generado se comporte de manera inesperada.

Para evitar esa discrepancia entre el comportamiento del programa fuente y el programa objeto también es posible realizar un análisis dinámico (como por ejemplo testing) sobre el programa objeto. Esto puede ser útil para detectar errores, tanto del programador como del compilador. Sin embargo, este tipo de análisis no es suficiente cuando se consideran sistemas críticos con rigurosos requerimientos de seguridad y confiabilidad.

Una manera de incrementar la confiabilidad en el compilador es demostrando su corrección. Esto es, dar una prueba formal (o certificación) de que el proceso de traducción preserva la semántica del programa fuente. A partir del surgimiento de las primeras pruebas de corrección [81, 84], el área de certificación

de compiladores ha estado en constante crecimiento [79, 86]. El proyecto CompCert [76] es probablemente uno de los trabajos más importantes en la actualidad, donde se ha logrado certificar un compilador de un subconjunto del lenguaje C a un lenguaje ensamblador; y se ha utilizado, por ejemplo, para compilar software embebido desarrollado por la empresa aeronáutica Airbus [14].

Es posible describir esquemáticamente qué significa obtener una prueba de corrección del compilador. Suponiendo que $[\![P]\!]$ es la semántica del programa fuente P, y $(\!| O |\!)$ es la semántica del programa objeto O, demostrar la corrección del compilador $(\!| \_ |\!)$ consiste en probar que $(\!| P |\!) = O \Rightarrow [\![P]\!] \simeq (\!| O |\!)$, para todo P y O.



La definición de la relación $\simeq$ puede variar de acuerdo a la propiedad concreta que se quiere demostrar, y de cómo están definidas las funciones semánticas $[\![\_]\!]$ y $(\!| \_ |\!)$. La función $(\!| \_ |\!)$ usualmente describe el comportamiento operacional del entorno de ejecución (definiendo cómo las instrucciones de un microprocesador o de una máquina abstracta modifican el estado computacional). En cambio, la función $[\![\_]\!]$ puede describirse operacionalmente (mediante reglas de contracción o de evaluación) o también denotacionalmente (asociando el significado de un programa con un objeto matemático que posee cierto contenido computacional). Cuando $[\![\_]\!]$ es una descripción operacional, la relación $\simeq$ establece una correspondencia entre el valor final que surge de evaluar P con el estado final obtenido al ejecutar O. Cuando $[\![\_]\!]$ se define de manera denotacional, la relación $\simeq$ es usualmente una relación lógica que describe la correspondencia operacional (o también observacional) entre el programa O y el objeto matemático $[\![P]\!]$. Es claro entonces que la estrategia de prueba de corrección de un compilador depende fuertemente de la semántica del lenguaje, por lo que en la tesis se explora tanto semántica operacional como denotacional.

Como un ejemplo de un compilador certificado basado en la semántica denotacional, podemos destacar el trabajo de Chlipala [32], un compilador del cálculo lambda tipado a un lenguaje ensamblador "idealizado" (con una cantidad infinita de registros y celdas de memoria). El compilador consta de varias etapas de traducción, donde se utilizan diferentes lenguajes intermedios, con

distintos niveles de abstracción, hasta llegar al lenguaje ensamblador. La prueba de corrección se construye usando relaciones lógicas indexadas por tipos que capturan la correspondencia entre las etapas de la traducción.

Otro ejemplo a destacar es el compilador certificado de Benton et al. [15], que traduce términos de un lenguaje funcional a las instrucciones de la máquina abstracta SECD [73]. Mediante la aplicación de técnicas como *step-indexing* [7, 11] y *biortogonalidad* [94], se definen relaciones lógicas entre las configuraciones de la máquina y los objetos denotacionales que describen la semántica del lenguaje.

Leroy [76] presenta un compilador certificado de C a lenguaje ensamblador, cuya prueba de corrección se construye siguiendo un esquema de simulación operacional, esto es, cada transición del programa fuente se corresponde con una secuencia de transiciones del programa objeto que tiene el mismo comportamiento observable y preserva ciertas invariantes con respecto al entorno de ejecución.

En la literatura se pueden encontrar muchas otras pruebas de corrección de compiladores, donde el entorno de ejecución tiene diferentes niveles de abstracción, que pueden variar desde lenguajes ensambladores concretos correspondientes microprocesadores reales, lenguajes ensambladores idealizados y, con mayor frecuencia, lenguajes de instrucciones correspondientes a una máquina abstracta. Las máquinas abstractas proveen un modelo de ejecución idealizado, en general más simple que una máquina real, ya que carecen de especificaciones de bajo nivel que complicarían de otra forma el razonamiento y el análisis del comportamiento de la ejecución.

Existen diferentes tipos de máquinas abstractas, utilizadas en general como lenguajes intermedios del proceso de compilación, y son diseñadas para lenguajes con diferentes características como son los lenguajes funcionales, imperativos u orientados a objeto. Referimos al lector a [49] para un relevamiento de algunas las máquinas abstractas existentes en la literatura.

En la tesis utilizamos como entorno de ejecución distintas variantes de la máquina abstracta de Krivine [71], diseñada originalmente para evaluar el cálculo lambda puro, pero que en este trabajo ha sido extendida para poder implementar lenguajes con un mayor nivel de expresividad. También utilizaremos una variante de la máquina de Sestoft [111] cuando analizamos la estrategia de evaluación lazy (call-by-need).

La tesis explora funciones de compilación que traducen directamente los términos del lenguaje a instrucciones de una máquina abstracta. Dejamos afuera del análisis a compiladores con múltiples etapas, y a la generación de código de bajo nivel correspondiente a microprocesadores. La mayoría de los trabajos actuales en certificación de compiladores parten desde la representación abstracta del lenguaje fuente y culminan en la generación del código, sin adentrar

en la verificación del parser [64, 67] o la verificación del hardware [28, 29, 43], áreas que tampoco se cubren en esta tesis.

Nos enfocamos en lenguajes con orden de evaluación normal (call-by-name), donde la evaluación del argumento de una aplicación se pospone hasta que su valor sea necesario para obtener el valor final del término. La investigación respecto a la certificación de compiladores para lenguajes con esta estrategia de evaluación se encuentra prácticamente ausente en la literatura y consideramos que esta tesis es un aporte en esa dirección.

En la tesis presentamos varios lenguajes de programación basados en el cálculo lambda y utilizamos distintos métodos para definir su semántica: reglas de contracción (Capítulo 2), reglas de evaluación (Capítulo 3) y semántica denotacional (Capítulo 6 y 7). La mayoría de los lenguajes que consideramos son funcionales, pero también analizamos un lenguaje imperativo y definimos su semántica usando un sistema de reglas de evaluación (Capítulo 4). Este sistema de reglas tiene la particularidad de expresar de manera explícita la disciplina de pila inherente a los lenguajes Algol-like [99].

El aporte más significativo de la tesis está en las demostraciones de corrección de compiladores con respecto a la semántica denotacional. Se define un nuevo enfoque de *realizabilidad* (Capítulo 5) basado en la noción usual de realizabilidad de Krivine [72] que luego aplicamos para definir relaciones lógicas de aproximación y demostrar la corrección del compilador (Capítulo 6), empleando a su vez técnicas como *step-indexing* y *biortogonalidad* para definir esas relaciones de manera composicional y modular. Si bien técnicas similares se han utilizado previamente para demostrar la corrección de compiladores [15, 16], fueron aplicadas a lenguajes estrictos (call-by-value), dejando sin explorar los lenguajes con evaluación normal. Consideramos que, dado el cambio en estrategia de evaluación y el hecho de haber elegido la máquina de Krivine como entorno de ejecución, los métodos de demostración que se han estudiado con anterioridad no son aplicables y por lo tanto esta tesis resulta una contribución en ese sentido.

En la estrategia de evaluación lazy (call-by-need), los argumentos de una aplicación se evalúan sólo cuando cuando es necesario y a lo sumo una vez. La máquina abstracta de Sestoft [111] es una adaptación de la máquina de Krivine que utiliza un *heap* en la configuración para evitar la repetición innecesaria de la ejecución de código. En la tesis se presenta un esquema de realizabilidad aplicado a una variante de la máquina de Sestoft (Capítulo 7). Hasta donde sabemos, esta tesis explora por primera vez la aplicación de realizabilidad sobre una máquina abstracta de evaluación lazy.

Las publicaciones que respaldan los resultados de esta tesis son [105, 107] para las demostraciones basadas en semántica big-step y [106] para las basadas en semántica denotacional. En [104] presentamos un sistema de tipos para un

lenguaje lazy y la prueba de preservación de tipos (*subject reduction*) sobre la máquina abstracta de Sestoft. El contenido del Capítulo 7, sin embargo, no está publicado aunque forma parte de un artículo en preparación.

Los teoremas de corrección fueron formalizados en Coq [3], un asistente de demostración con tipos dependientes basado en el Cálculo de Construcciones Inductivas [41, 42]. La formalización completa está disponible online[1], y en el Capítulo 8 se comentan algunos fragmentos importantes del código.

Para la formalización del Capítulo 6, se utilizó como base una librería de teoría de dominios desarrollada por Benton [19] que fue extendida de acuerdo a los requerimientos específicos de nuestra formalización.

[1] https://cs.famaf.unc.edu.ar/~leorodriguez/tesis.html

Capítulo 1

El Cálculo Lambda y su Semántica

En la tesis demostramos la corrección de compiladores para diferentes lenguajes de programación. El núcleo común de cada uno de esos lenguajes es el cálculo lambda [36, 37], tanto en la definición de la sintaxis como la semántica. En este capítulo haremos una presentación breve del cálculo lambda, con la intención de introducir algunos conceptos comunes que se desarrollarán a lo largo de los siguientes capítulos, y al mismo tiempo fijar algunas notaciones importantes.

A continuación introduciremos la sintaxis del cálculo lambda puro y presentaremos algunas de las formas más usuales de describir su semántica.

1.1 Sintaxis

Comenzaremos con la definición de los términos.

Definición 1 (Términos). .

$$x, y, z \in \mathit{Var} \qquad \textit{(Variables)}$$
$$t,\ t' \in \mathit{Term} ::= x \mid \lambda x.\, t \mid t\ t' \qquad \textit{(Términos)}$$

Las meta-variables x, y y z denotan variables del lenguaje, que a su vez son elementos del conjunto predefinido *Var*. De la misma forma, las meta-variables t y t' denotan elementos de *Term*, el conjunto de términos del lenguaje.

En el cálculo lambda, los términos pueden ser variables, abstracciones $\lambda x.\, t$ (donde al término t se lo llama *cuerpo* de la abstracción), o bien aplicaciones $t\ t'$ (donde al término t se lo llama *operador* y al término t' se lo llama *operando*).

A pesar de su simpleza, el cálculo lambda tiene un enorme poder expresivo que permite codificar distintos tipos de datos y funciones recursivas. Los números naturales, por ejemplo, pueden representarse mediante los numerales de Church [13, 59]. La recursión se codifica mediante la aplicación de combinadores de punto fijo (por ejemplo, los combinadores de Curry [57] y de Turing [115]).

Como es usual, las variables de un término pueden tener ocurrencias libres o ligadas. En el cálculo lambda, la única forma de introducir una variable ligada es mediante una abstracción, pero en otros lenguajes existen otros tipos de términos ligadores (o en inglés, binders), algunos de ellos se verán en los siguientes capítulos.

La noción de variable libre y ligada se puede definir formalmente mediante las siguientes funciones.

Definición 2 (Variables libres y ligadas).

$$\begin{aligned} FV(x) &= \{ x \} & BV(x) &= \emptyset \\ FV(\lambda x.\ t) &= FV(t) - \{ x \} & BV(\lambda x.\ t) &= BV(t) \cup \{ x \} \\ FV(t\ t') &= FV(t) \cup FV(t') & BV(t\ t') &= BV(t) \cup BV(t') \end{aligned}$$

Aquí $FV(t)$ es el conjunto de variables libres del término t, y $BV(t)$ es el conjunto de variables ligadas. Cuando un término t no tiene variables libres (es decir, $FV(t) = \emptyset$) se dice que t es un término *cerrado*. En otro caso, cuando tiene variables libres, se dice que t es *abierto*. De la definición se deduce que las variables libres de una abstracción $\lambda x.\ t$ son las variables libres de t excepto por la variable x, cuyas ocurrencias en t son ligadas. Una variable libre de una aplicación puede ocurrir tanto en el operador como en el operando.

La semántica del cálculo lambda se puede definir de diferentes maneras, y a continuación introduciremos algunas de ellas.

1.2 Semántica small-step

Comenzaremos con una descripción operacional, llamada semántica *small-step* [95], en la cual se definen distintas reglas de contracción que permiten obtener paso a paso, y en caso de ser posible, una *forma canónica* a partir de un término del cálculo. Las formas canónicas son aquellos términos que se consideran el resultado final o *valor* de una reducción, en el caso del cálculo lambda puro sólo las abstracciones se consideran formas canónicas.

A continuación se muestra un conjunto de reglas de contracción que permite la reducción de términos hasta una forma canónica *débil* (en inglés, *weak*

head normal form), que quiere decir que la reducción se detiene si se consigue llegar a una abstracción posiblemente sin haber reducido el cuerpo de la misma.

Definición 3 (Reglas de contracción).

$$(\beta)\ \frac{}{(\lambda x.\ t)\ t' \to t[x \backslash t']} \qquad (\nu)\ \frac{t \to t''}{t\ t' \to t''\ t'}$$

Estas reglas de contracción son suficientes si asumimos que el término que se pretende reducir es cerrado, es decir, que no tiene variables libres. La regla β realiza la contracción de un *redex*, es decir, un término de la forma $(\lambda x.\ t)\ t'$. El paso de reducción consiste en sustituir en t todas las ocurrencias libres de x por el término t', operación que denotamos $t[x \backslash t']$.

La operación de sustitución debe evitar la captura de variables libres. Por ejemplo, si se realiza la sustitución en el término $(\lambda x.\ \lambda y.\ z)[z \backslash y]$ la variable libre y quedaría capturada en la abstracción más interna. Es claro que si t' es cerrado, entonces la sustitución $t[x \backslash t']$ no captura nombres de variables.

La regla ν nos permite contraer el operador de una aplicación. Usualmente, esta regla se aplica repetidas veces hasta lograr reducir el operador a una abstracción, obteniendo de esa forma un redex, quedando entonces habilitados para aplicar la regla β y continuar la reducción.

Es importante notar que en la regla β la variable x se reemplaza por el término t' sin éste haber sido reducido. Esto se debe a que estamos considerando un conjunto restringido de las reglas usuales de contracción del cálculo lambda que definen una estrategia de *evaluación normal*: los argumentos se reducen sólo cuando sea necesario para obtener la forma canónica. Por esa misma razón la regla ν sólo permite la reducción contextual del operador, pero no la del operando.

La interpretación inductiva de la Definición 3 define a $\to \subseteq \mathit{Term} \times \mathit{Term}$ como la menor relación cerrada por las reglas de contracción. Dado que cada regla de contracción representa un paso en la reducción de un término, es natural tomar la clausura reflexiva-transitiva de la relación $\to$, que representará la aplicación sucesiva de las reglas de contracción.

Definimos la clausura reflexiva-transitiva de manera general, para cualquier relación $R \subseteq A \times A$. Las reglas *Refl y *Step extienden a la relación R agregando los pares necesarios para asegurar respectivamente la reflexividad y la transitividad de la relación.

Definición 4 (Clausura reflexiva-transitiva). *Si $R \subseteq A \times A$ es una relación, entonces la relación $R^* \subseteq A \times A$ queda definida inductivamente con las siguientes reglas:*

$$(^*\text{Refl})\ \frac{}{a\ R^*\ a} \qquad (^*\text{Step})\ \frac{a\ R\ b \qquad b\ R^*\ c}{a\ R^*\ c}$$

De esta manera, la clausura reflexiva-transitiva de $\to$ se escribe $\to^*$. En el cálculo lambda, la reducción de un término cerrado t consiste en encontrar una abstracción $\lambda x.\ t'$ tal que $t \to^* \lambda x.\ t'$. No siempre es posible encontrar dicha abstracción, por ejemplo, el término $t\ t$ donde $t = \lambda x.\ x\ x$ no puede reducirse a una forma canónica, veremos en el Capítulo 2 cómo se puede expresar ese hecho de manera formal.

1.3 Semántica big-step

En la semántica small-step, las reglas de contracción representan un único paso de computación que permiten reducir el término inicial a una forma canónica. En cambio, en la semántica *big-step* [65], se definen reglas de *evaluación*, donde cada regla representa la reducción completa del término hasta obtener el valor final. En el cálculo lambda, los valores que se obtienen durante la evaluación son únicamente abstracciones.

Definición 5 (Valores).

$$v \in \mathit{Val} ::= \lambda x.\ t$$

Las reglas de evaluación se describen con una relación $\Rightarrow \subseteq\ \mathit{Term} \times \mathit{Val}$, cuyo dominio son todos los términos cerrados. Las siguientes son las reglas de evaluación normal para el cálculo lambda.

Definición 6 (Reglas de evaluación).

$$(\text{Abs})\ \frac{}{\lambda x.\ t\ \Rightarrow\ \lambda x.\ t} \qquad (\text{App})\ \frac{t\ \Rightarrow\ \lambda x.\ t'' \qquad t''[x \backslash t']\ \Rightarrow\ v}{t\ t'\ \Rightarrow\ v}$$

La regla Abs expresa que la evaluación de una abstracción tiene como valor a la misma abstracción. Por otro lado, la regla App expresa que para evaluar una aplicación $t\ t'$, primero es necesario evaluar el operador t obteniendo un valor de la forma $\lambda x.\ t''$, luego sustituir la variable x en t'' con el operando y

finalmente evaluar el término resultante $t''[x \backslash t']$, obteniendo así el valor final. Es importante observar que en la regla App la sustitución de la segunda premisa se realiza con el término t' sin evaluar, debido a que nos enfocamos únicamente en la evaluación normal.

1.4 Semántica big-step con entornos

Más adelante veremos que para compilar términos del cálculo lambda a las instrucciones de una máquina abstracta resulta conveniente, por un lado, dar semántica a términos que no sean necesariamente cerrados, y por otro, evitar que en las reglas de evaluación sea necesario utilizar sustituciones de variables por términos. Esto se puede lograr introduciendo *entornos* en las reglas de evaluación. Intuitivamente, los entornos contienen la información necesaria para evaluar cada una de las variables libres de un término.

Definición 7 (Entornos y clausuras).

$$\begin{array}{lll} e \in \mathit{Env} & \subseteq \mathit{Var} \to \mathit{Clos} & \textit{(Entornos)} \\ (t,\ e) \in \mathit{Clos} & \subseteq \mathit{Term} \times \mathit{Env} & \textit{(Clausuras)} \end{array}$$

Los entornos son funciones de variables a clausuras que tienen dominio finito. Una clausura es a su vez un par de la forma $(t,\ e)$ donde las variables libres de t pertenecen al dominio del entorno e. Las reglas de evaluación pueden reformularse para usar entornos en lugar de sustituciones. Un valor de la evaluación es una clausura de la forma $(\lambda x.\ t,\ e)$, es decir, una abstracción junto con un su entorno.

Definición 8 (Valores).

$$v \in \mathit{Val} ::= (\lambda x.\ t,\ e)$$

Las reglas inductivas de evaluación se describen con juicios de la forma $e \vdash t \Rightarrow v$, indicando que el término t evalúa a v bajo el entorno e. A continuación describimos el conjunto de reglas de evaluación.

Definición 9 (Reglas de evaluación con entornos).

$$\text{(Abs)}\ \frac{}{e \vdash \lambda x.\, t \Rightarrow (\lambda x.\, t,\ e)} \qquad \text{(Var)}\ \frac{e' \vdash t' \Rightarrow v}{e \vdash x \Rightarrow v}\ e(x) = (t',\ e')$$

$$\text{(App)}\ \frac{e \vdash t \Rightarrow (\lambda x.\, t'',\ e') \qquad e'\,[x \mapsto (t',\ e)] \vdash t'' \Rightarrow v}{e \vdash t\ t' \Rightarrow v}$$

La regla Abs indica que bajo el entorno e, la abstracción evalúa a una clausura formada por la misma abstracción y el entorno e intacto. La regla Var permite evaluar términos con variables libres: si queremos conocer el valor de una variable x bajo el entorno e es necesario evaluar la clausura asociada en ese entorno, es decir $e(x)$. La regla App indica que para evaluar una aplicación $t\ t'$ bajo el entorno e, primero debemos evaluar el operador t bajo e y obtener un valor de la forma $(\lambda x.\, t'',\ e')$; para luego evaluar el término t'' bajo el entorno extendido $e'\,[x \mapsto (t',\ e)]$ y obtener así el valor resultante.

Notación: Cuando f es una función de A en B se define la función extendida $f[x \mapsto b]$ de dominio $A \cup \{ x \}$ e imagen $B \cup \{ b \}$ de la siguiente manera:

$$f[x \mapsto b]\, z = \begin{cases} b & z = x \\ f(z) & z \neq x \end{cases}.$$

Cuando una función f de A en B tiene dominio finito se la denomina *mapeo*; en ese sentido un entorno es un mapeo de *Var* a *Clos*. Si f es un mapeo, escribimos $f - x$ al mapeo con dominio $dom(f) - \{ x \}$ que coincide con f en todos los puntos de dicho dominio.

Es posible establecer una relación entre la semántica big-step con entornos y la semántica big-step sin entornos. Intuitivamente, toda clausura $(t,\ e)$ tiene asociado un término que se obtiene de reemplazar todas las variables libres de t por el contenido del entorno. Para formalizar esta idea se necesita que dicha clausura sea *bien formada* en el sentido de que todas las variables libres de t estén en el dominio de e, y que a su vez que todas las clausuras del entorno e estén bien formadas. A la operación que transforma una clausura en un término la denominamos *aplanamiento*.

Definición 10 (Aplanamiento de la clausura).

$$flat\,(t,\ e) = \begin{cases} t & si\ dom(e) = \emptyset \\ flat\,(t[x \backslash flat\,(t',\ e')],\ e - x) & si\ e(x) = (t',\ e') \end{cases} .$$

Cuando el entorno e está bien formado, y la clausura $(t,\ e)$ también lo está, se puede demostrar el siguiente lema que conecta las dos formas de evaluación.

Lema 1. *Si* $e \vdash t \Rightarrow (\lambda x.\ t,\ e')$ *entonces* $flat\,(t,\ e) \Rightarrow flat\,(\lambda x.\ t,\ e')$.

En la Sección 8.2 comentamos la formalización en Coq de este lema. En esa implementación, la función $flat\,(t,\ e)$ se define recursivamente en la estructura del entorno e que se representa mediante un tipo de datos inductivo.

1.5 Semántica denotacional

En las descripciones operacionales del cálculo lambda, la semántica de un término se obtiene mediante su evaluación o reducción hasta obtener un valor o forma canónica. La semántica *denotacional* [108], por otro lado, describe la semántica de un término asociándole un objeto matemático que describa su significado. Existen muchos modelos denotacionales del cálculo lambda, algunos de ellos más generales que otros [68, 82]. Aquí nos enfocaremos en el modelo de los *dominios* y las funciones continuas. Aunque en el Capítulo 6 presentaremos las definiciones básicas, referimos al lector a [5, 54, 109] para una introducción completa a la teoría de dominios.

Intuitivamente, la semántica de un término del cálculo lambda se corresponde con una función matemática. Por ejemplo, el término $\lambda x.\ x$ se corresponde con la función *identidad*, y el término $\lambda z.\ \lambda x.\ z\ x$ se corresponde con la función *aplicación* que toma como parámetros una función y su argumento. Existen, sin embargo, términos como $\lambda x.\ x\ x$ cuyo significado intuitivo es menos claro. En ese caso, el parámetro x es una función que es aplicada a sí misma. Cualquier modelo del cálculo lambda puro debe reflejar la capacidad de auto-aplicación. Por ello, la semántica de un término debe ser una función $f\colon S \to S$ tal que $f \in S$, es decir que el conjunto S debe cumplir $S \to S \subseteq S$. Como muestra Reynolds [100, página 208], un conjunto S con esas características da lugar a inconsistencias relacionadas con la existencia de puntos fijos: se puede demostrar que todas las funciones $f\colon S \to S$ tienen un punto fijo (es decir, un elemento p tal que $fp = p$) y a su vez se pueden encontrar contraejemplos de funciones en $S \to S$ que carecen del mismo. A grandes rasgos, los *dominios* son conjuntos con cierta estructura adicional que garantiza la existencia de puntos fijos

evitando la aparición de dicha inconsistencia, por lo que las funciones denotadas por el cálculo lambda se limitan a funciones *continuas* $f\colon S \to S$ tal que S es un dominio. Veremos la definición formal de dominio y continuidad en el Capítulo 6.

Para modelar el cálculo lambda usualmente se considera un dominio D, isomorfo con el espacio de funciones continuas $[D \to D]$, es decir, $D \simeq [D \to D]$. Se asume entonces la existencia de un isomorfismo $\phi\colon D \to [D \to D]$, y $\psi\colon [D \to D] \to D$. Este isomorfismo permite modelar la auto-aplicación, pues si $f \in [D \to D]$ entonces $f\,(\psi f) \in D$. Para definir la semántica de los términos será necesario contar también con un dominio D^* cuyos elementos se denominan *ambientes*. Los ambientes se utilizan para determinar la semántica de las variables libres de un término, pues asignan a cada variable un elemento de D. Usaremos la meta-variable ρ para denotar ambientes.

Definición 11 (Ambientes).

$$\rho \in D^* = \mathit{Var} \to D$$

La semántica denotacional queda determinada con la definición de una función $[\![\ \_\ ]\!] : \mathit{Term} \to [D^* \to D]$, cuyas ecuaciones se muestran a continuación.

Definición 12 (Semántica denotacional).

$$\begin{aligned} [\![\, \lambda x.\, t \,]\!]\, \rho &= \psi(\hat{\lambda}\, d\,.\, [\![\, t \,]\!]\, (\rho\, [x \mapsto d])) \\ [\![\, x \,]\!]\, \rho &= \rho(x) \\ [\![\, t_1\ t_2 \,]\!]\, \rho &= \phi\, ([\![\, t_1 \,]\!]\, \rho)\, ([\![\, t_2 \,]\!]\, \rho) \end{aligned}$$

Notación: Escribimos $(\hat{\lambda}\, d\,.\ \_)$ para denotar una función en el meta-lenguaje cuyo único parámetro tiene nombre d.

Esas ecuaciones describen formalmente la correspondencia intuitiva entre los términos del cálculo lambda y las funciones en $[D \to D]$. La semántica de la abstracción $\lambda x.\, t$ bajo un ambiente ρ es el objeto en D que resulta de aplicar ψ a la función $(\hat{\lambda}\, d\,.\, [\![\, t \,]\!]\, (\rho\, [x \mapsto d]))$. El resultado de esa función es la semántica del término t en un ambiente que extiende a ρ con el mapeo $x \mapsto d$.

1.6 Sistema de tipos

En general, los sistemas de tipos definen reglas que restringen el conjunto total de términos del lenguaje seleccionando aquellos que cumplen con cierta

propiedad o que poseen cierta estructura. En los lenguajes de programación, los sistemas de tipos se utilizan para evitar errores en tiempo de ejecución y para garantizar un manejo consistente de los datos.

En el cálculo lambda puro con el *sistema de tipos simple* [38] todos los términos bien tipados pueden reducirse a una forma canónica. Existen términos como la auto-aplicación $\lambda x.\ x\ x$ que no son tipables en ese sistema. Como veremos, la semántica denotacional del cálculo lambda puro con el sistema de tipos simple puede definirse usando teoría de conjuntos, evitando la necesidad de los dominios.

Introducimos a continuación el sistema de tipos simple que posee un constructor de tipo básico b y otro constructor para tipos funcionales.

Definición 13 (Tipos).

$$\theta, \theta' \in \mathit{Type} ::= b \mid \theta \to \theta'$$

Definiremos reglas de tipado para términos del lenguaje que pueden ser abiertos (es decir, pueden tener variables libres). Para poder asignar un tipo a un término abierto es necesario tener un *contexto* que le asigne tipos a todas las variables libres de dicho término. Los contextos se definen entonces como mapeos de variables a tipos.

Definición 14 (Contextos).

$$\pi \in \mathit{Ctx} \subseteq \mathit{Var} \to \mathit{Type}$$

Un juicio de tipado de la forma $\pi \vdash t : \theta$ establece que bajo el contexto π el término t tiene tipo θ. Estos juicios aparecen tanto en las premisas como en la conclusión de las reglas de tipado, definidas a continuación.

Definición 15 (Reglas de tipado).

$$\text{(Abs)}\ \frac{\pi[x \mapsto \theta] \vdash t : \theta'}{\pi \vdash \lambda x.\ t : \theta \to \theta'} \qquad \text{(Var)}\ \frac{}{\pi \vdash x : \theta}\ \pi(x) = \theta$$

$$\text{(App)}\ \frac{\pi \vdash t : \theta \to \theta' \qquad \pi \vdash t' : \theta}{\pi \vdash t\ t' : \theta'}$$

La regla Abs establece que una abstracción $\lambda x.\ t$ tiene el tipo funcional $\theta \to \theta'$ bajo el contexto π si el cuerpo de la abstracción t tiene tipo θ' bajo un contexto que extiende a π con el mapeo $x \mapsto \theta$. Como indica la regla Var, el tipo de una variable se obtiene consultando en el contexto. Si una variable libre no está en

el dominio del contexto, entonces el término no estará bien tipado. Por último, la regla App establece que el tipo del operador de una aplicación debe ser de la forma $\theta \to \theta'$ y el operando debe tener tipo θ bajo el mismo contexto.

Combinando las reglas de tipado se pueden construir *derivaciones* de tipos que demuestran que cierto juicio de tipado es verdadero. Por ejemplo:

Ejemplo 1 (Derivación de tipo). *Sea* $\pi' = \pi[x \mapsto \theta \to \theta'][y \mapsto \theta]$ *entonces*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\pi' \vdash x : \theta \to \theta'}\,\pi'(x) = \theta \to \theta' \qquad \dfrac{}{\pi' \vdash y : \theta}\,\pi'(y) = \theta}{\pi[x \mapsto \theta \to \theta'][y \mapsto \theta] \vdash x\,y : \theta'}}{\pi[x \mapsto \theta \to \theta'] \vdash \lambda y.\, x\,y : \theta \to \theta'}}{\pi \vdash \lambda x.\, \lambda y.\, x\,y : (\theta \to \theta') \to (\theta \to \theta')}$$

es una derivación con conclusión $\pi \vdash \lambda x.\, \lambda y.\, x\,y : (\theta \to \theta') \to (\theta \to \theta')$.

Del ejemplo anterior se desprende que un término puede tener distintos tipos bajo el mismo contexto (cuando se fijan distintos valores para las metavariables θ y θ'). Es posible también encontrar dos derivaciones distintas que tengan el mismo juicio como conclusión, como muestra el siguiente ejemplo:

Ejemplo 2 (Derivación de tipo).

$$\dfrac{\dfrac{\pi[y \mapsto \theta][x \mapsto \theta' \to \theta'] \vdash y : \theta}{\pi[y \mapsto \theta] \vdash \lambda x.\, y : (\theta' \to \theta') \to \theta} \qquad \dfrac{\pi[y \mapsto \theta][x \mapsto \theta'] \vdash x : \theta'}{\pi[y \mapsto \theta] \vdash \lambda x.\, x : \theta' \to \theta'}}{\pi[y \mapsto \theta] \vdash (\lambda x.\, y)\ (\lambda x.\, x) : \theta}$$

Si definimos, por ejemplo, $\theta' = b$ *o bien* $\theta' = b \to b$ *obtenemos derivaciones diferentes con la misma conclusión.*

Existen otros sistemas de tipos más complejos y de mayor expresividad que pueden definirse sobre el cálculo lambda, por ejemplo, tipos polimórficos y tipos recursivos [58, 83]. En la tesis, sin embargo, hemos utilizado únicamente algunas variantes del sistema de tipos simple.

Veremos a continuación cómo definir la semántica denotacional de términos bien tipados.

1.7 Semántica intrínseca

Hasta ahora sólo hemos considerado definiciones de la semántica del cálculo lambda que no tienen en cuenta el tipo de los términos. En particular, la

semántica denotacional que presentamos anteriormente está definida incluso para términos que no son tipables en el sistema de tipos simple. Ahora describiremos una manera de definir semántica en la cual el tipo de los términos tiene un rol relevante.

La semántica de los tipos puede definirse de manera extrínseca o bien de manera intrínseca [102]. En la semántica extrínseca, cada tipo tiene asociado un conjunto de valores que lo representa (por ejemplo, si tuviésemos el tipo **int**, el conjunto de valores puede ser el de los números enteros). Además, para cada tipo se tiene una relación de equivalencia que identifica a aquellos términos que denotan el mismo valor. Con este método, la semántica de los términos se mantiene independiente del sistema de tipos.

En la semántica intrínseca, por otro lado, se describe el significado de las derivaciones de tipos en lugar de los términos del lenguaje. Como consecuencia, cuando se usa este método, únicamente tienen semántica los términos bien tipados. Nos enfocamos a continuación en la semántica denotacional intrínseca.

La semántica denotacional intrínseca del cálculo lambda con el sistema de tipos simple puede definirse usando teoría de conjuntos. En el Capítulo 6 veremos que al extender el lenguaje con un operador de punto fijo vuelven a ser necesarios los dominios para definir la semántica.

Comenzaremos definiendo la semántica de los tipos. Cada tipo θ tiene asociado un conjunto. Al tipo básico b es posible asociarle cualquier conjunto D, y al tipo $\theta \to \theta'$ se le asocia el espacio de funciones entre el conjunto de θ y el conjunto de θ'.

Definición 16 (Semántica de tipos).

$$\begin{aligned} [\![b]\!] &= D \\ [\![\theta \to \theta']\!] &= [\![\theta]\!] \to [\![\theta']\!] \end{aligned}$$

La semántica de un contexto π es un conjunto cuyos elementos son funciones que asignan a cada variable $x \in dom(\pi)$ un elemento en el conjunto $[\![\pi(x)]\!]$.

Definición 17 (Semántica de contextos).

$$[\![\pi]\!] = \prod_{x \in dom(\pi)} [\![\pi(x)]\!]$$

Los elementos $\rho \in [\![\pi]\!]$ se denominan ambientes, pues al igual que los elementos de D^* de la Sección 1.5, se utilizan para determinar la semántica de las variables libres.

Por otro lado, la semántica de una derivación de tipos cuya conclusión es el juicio $\pi \vdash t : \theta$ se define mediante una función en el conjunto $[\![\pi]\!] \to [\![\theta]\!]$.

Definición 18 (Semántica intrínseca).

$$\begin{aligned}
[\![\lambda x.\, t]\!]_{\pi,\, \theta \to \theta'}\, \rho &= \widehat{\lambda}\, d\,.\, [\![t]\!]_{\pi[x \mapsto \theta],\, \theta'}\, (\rho\,[x \mapsto d]) \\
[\![x]\!]_{\pi,\, \theta}\, \rho &= \rho(x) \\
[\![t\ t']\!]_{\pi,\, \theta'}\, \rho &= [\![t]\!]_{\pi,\, \theta \to \theta'}\, ([\![t']\!]_{\pi,\, \theta}\, \rho)
\end{aligned}$$

Esta semántica cumple con una propiedad llamada *coherencia*: todas las derivaciones con la misma conclusión tienen la misma semántica. Por esta propiedad, es posible utilizar sin ambigüedad la notación $[\![t]\!]_{\pi,\, \theta}$ para denotar la semántica de todas las derivaciones con conclusión $\pi \vdash t : \theta$. Para profundizar en el concepto de coherencia referimos al lector a [97].

1.8 Índices de De Bruijn

En los capítulos siguientes utilizaremos una notación llamada índices de *De Bruijn* [30] que permite representar los términos de un lenguaje sin utilizar nombres de variables. Con esa notación se pueden manipular sintácticamente los términos del lenguaje sin preocuparse por la equivalencia de términos módulo renombre de variables. La conveniencia del método se ve reflejada, por ejemplo, cuando se compilan los términos del cálculo a las instrucciones de una máquina abstracta, y también en la formalización del cálculo en lenguajes con tipos dependientes como Coq.

Con este método, la representación de una variable del cálculo lambda es un número natural que indica cuál es la abstracción que produce su ligamiento. El índice 0 indica que la variable está ligada por la abstracción más interna que la contiene. De la misma manera, el índice $n + 1$ está asociado con la abstracción más interna que contiene a las abstracciones asociadas con los índices desde 0 hasta n. Utilizamos la notación $\overline{n}$ para denotar a la variable con índice n.

Ejemplo 3 (Representación usando índices de De Bruijn).

$$\begin{array}{lcl}
\lambda x.\, \lambda y.\, x\ y & \hookrightarrow & \lambda\, \lambda\, \overline{1}\ \overline{0} \\
(\lambda x.\, \lambda y.\, y\ x)\ (\lambda y.\, \lambda z.\, \lambda x.\, (z\ y)\ (x\ y)) & \hookrightarrow & (\lambda\, \lambda\, \overline{0}\ \overline{1})\ (\lambda\, \lambda\, \lambda\, (\overline{1}\ \overline{2})\ (\overline{0}\ \overline{2}))
\end{array}$$

Notemos que la traducción descarta las ocurrencias ligadoras de las variables, es decir, el término $\lambda x.\, x$ se presenta como $\lambda\, \overline{0}$, y la primera ocurrencia de x no se tiene en cuenta. Es claro también que dos términos sintácticamente diferentes (como $\lambda x.\, x$ y $\lambda z.\, z$) pueden tener la misma representación utilizando los índices de De Bruijn. Es posible definir una operación de sustitución apropiada para esta representación de los términos [30]. Sin embargo, como se mencionó

anteriormente, la utilización de entornos de variables nos permite trabajar sin necesidad de usar sustituciones sintácticas.

Respecto al sistema de tipos del cálculo lambda, la forma en la que se estructuran los juicios de tipado debe adaptarse a la representación de De Bruijn. En particular, los contextos pueden representarse usando listas de tipos (en lugar de mapeos de variables a tipos). El tipo asociado a la variable $\bar{n}$ es el tipo en la n-ésima posición del contexto. Las reglas de tipado pueden expresarse entonces con las siguiente definición.

Definición 19 (Reglas de tipado, usando índices De Bruijn).

$$\text{(Abs)}\ \frac{\theta :: \pi \vdash t : \theta'}{\pi \vdash \lambda t : \theta \to \theta'} \qquad \text{(Var)}\ \frac{}{\pi \vdash \bar{n} : \theta}\ \pi \cdot n = \theta$$

$$\text{(App)}\ \frac{\pi \vdash t : \theta \to \theta' \qquad \pi \vdash t' : \theta}{\pi \vdash t\ t' : \theta'}$$

Notación sobre listas: Como en el caso de los contextos, frecuentemente usaremos listas de elementos y operaciones sobre las mismas. Se utilizará $[]$ para denotar la lista vacía y $x :: xs$ para el constructor usual de listas donde x es el primer elemento, y xs el resto de la lista. La longitud de la lista xs se escribe $|xs|$. Cuando $n < |xs|$ se usará $xs \cdot n$ para denotar al elemento que se encuentra en la posición n-ésima de xs, donde el primer elemento está en la posición 0.

Los índices de De Bruijn es sólo uno de los varios métodos existentes para codificar variables, que son útiles para facilitar la compilación o bien para formalizar el lenguaje en algún asistente de demostración [17, 31, 33, 91]. En general, para toda representación de las variables, existen formas bien conocidas de traducir el lenguaje con variables nombradas a los términos de dicha representación.

Capítulo 2

Compilación Certificada Usando Semántica Small-step

La prueba de corrección de un compilador debe establecer una relación entre la semántica del lenguaje fuente y la semántica del lenguaje objeto. La definición de esta relación dependerá de la manera en que se describe la semántica del lenguaje fuente, ya sea de modo operacional o denotacional.

Comenzaremos con una prueba de corrección del compilador basada en semántica operacional small-step [95], también llamada semántica de transiciones, o semántica de reducción. Cuando se usa este tipo de semántica se definen diferentes reglas que describen cómo reducir paso a paso un término hasta obtener, si fuese posible, un valor.

En este capítulo tomaremos como lenguaje fuente al cálculo lambda con evaluación normal, cuya semántica operacional puede describirse por medio del Cálculo de Clausuras [44]. Aquí usaremos una variante de ese cálculo que fue propuesto por Biernacka y Danvy [24]. Los términos del cálculo serán compilados a una secuencia de instrucciones de la máquina de Krivine [71], y demostraremos la corrección de esa compilación.

2.1 El cálculo de clausuras

El cálculo de clausuras es una versión del cálculo lambda con sustituciones explícitas. Esto significa que su principal diferencia con respecto a la presentación usual del cálculo lambda es la manera en que se emplean las sustituciones. En el cálculo lambda puro, la sustitución es una operación que pertenece al meta-lenguaje, mientras que en el cálculo de clausuras la sustitución se integra al lenguaje y se representa como un término equipado con un entorno.

En particular, un término cerrado del cálculo lambda se puede ver como una clausura si lo equipamos con un entorno vacío. Como la sustitución se maneja de forma explícita, se vuelve necesario incorporar nuevas reglas para lograr la reducción de un redex. A continuación presentamos los términos del cálculo de clausuras.

Definición 20 (Términos y clausuras).

$$\begin{array}{lcll} t,\ t' \in \mathit{Term} & ::= & \bar{n} \mid \lambda\, t \mid t\ t' & \textit{(Términos)} \\ c,\ c' \in \mathit{Clos} & ::= & (t,\ e) \mid c\ c' & \textit{(Clausuras)} \\ e \in \mathit{Env} & ::= & [] \mid c :: e & \textit{(Entornos)} \end{array}$$

Los términos son los usuales del cálculo lambda donde se utilizan índices de De Bruijn (Sección 1.8) para representar variables. Una clausura puede ser un término equipado con un entorno o bien la aplicación de dos clausuras. En el trabajo de Curien [44] sólo la primera producción de la gramática de las clausuras está presente, la segunda fue una extensión introducida por Biernacka y Danvy [24] que facilita la definición de la semántica operacional del lenguaje.

Por otra parte, un entorno puede verse como una lista de clausuras, y representa una sustitución para todas las variables libres de un término. Por ejemplo, en la clausura $(t,\ e)$, las ocurrencias libres de la variable $\bar{0}$ en el término t están asociadas con la primer clausura de la lista e.

A continuación damos las reglas de contracción del cálculo de clausuras, enfocándonos en aquellas que permiten la reducción de una clausura siguiendo la estrategia de evaluación normal.

Definición 21 (Reglas de contracción).

$$(\beta)\ \frac{}{(\lambda\, t,\ e)\ c\ \to\ (t,\ c :: e)} \qquad (\textsc{App})\ \frac{}{(t\ t',\ e)\ \to\ (t,\ e)\ (t',\ e)}$$

$$(\textsc{Var})\ \frac{}{(\bar{n},\ e)\ \to\ e \cdot n}\ n < |e| \qquad (\nu)\ \frac{c\ \to\ c''}{c\ c'\ \to\ c''\ c'}$$

La regla β contrae un redex colocando el operando como primer elemento del entorno, asociándolo de esta manera con las ocurrencias libres de la variable $\bar{0}$ en el cuerpo de la abstracción. La regla App contrae una clausura cuyo término es una aplicación, produciendo una aplicación de clausuras donde el entorno inicial se propaga tanto al operador como al operando. En la regla Var se contrae una clausura cuyo término es una variable $\bar{n}$, produciendo como resultado la clausura que se encuentra en la n-ésima posición del entorno. Por último, la regla (ν) permite reducir una aplicación de clausuras hasta obtener

un redex, admitiendo únicamente la contracción del operador y no del operando, respetando de esa manera la estrategia de evaluación normal. Es fácil ver que este sistema de reglas es determinista:

Lema 2 (Determinismo). *Si $c \to c_1$ y $c \to c_2$, entonces $c_1 = c_2$, para todas las clausuras c, c_1 y c_2.*

Resulta de particular interés considerar aquellas reducciones que terminan en una clausura *bloqueante* (es decir, una clausura a partir de la cual no es posible aplicar ninguna de las reglas de contracción). En general, la noción de elemento bloqueante se puede definir para cualquier relación $R \subseteq A \times A$ como sigue.

Definición 22 (Bloqueante). *Si $R \subseteq A \times A$, se dice que un elemento $a \in A$ es bloqueante si no existe ningún $b \in A$ tal que $a\ R\ b$.*

Cuando una clausura c se puede reducir a una clausura bloqueante c', decimos que c *converge* a c'. Esta noción también se define de manera general.

Definición 23 (Convergencia). *Si $R \subseteq A \times A$, se dice que a converge a b, si $a\ R^*\ b$ y b es bloqueante.*

Dada nuestra relación de contracción $\to$, a partir de la Definición 4 podemos obtener una noción de reducción finita $\to^*$. Aquí $c \to^* c'$ establece que c reduce a c' en 0 o más pasos de contracción. Si quisiéramos exigir que la reducción de c a c' conste de al menos un paso de contracción, podemos usar la clausura transitiva $\to^+$:

Definición 24 (Clausura transitiva). *Si $R \subseteq A \times A$ es una relación, entonces la relación $R^+ \subseteq A \times A$ se define como sigue:*

$$a\ R^+\ b \quad \text{sí y sólo si} \quad \text{existe } a' \text{ tal que } a\ R\ a' \text{ y } a'\ R^*\ b.$$

A continuación veremos que, usando coinducción, es posible definir una noción de reducción infinita $\to^\infty$.

2.2 Coinducción y divergencia

Hasta aquí hemos usado frecuentemente definiciones inductivas basadas en un sistema de reglas. Por ejemplo, en la Definición 21, la relación $\to$ fue definida como la menor relación en el conjunto $Clos \times Clos$ cerrada por las reglas de contracción. Además de las definiciones inductivas, los sistemas de reglas

permiten también una interpretación *coinductiva*. Para introducir ese concepto, consideremos el siguiente sistema de reglas:

$$\frac{}{[] \in L_A} \qquad \frac{l \in L_A}{a :: l \in L_A}\ a \in A$$

La interpretación inductiva de este sistema define a L_A como el conjunto de todas las listas finitas de tipo A. Intuitivamente, el conjunto L_A contiene la mínima cantidad de elementos que son necesarios para satisfacer ambas reglas. Por otro lado, la interpretación coinductiva define a L_A como el conjunto de todas las listas (finitas e infinitas) de tipo A. En ese caso, L_A es el mayor conjunto *consistente* bajo ese sistema de reglas, es decir, es el conjunto más grande tal que todos sus elementos pueden obtenerse mediante la aplicación sucesiva (y potencialmente infinita) de cualquiera de las reglas. La existencia del "menor conjunto cerrado" y del "mayor conjunto consistente" para ciertos sistemas de reglas está garantizada por el teorema de punto fijo de Tarski [114]. Referimos a Aczel [6] para una formulación matemática de la noción de inferencia sobre sistemas de reglas y a Leroy [78] o Jacobs et al. [63] para una introducción más completa a la noción de coinducción.

De la misma manera en que las funciones recursivas "recorren" estructuras finitas definidas de manera inductiva, las funciones *corecursivas* "construyen" estructuras potencialmente infinitas definidas de manera coinductiva. Por ejemplo, la lista infinita $S(n) \in L_{\mathbb{N}}$ que contiene a todos los números naturales mayores o iguales que n se define mediante corecursión como sigue:

$$\begin{aligned} &S : \mathbb{N} \mapsto L_{\mathbb{N}} \\ &S(n) = n :: S(n+1)\ . \end{aligned}$$

Para asegurarse de que una función corecursiva es efectivamente una buena definición, algunos lenguajes de programación funcionales como Agda [2] y Coq [3] sólo permiten que las llamadas corecursivas aparezcan *protegidas* con un constructor. Es decir, así como las llamadas recursivas deben "decrecer" en su argumento, las funciones corecursivas deben "extender" su resultado mediante la aplicación de un constructor. Por ejemplo, la llamada a $S(n+1)$ aparece como segundo argumento del constructor de listas $\_ :: \_$ y de esa forma la lista infinita $S(n)$ es una extensión de $S(n+1)$. Dado que hemos formalizado todas las demostraciones en Coq, adoptaremos dicha restricción en nuestras pruebas coinductivas.

Regresando a la semántica small-step, usaremos coinducción para definir el conjunto $\rightarrow^{\infty} \subseteq \mathit{Clos}$ de clausuras divergentes. En realidad, la noción de divergencia se puede definir para cualquier relación binaria R como sigue.

Definición 25 (Divergencia). *Sea $R \subseteq A \times A$ una relación, entonces R^∞ se define de forma coinductiva por la siguiente regla.*

$$(\infty\textsc{Step})\ \frac{a\ R\ b \qquad b\ R^\infty}{a\ R^\infty}$$

Notación: La línea punteada indica que la regla debe interpretarse de manera coinductiva.

A modo de ejemplo, demostraremos que la clausura $(\delta\,\delta,\ [])$ donde $\delta = \lambda\,\overline{0}\ \overline{0}$ pertenece a $\to^\infty$. Pero antes de eso demostraremos dos lemas necesarios para la prueba.

Lema 3. *Si $a\,R^{+\infty}$, entonces $a\,R^\infty$.*

Prueba. La demostración es coinductiva. Nuestra hipótesis coinductiva es:

$$H : \text{para todo } a, \text{ si } a\,R^{+\infty} \text{ entonces } a\,R^\infty\ ,$$

que sólo puede usarse de manera *protegida*, es decir, se debe aplicar la regla ∞Step y usar H para construir su segunda premisa. Suponiendo $a\,R^{+\infty}$, sabemos que existe b tal que $a\,R^+\,b$ y $b\,R^\infty$. Por definición existe a' tal que $a\,R\,a'$ y $a'\,R^*\,b$. Como $b\,R^\infty$ y $a'\,R^*\,b$, se puede demostrar que $a'\,R^{+\infty}$. Por lo tanto, aplicando ∞Step y H obtenemos:

$$(\infty\textsc{Step})\ \frac{a\ R\ a' \qquad H\ \dfrac{a'\ R^{+\infty}}{a'\ R^\infty}}{a\ R^\infty}$$

□

Lema 4. *Si $(t,\ e) \to^* (\delta,\ e')$ entonces $(\overline{0}\ \overline{0},\ (t,\ e) :: e'') \to^{+\infty}$ (cualquiera sea la elección de los entornos e, e' y e'').*

Prueba. La prueba se realiza por coinducción. Sea $c = (t,\ e)$. Es fácil demostrar que $(\overline{0}\ \overline{0},\ c :: e'') \to^+ c'$ donde $c' = (\overline{0}\ \overline{0},\ (\overline{0},\ c :: e'') :: e')$. Como además se tiene $(\overline{0},\ c :: e'') \to c \to^* (\delta,\ e')$, tenemos por hipótesis coinductiva $c' \to^{+\infty}$. Por lo tanto aplicando (∞Step) concluimos:

$$(\infty\textsc{Step})\ \frac{(\overline{0}\ \overline{0},\ c :: e'') \to^+ c' \qquad c' \to^{+\infty}}{(\overline{0}\ \overline{0},\ c :: e'') \to^{+\infty}}$$

□

Lema 5. $(\delta\ \delta,\ [\,]) \to^{\infty}$.

Prueba. Por Lema 3 alcanza con ver $(\delta\ \delta,\ [\,]) \to^{+\infty}$. Es sencillo demostrar que $(\delta\ \delta,\ [\,]) \to^{+} (\bar{0}\ \bar{0},\ (\delta,\ [\,])::[\,])$. Por Lema 4 se tiene $(\bar{0}\ \bar{0},\ (\delta,\ [\,])::[\,]) \to^{+\infty}$ y por lo tanto aplicando ∞Step obtenemos $(\delta\ \delta,\ [\,]) \to^{+\infty}$. □

En la Sección 2.5 usaremos coinducción para demostrar la corrección del compilador de clausuras divergentes. Nos enfocaremos ahora en describir la compilación de los términos y la ejecución de las instrucciones resultantes en una máquina abstracta.

2.3 La máquina de Krivine

La máquina de Krivine [71] es un modelo de ejecución abstracto que permite evaluar términos del cálculo lambda siguiendo la estrategia de evaluación normal. Los términos del cálculo de clausuras serán compilados a instrucciones de la máquina; que luego pueden ejecutarse siguiendo una serie de reglas de transición. La ejecución se detiene cuando no es posible aplicar ninguna de las reglas. El valor final de la evaluación se obtiene a partir de la configuración final de la máquina.

Continuamos con la definición de las instrucciones de la máquina y de los componentes de la misma.

Definición 26 (Componentes).

$i \in \mathit{Code}$	$::=$	`Access` n	*(Código)*
	$\mid$	`Grab` $\triangleright$ i	
	$\mid$	`Push` i' $\triangleright$ i	
$\alpha \in \mathit{MClos}$	$::=$	(i, η)	*(Clausura de máquina)*
$\eta \in \mathit{MEnv}$	$::=$	$[\,] \mid \alpha :: \eta$	*(Entorno de máquina)*
$s \in \mathit{Stack}$	$::=$	$[\,] \mid \alpha :: s$	*(Pila)*
$w \in \mathit{Conf}$	$::=$	$(i,\ \eta,\ s)$	*(Configuración)*

La máquina posee tres instrucciones, cada una se corresponde con uno de los constructores del cálculo lambda (variable, abstracción y aplicación). Una clausura (de máquina) es un par cuyo primer componente es código, y el segundo un entorno (de máquina). Si bien tanto los entornos como las pilas son listas de clausuras, tienen roles diferentes durante la ejecución. La función del entorno es almacenar la información correspondiente a las variables libres. La pila se utiliza para almacenar temporalmente los operandos de una aplicación

hasta que sea necesario evaluarlos. La configuración de la máquina incluye a los tres componentes: código, entorno y pila.

La acción de las instrucciones sobre la configuración de la máquina se describe con las siguientes reglas de transición.

Definición 27 (Transiciones).

$$
\begin{array}{lcl}
(\texttt{Access}\ n,\ \eta,\ s) & \longmapsto & (i',\ \eta',\ s) \quad si\ n < |\eta|\ y\ \eta \cdot n = (i', \eta') \\
(\texttt{Grab} \triangleright i,\ \eta,\ \alpha :: s) & \longmapsto & (i,\ \alpha :: \eta,\ s) \\
(\texttt{Push}\ i' \triangleright i,\ \eta,\ s) & \longmapsto & (i,\ \eta,\ (i', \eta) :: s)
\end{array}
$$

La instrucción $\texttt{Access}\ n$ comienza la ejecución de la clausura asociada con la posición n-ésima del entorno. La instrucción $\texttt{Grab} \triangleright i$ toma una clausura del tope de la pila y la inserta en el entorno, continuando con la ejecución de i. Finalmente, la instrucción $\texttt{Push}\ i' \triangleright i$ inserta la clausura (i', η) (donde η es el entorno actual) en el tope de la pila y continua con la ejecución de i.

2.4 Compilación

Los términos del cálculo de clausuras pueden compilarse a instrucciones de la máquina abstracta. La función de *compilación* $(\!|\ \_\ |\!) : \mathit{Term} \mapsto \mathit{Code}$ se define por inducción estructural en el término.

Para demostrar algunas propiedades útiles de la máquina abstracta, necesitaremos también trabajar con la inversa de la función de compilación, que llamaremos función de *decompilación*.

Definición 28 (Compilación y decompilación de términos).

$$
\begin{array}{ll}
(\!|\ \_\ |\!) : \mathit{Term} \to \mathit{Code} & \mathit{Term}\{\!|\ \_\ |\!\} : \mathit{Code} \to \mathit{Term} \\
(\!|\ \overline{n}\ |\!) = \texttt{Access}\ n & \mathit{Term}\{\!|\ \texttt{Access}\ n\ |\!\} = \overline{n} \\
(\!|\ \lambda\, t\ |\!) = \texttt{Grab} \triangleright (\!|\ t\ |\!) & \mathit{Term}\{\!|\ \texttt{Grab} \triangleright i\ |\!\} = \lambda\, \mathit{Term}\{\!|\ i\ |\!\} \\
(\!|\ t\ t'\ |\!) = \texttt{Push}\ (\!|\ t'\ |\!) \triangleright (\!|\ t\ |\!) & \mathit{Term}\{\!|\ \texttt{Push}\ i' \triangleright i\ |\!\} = \mathit{Term}\{\!|\ i\ |\!\}\ \mathit{Term}\{\!|\ i'\ |\!\}
\end{array}
$$

Extendemos de forma homomórfica la definición de la función de decompilación para clausuras y entornos de la máquina.

Definición 29 (Decompilación de clausuras).

$$\begin{aligned} &Clos\{\!|\ \_\ |\!\} : MClos \to Clos \\ &Clos\{\!|\ (i,\ \eta)\ |\!\} \quad = (Term\{\!|\ i\ |\!\},\ Env\{\!|\ \eta\ |\!\}) \end{aligned}$$

Definición 30 (Decompilación de entornos).

$$\begin{aligned} &Env\{\!|\ \_\ |\!\} : MEnv \to Env \\ &Env\{\!|\ []\ |\!\} \quad\quad = \ [] \\ &Env\{\!|\ \alpha :: \eta\ |\!\} \ = \ Clos\{\!|\ \alpha\ |\!\} :: Env\{\!|\ \eta\ |\!\} \end{aligned}$$

Es posible obtener un término del cálculo de clausuras mediante la decompilación de una configuración. La configuración $(i,\ \eta,\ s)$ se decompila a una aplicación donde el operador es el término $Clos\{\!|\ (i,\ \eta)\ |\!\}$ y todas las clausuras de la pila s se decompilan y se colocan como operandos.

Definición 31 (Decompilación de configuraciones).
Sea $s = \alpha_1 :: \alpha_2 :: \ldots :: \alpha_n :: []$, *entonces*

$$\{\!|\ (i,\ \eta,\ s)\ |\!\} = Clos\{\!|\ (i,\ \eta)\ |\!\}\ Clos\{\!|\ \alpha_1\ |\!\} \ldots Clos\{\!|\ \alpha_n\ |\!\}\ .$$

El objetivo es demostrar que la función de compilación preserva semántica. Para lograr ese objetivo primero será necesario probar algunas propiedades útiles de la máquina abstracta.

2.5 Corrección del compilador

Antes de demostrar la corrección del compilador analizaremos algunas propiedades conocidas de la máquina de Krivine respecto de la semántica small-step del cálculo de clausuras.

En primer lugar, se puede probar que una transición de la máquina se corresponde con una contracción del cálculo. Las clausuras involucradas en la contracción se obtienen mediante la decompilación de las configuraciones correspondientes.

Lema 6. *Si* $w \longmapsto w'$, *entonces* $\{\!|\ w\ |\!\} \to \{\!|\ w'\ |\!\}$.

Prueba. La demostración se realiza por casos en la relación $\longmapsto$ (Definición 27). Mostraremos la prueba para el caso $(\mathsf{Push}\ i' \triangleright i,\ \eta,\ s) \longmapsto (i,\ \eta,\ (i', \eta) :: s)$. Supongamos $s = \alpha_1 :: \alpha_2 :: \ldots :: \alpha_n :: []$, donde cada α_k es una clausura de máquina,

entonces se tiene:

$$
\begin{aligned}
&\{\!|\ (\mathsf{Push}\ i' \triangleright i, \eta, s)\ |\!\} && = \\
&\mathit{Clos}\{\!|\ (\mathsf{Push}\ i' \triangleright i, \eta)\ |\!\}\ \mathit{Clos}\{\!|\ \alpha_1\ |\!\}\ \ldots\ \mathit{Clos}\{\!|\ \alpha_n\ |\!\} && = \\
&(\mathit{Term}\{\!|\ \mathsf{Push}\ i' \triangleright i\ |\!\}, \mathit{Env}\{\!|\ \eta\ |\!\})\ \mathit{Clos}\{\!|\ \alpha_1\ |\!\}\ \ldots\ \mathit{Clos}\{\!|\ \alpha_n\ |\!\} && = \\
&(\mathit{Term}\{\!|\ i\ |\!\}\ \mathit{Term}\{\!|\ i'\ |\!\}, \mathit{Env}\{\!|\ \eta\ |\!\})\ \mathit{Clos}\{\!|\ \alpha_1\ |\!\}\ \ldots\ \mathit{Clos}\{\!|\ \alpha_n\ |\!\} && \to \\
&(\mathit{Term}\{\!|\ i\ |\!\}, \mathit{Env}\{\!|\ \eta\ |\!\})\ (\mathit{Term}\{\!|\ i'\ |\!\}, \mathit{Env}\{\!|\ \eta\ |\!\})\ \mathit{Clos}\{\!|\ \alpha_1\ |\!\}\ \ldots\ \mathit{Clos}\{\!|\ \alpha_n\ |\!\} && = \\
&(\mathit{Term}\{\!|\ i\ |\!\}, \mathit{Env}\{\!|\ \eta\ |\!\})\ \mathit{Clos}\{\!|\ (i', \eta)\ |\!\}\ \mathit{Clos}\{\!|\ \alpha_1\ |\!\}\ \ldots\ \mathit{Clos}\{\!|\ \alpha_n\ |\!\} && = \\
&\mathit{Clos}\{\!|\ (i, \eta)\ |\!\}\ \mathit{Clos}\{\!|\ (i', \eta)\ |\!\}\ \mathit{Clos}\{\!|\ \alpha_1\ |\!\}\ \ldots\ \mathit{Clos}\{\!|\ \alpha_n\ |\!\} && = \\
&\{\!|\ (i, \eta, (i', \eta) :: s)\ |\!\}\ . &&
\end{aligned}
$$

Por lo tanto $\{\!|\ (\mathsf{Push}\ i' \triangleright i, \eta, s)\ |\!\} \to \{\!|\ (i, \eta, (i', \eta) :: s)\ |\!\}$. □

Por otro lado, una contracción del cálculo también se corresponde con una transición, en el sentido que establece el siguiente lema.

Lema 7. *Si $\{\!|\ w\ |\!\} \to c'$, entonces existe una configuración w' tal que $w \longmapsto w'$.*

Prueba. La demostración es un simple análisis por casos en la forma de la configuración w. A modo de ejemplo, supongamos $w = (\mathsf{Grab} \triangleright i, \eta, s)$. Si $s = []$ se tiene que $\{\!|\ w\ |\!\} = (\lambda\, \mathit{Term}\{\!|\ i\ |\!\}, \mathit{Env}\{\!|\ \eta\ |\!\})$ es bloqueante (ya que no hay contracciones que partan de una abstracción), lo cual contradice la hipótesis $\{\!|\ w\ |\!\} \to c'$. Luego la pila s tiene la forma $\alpha :: s'$. Es claro entonces que si definimos $w' = (i, \alpha :: \eta, s')$ tenemos $w \longmapsto w'$. □

Es posible demostrar una versión más fuerte del lema anterior, usando el determinismo de las reglas de contracción.

Lema 8. *Si $\{\!|\ w\ |\!\} \to c'$, entonces existe una configuración w' tal que $w \longmapsto w'$ y $\{\!|\ w'\ |\!\} = c'$.*

Prueba. Dado que $\{\!|\ w\ |\!\} \to c'$, por Lema 7 tenemos que existe una configuración w' tal que $w \longmapsto w'$. Luego por Lema 6 se tiene $\{\!|\ w\ |\!\} \to \{\!|\ w'\ |\!\}$. Por Lema 2 concluimos $\{\!|\ w'\ |\!\} = c'$. □

Podemos demostrar un resultado análogo donde consideramos la clausura reflexiva-transitiva de la relación de contracción y de transición.

Lema 9. *Si $\{\!|\ w\ |\!\} \to^* c'$, entonces existe una configuración w' tal que $w \longmapsto^* w'$ y $\{\!|\ w'\ |\!\} = c'$.*

Prueba. La demostración se realiza por inducción estructural en la reducción $\{\!|\ w\ |\!\} \to^* c'$. Se analiza por casos cuál fue la última regla de la Definición 4 que fue aplicada para construir dicha reducción.

- Caso (*Refl). Es trivial ya que $c' = \{\!|\, w \,|\!\}$ y por lo tanto si tomamos $w' = w$ queda demostrado el caso.
- Caso (*Step). En este caso tenemos que existe c'' tal que $\{\!|\, w \,|\!\} \to c''$ y $c'' \to^* c'$. Por Lema 8 tenemos que existe w'' tal que $w \longmapsto w''$ y además $\{\!|\, w'' \,|\!\} = c''$. Por lo tanto $\{\!|\, w'' \,|\!\} = c'' \to^* c'$. Por hipótesis inductiva existe w' tal que $w'' \longmapsto^* w'$ y $\{\!|\, w' \,|\!\} = c'$. Obtenemos entonces $w \longmapsto w'' \longmapsto^* w'$ y $\{\!|\, w' \,|\!\} = c'$ como queríamos demostrar.

□

Escribimos $c \downarrow^* c'$ cuando c converge a c' respecto a la relación $\to$, y escribimos $w \mathbin{\downarrow\!\!\!\downarrow^*} w'$ cuando w converge a w' respecto a la relación $\longmapsto$ (véase Definición 23). Cuando la clausura c' del Lema 9 es bloqueante, se puede ver que la configuración w' también lo es.

Lema 10. *Si* $\{\!|\, w \,|\!\} \downarrow^* c'$*, entonces existe* w' *tal que* $w \mathbin{\downarrow\!\!\!\downarrow^*} w'$ *y* $\{\!|\, w' \,|\!\} = c'$.

Prueba. Suponiendo $\{\!|\, w \,|\!\} \to^* c'$ con c' bloqueante, por el Lema 9 tenemos que existe w' tal que $w \longmapsto^* w'$ y $\{\!|\, w' \,|\!\} = c'$. Si w' es bloqueante, concluimos la demostración. Si no lo es, existe w'' tal que $w' \longmapsto w''$. Por Lema 6 tenemos que $c' = \{\!|\, w' \,|\!\} \to \{\!|\, w'' \,|\!\}$, lo cual contradice el hecho de que c' es bloqueante. □

El teorema de corrección del compilador establece una correspondencia entre la reducción de un término cerrado t y la ejecución del código $(\!|\, t \,|\!)$. Si $(t, [])$ converge a una clausura c', entonces $((\!|\, t \,|\!), [], [])$ converge a una configuración que decompila a c'.

Teorema 1. *Si* $(t, []) \downarrow^* c'$*, entonces existe* w' *tal que* $((\!|\, t \,|\!), [], []) \mathbin{\downarrow\!\!\!\downarrow^*} w'$ *y además* $\{\!|\, w' \,|\!\} = c'$.

Prueba. Se puede demostrar que $\{\!|\, ((\!|\, t \,|\!), [], []) \,|\!\} = (t, [])$, por lo tanto este teorema es un caso particular del Lema 10. □

El Teorema 1 sólo establece la corrección del compilador para términos cerrados que convergen a una clausura bloqueante. Sin embargo, para demostrar completamente la corrección del compilador, es necesario analizar aquellos términos cerrados t tales que $(t, [])$ diverge.

Una clausura *diverge* si durante su reducción es siempre posible aplicar alguna de las reglas de contracción, y por lo tanto nunca se alcanza una clausura bloqueante. Como un caso particular de la Definición 25, cuando una clausura c diverge con respecto a la relación de contracción, escribimos $c \to^\infty$. De la misma forma, si una configuración w diverge con respecto a la relación de transición, escribimos $w \longmapsto^\infty$. El siguiente resultado establece que si $\{\!|\, w \,|\!\}$ diverge, también lo hace w.

Lema 11. *Si* $\{\!|\ w\ |\!\} \to^{\infty}$ *entonces* $w \longmapsto^{\infty}$.

Prueba. Procedemos por coinducción. La hipótesis coinductiva es:

$$H : \text{para todo } w, \text{ si } \{\!|\ w\ |\!\} \to^{\infty} \text{ entonces } w \longmapsto^{\infty}.$$

Como $\{\!|\ w\ |\!\} \to^{\infty}$ existe c' tal que $\{\!|\ w\ |\!\} \to c'$ y $c' \to^{\infty}$. Por Lema 8 tenemos que existe w' tal que $w \longmapsto w'$ y $\{\!|\ w'\ |\!\} = c'$. Por lo tanto se cumple $c' = \{\!|\ w'\ |\!\} \to^{\infty}$. Aplicando ∞Step obtenemos:

$$(\infty\textsc{Step})\ \frac{w \longmapsto w' \qquad H\ \dfrac{\{\!|\ w'\ |\!\} \to^{\infty}}{w' \longmapsto^{\infty}}}{w \longmapsto^{\infty}}$$

□

Finalmente, el siguiente teorema establece la corrección del compilador para términos cerrados divergentes.

Teorema 2. *Si* $(t, [\,]) \to^{\infty}$ *entonces* $(\langle\!|\ t\ |\!\rangle, [\,], [\,]) \longmapsto^{\infty}$.

Prueba. Consecuencia directa del Lema 11 y de que además se puede demostrar $\{\!|\ (\langle\!|\ t\ |\!\rangle, [\,], [\,])\ |\!\} = (t, [\,])$. □

Se puede observar que la función de decompilación facilita la demostración de las propiedades de la máquina abstracta respecto a las reglas de contracción y por ende a la demostración del teorema de corrección. Para otros lenguajes, sin embargo, la función de decompilación no es fácil de definir y puede ser necesario establecer relaciones de bisimulación entre la semántica operacional y las reglas de transición de la máquina abstracta [56, 103] si se desea obtener una prueba de corrección respecto a la semántica small-step.

Para lenguajes más sofisticados, la semántica big-step ofrece pruebas más simples de la corrección de compiladores [78]. En capítulos siguientes usaremos semántica big-step para demostrar la corrección del compilador de dos lenguajes: cálculo lambda de evaluación normal con operadores aritméticos estrictos, y un lenguaje imperativo de alto orden.

Capítulo 3

Compilación Certificada Usando Semántica Big-step

La semántica operacional de un lenguaje se define a partir de un conjunto de reglas que describen cómo reducir un término hasta obtener un valor o una forma canónica. En la semántica small-step, cada regla describe un único paso de computación, y la reducción consiste en repetir sucesivamente esos pasos hasta obtener, de ser posible, el valor del término. Por otro lado, en la semántica big-step, las reglas describen la evaluación completa del término, sin explicitar los pasos intermedios involucrados en la reducción.

En este capítulo se presenta una prueba de corrección de un compilador basada en la semántica big-step del lenguaje fuente. Se ha utilizado un enfoque similar al de Leroy [78], con la diferencia principal de que nuestro lenguaje tiene evaluación normal y el entorno de ejecución es una variante de la máquina abstracta de Krivine en lugar de la máquina SECD [73].

3.1 Sintaxis y semántica

El lenguaje que usaremos en este capítulo es el cálculo lambda extendido con constantes enteras y el operador de suma. A continuación presentamos la definición de los términos del lenguaje.

Definición 32 (Términos).

$$t,\ t' \in \mathit{Term} := \bar{n} \mid \lambda\, t \mid t\ t' \mid \underline{k} \mid t + t'$$

Sólo consideramos el operador aritmético de suma para ilustrar el comportamiento de la máquina abstracta durante la evaluación. Es trivial extender los

resultados a otros operadores aritméticos. En los capítulos siguientes, los operadores aritméticos se presentarán de manera más general.

Las clausuras y los entornos del lenguaje se definen de la siguiente forma:

Definición 33 (Clausuras y Entornos).

$$c \in \mathit{Clos} ::= (t,\ e) \qquad e \in \mathit{Env} ::= [] \mid c :: e$$

A diferencia de la Definición 20, notemos que aquí no hay aplicación de clausuras, esto es consecuencia de pasar de una semántica de reducción, a una semántica big-step, donde los valores intermedios de una computación no pueden observarse. Los únicos valores observables son los resultantes de la evaluación de un término, que en nuestro caso pueden ser una abstracción junto con su entorno o una constante numérica.

Definición 34 (Valores).

$$v \in \mathit{Val} ::= (\lambda\, t,\ e) \mid k$$

Definimos ahora la semántica big-step del lenguaje. Un juicio de la forma $e \vdash t \Rightarrow v$ establece que el término t evalúa a v bajo el entorno e.

Definición 35 (Semántica big-step).

$$\text{(Abs)}\ \frac{}{e \vdash \lambda\, t \Rightarrow (\lambda\, t,\ e)} \qquad \text{(Const)}\ \frac{}{e \vdash \underline{k} \Rightarrow k}$$

$$\text{(App)}\ \frac{e \vdash t \Rightarrow (\lambda\, t'',\ e') \qquad (t',\ e) :: e' \vdash t'' \Rightarrow v}{e \vdash t\ t' \Rightarrow v}$$

$$\text{(Var)}\ \frac{e' \vdash t' \Rightarrow v}{e \vdash \bar{n} \Rightarrow v}\ e \cdot n = (t',\ e') \qquad \text{(Add)}\ \frac{e \vdash t \Rightarrow k \qquad e \vdash t' \Rightarrow k'}{e \vdash t + t' \Rightarrow k + k'}$$

Las reglas para la abstracción y las constantes son triviales, ya que las formas canónicas evalúan a sí mismas sin cambiar el entorno. Notemos que en la regla de la aplicación el operando no es evaluado sino que forma parte del entorno extendido bajo el cual se evalúa el cuerpo de la abstracción.

En la regla Var observamos que para obtener el valor de una variable $\bar{n}$ se debe evaluar la clausura en la n-ésima posición del entorno. La regla de la suma es bastante sencilla, uno primero debe evaluar los dos argumentos y finalmente obtener el valor final sumando las dos constantes. Ilustramos las reglas de evaluación con los siguientes ejemplos.

Ejemplo 4 (Evaluación de un redex).

$$\dfrac{e \vdash (\lambda\, \lambda\, t) \Rightarrow (\lambda\, \lambda\, t,\ e) \qquad (t',\ e) :: e \vdash \lambda\, t \Rightarrow (\lambda\, t,\ (t',\ e) :: e)}{e \vdash (\lambda\, \lambda\, t)\ t' \Rightarrow (\lambda\, t,\ (t',\ e) :: e)}$$

En este ejemplo se reduce el redex $(\lambda\, \lambda\, t)\ t'$ a una abstracción $\lambda\, t$, cuyo entorno contiene la clausura $(t',\ e)$ en el primer elemento. El operando t' no es evaluado, sino que es puesto directamente en el entorno, y la clausura $(t',\ e)$ queda asociada con las ocurrencias libres de la variable $\overline{1}$ del término t, el cuerpo de la abstracción.

Ejemplo 5 (Evaluación de una expresión entera). *Sea* $t = \overline{0} + \underline{3}$ *entonces:*

$$\dfrac{\dfrac{}{e \vdash \lambda\, t \Rightarrow (\lambda\, t,\ e)} \qquad \dfrac{\dfrac{e \vdash \underline{2} \Rightarrow 2}{(\underline{2},\ e) :: e \vdash \overline{0} \Rightarrow 2} \qquad \dfrac{}{(\underline{2},\ e) :: e \vdash \underline{3} \Rightarrow 3}}{(\underline{2},\ e) :: e \vdash \overline{0} + \underline{3} \Rightarrow 5}}{e \vdash (\lambda\, t)\ \underline{2} \Rightarrow 5}$$

En este ejemplo utilizamos todas las reglas de evaluación para obtener el valor del redex $(\lambda\, (\overline{0} + \underline{3}))\ \underline{2}$. El valor obtenido es la constante 5. Notemos cómo al combinar varias reglas de evaluación se obtiene un *árbol* o *derivación* que permite observar cada aplicación de las reglas necesarias para obtener el valor resultante.

A pesar de que en estos ejemplos se logra encontrar un valor final (una abstracción o una constante), existen otros términos que no son evaluables con este sistema de reglas. En el siguiente ejemplo se muestra que al intentar evaluar el término $\delta\ \delta$ con $\delta = \lambda\, \overline{0}\ \overline{0}$ no se puede construir el árbol de evaluación, ya que siempre es posible aplicar alguna de las reglas:

Ejemplo 6 (Término no evaluable). *Sea* $e' = (\delta,\ e) :: e$ *entonces:*

$$\dfrac{\dfrac{}{e \vdash \delta \Rightarrow (\delta,\ e)} \qquad \dfrac{\dfrac{\overline{e' \vdash \overline{0} \Rightarrow (\delta,\ e)} \qquad \dfrac{\vdots}{(\overline{0},\ e') :: e \vdash \overline{0}\ \overline{0} \Rightarrow \_}}{e' \vdash \overline{0}\ \overline{0} \Rightarrow \_}}{e \vdash \delta\ \delta \Rightarrow \_}$$

Los puntos suspensivos indican que el árbol de evaluación podría crecer indefinidamente, puesto que siempre hay reglas de evaluación aplicables. Con la notación $e \vdash t \Rightarrow \_$ indicamos que el valor del término t es indeterminado, es decir, que no es posible completar su evaluación.

Si al intentar evaluar un término el árbol de evaluación es infinito (o crece indefinidamente), se dice que el término *diverge*. Leroy [78] muestra que usando coinducción es posible definir reglas que capturen la noción de divergencia de manera más rigurosa. Para ello define varias reglas coinductivas para cada uno de los constructores del lenguaje. Aquí definimos reglas similares, aunque adaptadas para nuestro lenguaje con evaluación normal. Escribimos $e \vdash t \Rightarrow \infty$ para denotar la divergencia de un término t bajo un entorno e.

Definición 36 (Semántica big-step coinductiva).

$$\text{(App-l)}\ \frac{e \vdash t \Rightarrow \infty}{e \vdash t\ t' \Rightarrow \infty} \qquad \text{(App-r)}\ \frac{e \vdash t \Rightarrow (\lambda\, t'',\ e') \qquad (t',\ e) :: e' \vdash t'' \Rightarrow \infty}{e \vdash t\ t' \Rightarrow \infty}$$

$$\text{(Var)}\ \frac{e' \vdash t' \Rightarrow \infty}{e \vdash \bar{n} \Rightarrow \infty}\ e \cdot n = (t',\ e')$$

$$\text{(Add-l)}\ \frac{e \vdash t \Rightarrow \infty}{e \vdash t + t' \Rightarrow \infty} \qquad \text{(Add-r)}\ \frac{e \vdash t \Rightarrow k \qquad e \vdash t' \Rightarrow \infty}{e \vdash t + t' \Rightarrow \infty}$$

Recordemos que la línea entrecortada indica que las reglas se interpretan de manera coinductiva. Hay dos posibles razones por las que una aplicación $t\ t'$ puede diverger, que fueron reflejadas en las reglas App-l y App-r. La primera posibilidad es que el término t diverge. La segunda es que el término t evalúe a una abstracción $(\lambda\, t'',\ e')$, pero la evaluación del término t'' diverge. Notar que, dado que estamos usando evaluación normal, no hacemos ninguna afirmación con respecto a si la evaluación del argumento t' diverge o no. La interpretación para el resto de las reglas es similar. A modo de ejemplo veamos que $\delta\ \delta$ diverge para cualquier entorno e.

Lema 12. $e \vdash \delta\ \delta \Rightarrow \infty$.

Prueba. Si aplicamos App-r obtenemos:

$$\text{(App-r)}\ \frac{e \vdash \delta \Rightarrow (\lambda\, \bar{0}\ \bar{0},\ e) \qquad (\delta,\ e) :: e \vdash \bar{0}\ \bar{0} \Rightarrow \infty}{e \vdash \delta\ \delta \Rightarrow \infty}$$

La prueba de $(\delta,\ e) :: e \vdash \bar{0}\ \bar{0} \Rightarrow \infty$ es consecuencia directa del Lema 13. □

Lema 13. *Si* $e' \vdash t' \Rightarrow (\delta,\ e'')$ *entonces* $(t',\ e') :: e \vdash \bar{0}\ \bar{0} \Rightarrow \infty$.

Prueba. Se demuestra usando coinducción. Sea $\widetilde{e} = (t', e') :: e$, entonces obtenemos:

$$\text{(App-r)}\ \frac{\text{(Var)}\ \dfrac{e' \vdash t' \Rightarrow (\delta,\ e'')}{\widetilde{e} \vdash \bar{0} \Rightarrow (\delta,\ e'')} \qquad (\bar{0},\ \widetilde{e}) :: e'' \vdash \bar{0}\ \bar{0} \Rightarrow \infty}{\widetilde{e} \vdash \bar{0}\ \bar{0} \Rightarrow \infty}$$

La prueba de $(\bar{0},\ \widetilde{e}) :: e'' \vdash \bar{0}\ \bar{0} \Rightarrow \infty$ se obtiene por hipótesis coinductiva a partir del hecho de que $\widetilde{e} \vdash \bar{0} \Rightarrow (\delta,\ e'')$. □

Hasta aquí hemos presentado el lenguaje fuente y su semántica operacional big-step, procedemos a continuación con la descripción de la máquina abstracta.

3.2 La máquina abstracta

Extenderemos la máquina de Krivine con las instrucciones necesarias para implementar las constantes enteras y la suma. La máquina de Krivine sigue la estrategia de evaluación normal, esto implica que el operando de una aplicación se evalúa sólo cuando se necesita. Pero si queremos incorporar operadores estrictos, como la suma, necesitamos una manera de forzar la evaluación de los argumentos antes de computar el resultado de la operación. Para ello usaremos una estructura de datos llamada *frame*, que se ha utilizado antes en otros trabajos, por ejemplo [110]. Esta estructura es útil para almacenar el código que se necesita para computar el valor de cada argumento de los operadores. A continuación mostramos los diferentes componentes de la máquina abstracta:

Definición 37 (Componentes).

$i,\ i' \in \mathit{Code}$	$::=$	$\texttt{Access}\ n$	*(Código)*
	$\mid$	$\texttt{Grab} \triangleright i$	
	$\mid$	$\texttt{Push}\ i' \triangleright i$	
	$\mid$	$\texttt{Const}\ k$	
	$\mid$	$\texttt{Add}$	
$\alpha \in \mathit{MClos}$	$::=$	$(i,\ \eta)$	*(Clausura de máquina)*
$\eta \in \mathit{MEnv}$	$::=$	$[] \mid \alpha :: \eta$	*(Entorno de máquina)*
$\mu \in \mathit{StackVal}$	$::=$	$\alpha \mid \{+ \bullet\ \alpha\} \mid \{+\ k\ \bullet\}$	*(Valor de pila)*
$s \in \mathit{Stack}$	$::=$	$[] \mid \mu :: s$	*(Pila)*
$w \in \mathit{Conf}$	$::=$	$(i,\ \eta,\ s)$	*(Configuración)*

Al igual que en la Definición 26, una clausura es un par compuesto por código junto con su entorno, donde el entorno es una lista de clausuras. Esta vez,

la pila puede almacenar tanto clausuras como frames. El frame $\{+ \bullet \alpha\}$ almacena la clausura necesaria para computar el segundo argumento de la suma; esta clausura permanece almacenada en la pila dentro del frame mientras se computa el valor del primer argumento. Cuando finalmente se conoce el valor k del primer argumento, se lo almacena en un frame $\{+\ k\ \bullet\}$ y se comienza con la evaluación del segundo. Intuitivamente, el símbolo $\bullet$ indica el lugar donde se almacenará el valor del argumento que se está evaluando. En el siguiente capítulo generalizaremos la definición de frames para soportar operadores estrictos de cualquier aridad finita.

Recordemos que la Definición 27 tiene una regla de transición por cada instrucción. Con el agregado de la suma, necesitamos, sin embargo, tres reglas nuevas.

Definición 38 (Transiciones).

$$
\begin{array}{lll}
(\texttt{Access}\ n,\ \eta,\ s) & \longmapsto & (i',\ \eta',\ s) \\
 & & si\ n < |\eta|\ y\ \eta \cdot n = (i',\ \eta') \\
(\texttt{Grab} \triangleright i,\ \eta,\ \alpha :: s) & \longmapsto & (i,\ \alpha :: \eta,\ s) \\
(\texttt{Push}\ i' \triangleright i,\ \eta,\ s) & \longmapsto & (i,\ \eta,\ (i',\ \eta) :: s) \\
(\texttt{Add},\ \eta,\ \alpha' :: \alpha :: s) & \longmapsto & (i',\ \eta',\ \{+ \bullet \alpha\} :: s) \\
 & & si\ \alpha' = (i',\ \eta') \\
(\texttt{Const}\ k,\ \eta,\ \{+ \bullet \alpha'\} :: s) & \longmapsto & (i',\ \eta',\ \{+\ k \bullet\} :: s) \\
 & & si\ \alpha' = (i',\ \eta') \\
(\texttt{Const}\ k,\ \eta,\ \{+\ k' \bullet\} :: s) & \longmapsto & (\texttt{Const}\ (k + k'),\ \eta,\ s)
\end{array}
$$

La instrucción Add espera en el tope de la pila una clausura para cada uno de los argumentos de la suma. Esta instrucción coloca en el tope de la pila un nuevo frame con la clausura del segundo argumento, y comienza con la ejecución de la clausura correspondiente al primero. Para el caso de la instrucción Const k, hay dos nuevas reglas de transición, que surgen de dos escenarios distintos: cuando k es el valor del primer argumento de la suma, y cuando k es el valor del segundo argumento. En el primer caso, se ejecutará el código α' almacenado en el frame, y se actualiza el frame con la constante k. En el segundo caso, se toma el valor del primer argumento k' del frame y se ejecuta Const $(k + k')$.

3.3 Compilación

Continuamos con la definición de la función de compilación. Esta función define una traducción de los términos del lenguaje a instrucciones de la máquina abstracta.

Definición 39 (Compilación de términos).

$$
\begin{aligned}
(\!|\ \_\ |\!) &: \mathit{Term} \to \mathit{Code} \\
(\!|\ \overline{n}\ |\!) &= \mathtt{Access}\ n \\
(\!|\ \lambda\, t\ |\!) &= \mathtt{Grab} \triangleright (\!|\ t\ |\!) \\
(\!|\ t\ t'\ |\!) &= \mathtt{Push}\ (\!|\ t'\ |\!) \triangleright (\!|\ t\ |\!) \\
(\!|\ \underline{k}\ |\!) &= \mathtt{Const}\ k \\
(\!|\ t_1 + t_2\ |\!) &= \mathtt{Push}\ (\!|\ t_2\ |\!) \triangleright (\mathtt{Push}\ (\!|\ t_1\ |\!) \triangleright \mathtt{Add})
\end{aligned}
$$

Los términos del cálculo lambda se compilan de la misma forma que en el Capítulo 2. La compilación de un término $\underline{k}$ es simplemente la instrucción `Const` k. La compilación de una suma consiste en dos instrucciones `Push` con el código de la compilación de ambos argumentos, seguidos de la instrucción `Add`. Durante la ejecución, las instrucciones `Push` insertan el código de los argumentos de la suma en la pila, y la instrucción `Add` comienza a ejecutar el código del primer argumento, construyendo a su vez un frame que almacena el código del segundo.

El enunciado del teorema de corrección del compilador (que veremos en la siguiente sección) establece una correspondencia entre la compilación de un término y el resultado de su evaluación. Dado que la evaluación de un término depende de un entorno, resulta necesario definir una traducción entre los entornos de evaluación y los entornos de máquina. Extendemos entonces la definición del compilador para clausuras y entornos como sigue:

Definición 40 (Compilación de clausuras y entornos).

$$
\begin{aligned}
&\mathit{MClos}(\!|\ \_\ |\!) : \mathit{Clos} \to \mathit{MClos} \\
&\mathit{MClos}(\!|\ (t,\ e)\ |\!) \quad = \quad ((\!|\ t\ |\!),\ \mathit{MEnv}(\!|\ e\ |\!)) \\
&\mathit{MEnv}(\!|\ \_\ |\!) : \mathit{Env} \to \mathit{MEnv} \\
&\mathit{MEnv}(\!|\ []\ |\!) \quad = \quad [] \\
&\mathit{MEnv}(\!|\ c :: e\ |\!) \quad = \quad \mathit{MClos}(\!|\ c\ |\!) :: \mathit{MEnv}(\!|\ e\ |\!)
\end{aligned}
$$

Aquí las funciones $\mathit{MClos}(\!|\ \_\ |\!)$ y $\mathit{MEnv}(\!|\ \_\ |\!)$ son mutuamente recursivas. La compilación de una clausura (a nivel del lenguaje fuente) es una clausura de máquina, compuesta por el código del término y el código del entorno. Por el otro lado, la compilación de un entorno se obtiene compilando cada clausura que se encuentra dentro del mismo.

Para ilustrar cómo se ejecuta el código resultante de la compilación, presentamos los mismos términos de los Ejemplos 4 y 5, mostrando paso a paso la ejecución del código correspondiente.

Ejemplo 7 (Ejecución del código $(\!|\ (\lambda\,\lambda\,t)\ t'\ |\!)$).

$$(\!|\ (\lambda\,\lambda\,t)\ t'\ |\!) = \mathsf{Push}\ (\!|\ t'\ |\!) \triangleright \mathsf{Grab} \triangleright \mathsf{Grab} \triangleright (\!|\ t\ |\!)$$

$$\begin{array}{lll}
(\mathsf{Push}\ (\!|\ t'\ |\!) \triangleright \mathsf{Grab} \triangleright \mathsf{Grab} \triangleright (\!|\ t\ |\!), & \eta, & s) \\
\longmapsto & & \\
(\mathsf{Grab} \triangleright \mathsf{Grab} \triangleright (\!|\ t\ |\!), & \eta, & (\!|\ t'\ |\!), \eta) :: s) \\
\longmapsto & & \\
(\mathsf{Grab} \triangleright (\!|\ t\ |\!), & (\!|\ t'\ |\!), \eta) :: \eta, & s)
\end{array}$$

Ejemplo 8 (Ejecución del código $(\!|\ (\lambda\,(\bar{0} + \underline{3}))\ \underline{2}\ |\!)$).

$$(\!|\ (\lambda\,(\bar{0} + \underline{3}))\ \underline{2}\ |\!) =$$
$$\mathsf{Push}\ (\mathsf{Const}\ 2) \triangleright \mathsf{Grab} \triangleright \mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}$$

$$\begin{array}{lll}
(\mathsf{Push}\ (\mathsf{Const}\ 2) \triangleright \ldots, & \eta, & s) \\
\longmapsto & & \\
(\mathsf{Grab} \triangleright \ldots, & \eta, & (\mathsf{Const}\ 2, \eta) :: s) \\
\longmapsto & & \\
(\mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \ldots, & \eta', & s) \\
\textit{donde}\ \eta' = (\mathsf{Const}\ 2, \eta) :: \eta & & \\
\longmapsto & & \\
(\mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, & \eta', & (\mathsf{Const}\ 3, \eta') :: s) \\
\longmapsto & & \\
(\mathsf{Add}, & \eta', & s') \\
\textit{donde}\ s' = (\mathsf{Access}\ 0, \eta') :: (\mathsf{Const}\ 3, \eta') :: s & & \\
\longmapsto & & \\
(\mathsf{Access}\ 0, & \eta', & \{+ \bullet \alpha\} :: s) \\
\textit{donde}\ \alpha = (\mathsf{Const}\ 3, \eta') & & \\
\longmapsto & & \\
(\mathsf{Const}\ 2, & \eta, & \{+ \bullet \alpha\} :: s) \\
\longmapsto & & \\
(\mathsf{Const}\ 3, & \eta', & \{+\ 2\ \bullet\} :: s) \\
\longmapsto & & \\
(\mathsf{Const}\ 5, & \eta', & s)
\end{array}$$

Tenemos entonces definida la función de compilación, ahora nos enfocamos en demostrar su corrección con respecto a la semántica big-step (inductiva y coinductiva).

3.4 Corrección del compilador

El Ejemplo 8 sugiere el siguiente comportamiento general de la máquina abstracta: si un término t evalúa a una constante k, entonces, la ejecución del código $(\!| t |\!)$ con la pila inicial s, alcanza una configuración de la forma $(\texttt{Const}\ k,\ \eta',\ s)$ para algún entorno η'. De manera similar, el Ejemplo 7 se puede generalizar como sigue: si un término t evalúa a una clausura $(\lambda\, t',\ e')$ en un entorno e, entonces, la ejecución de $(\!| t |\!)$ en el entorno $MEnv(\!| e |\!)$ alcanza la configuración $(\texttt{Grab} \triangleright (\!| t' |\!),\ MEnv(\!| e' |\!),\ s)$.

Este comportamiento puede expresarse de manera general con el siguiente enunciado:

Teorema 3. *Para cualquier $e \in Env$, $t \in Term$ y $v \in Val$, si $e \vdash t \Rightarrow v$ entonces para todo $s \in Stack$, se cumple:*

- *si $v = k$ para alguna constante k, entonces*

$$(\!| t |\!),\ MEnv(\!| e |\!),\ s) \longmapsto^* (\texttt{Const}\ k,\ \eta',\ s)$$

  para algún $\eta' \in MEnv$,

- *si $v = (\lambda\, t',\ e')$ para algún $t' \in Term$ y $e' \in Env$, entonces*

$$(\!| t |\!),\ MEnv(\!| e |\!),\ s) \longmapsto^* (\texttt{Grab} \triangleright (\!| t' |\!),\ MEnv(\!| e' |\!),\ s)\ .$$

El enunciado de este teorema puede simplificarse significativamente definiendo una relación $\rightarrowtail\ \subseteq\ Conf \times Val$ que denominamos relación de *alcanzabilidad.*

Definición 41 (Alcanzabilidad).

$$\begin{aligned}
&(\alpha,\ s) \rightarrowtail k \qquad \textit{si y sólo si}\\
&\quad (\alpha,\ s) \longmapsto^* (\texttt{Const}\ k,\ \eta',\ s)\ \textit{para algún}\ \eta' \in MEnv\\
&(\alpha,\ s) \rightarrowtail (\lambda\, t,\ e) \quad \textit{si y sólo si}\\
&\quad (\alpha,\ s) \longmapsto^* (\texttt{Grab} \triangleright (\!| t |\!),\ MEnv(\!| e |\!),\ s)\ .
\end{aligned}$$

Notación: Cuando $\alpha = (i,\ \eta) \in MClos$ escribimos $(\alpha,\ s) \in Conf$ a la configuración $(i,\ \eta,\ s)$.

Es trivial demostrar que la relación de alcanzabilidad es cerrada por antiejecución:

Lema 14. *Para cualquier $\alpha, \alpha' \in MClos$, $s \in Stack$ y $v \in Val$, si $(\alpha,\ s) \longmapsto^* (\alpha',\ s)$ y $(\alpha',\ s) \rightarrowtail v$, entonces $(\alpha,\ s) \rightarrowtail v$.*

La relación $\rightarrowtail$ lleva a una prueba más simple del Teorema 3, facilitando también la generalización del enunciado en el caso de que se agregaran más valores al lenguaje (por ejemplo, valores booleanos) manteniendo el enunciado del teorema sin cambios.

Teorema 4. *Para cualquier $e \in Env$, $t \in Term$ y $v \in Val$, si $e \vdash t \Rightarrow v$ entonces para todo $s \in Stack$, $(MClos(\!| (t, e) |\!), s) \rightarrowtail v$.*

Prueba. El teorema se demuestra por inducción en la estructura de la derivación $e \vdash t \Rightarrow v$. Ilustraremos la demostración en dos casos: para la regla Const y la regla App (Definición 35).

En el caso de la regla $e \vdash \underline{k} \Rightarrow k$, tenemos

$$(MClos(\!| (\underline{k}, e) |\!), s) = (\texttt{Const}\ k, MEnv(\!| e |\!), s) \longmapsto^* (\texttt{Const}\ k, MEnv(\!| e |\!), s) \ ,$$

para todo $s \in Stack$, y por lo tanto $(MClos(\!| (\underline{k}, e) |\!), s) \rightarrowtail k$.

Ahora analicemos el caso de la aplicación, recordemos la regla App:

$$(\textsc{App})\ \frac{e \vdash t \Rightarrow (\lambda\, t'', e') \qquad (t', e) :: e' \vdash t'' \Rightarrow v}{e \vdash t\ t' \Rightarrow v}$$

Tenemos una hipótesis inductiva para cada premisa de la regla. En este caso tenemos:

(i) para todo s', $(MClos(\!| (t, e) |\!), s') \rightarrowtail (\lambda\, t'', e')$

(ii) para todo s', $(MClos(\!| (t'', (t', e) :: e') |\!), s') \rightarrowtail v$.

Luego, por definición de $\rightarrowtail$ y (i), obtenemos

(iii) para todo s', $(MClos(\!| (t, e) |\!), s') \longmapsto^* (\texttt{Grab} \triangleright (\!| t'' |\!), MEnv(\!| e' |\!), s')$.

Usando el Lema 14, comenzaremos con la configuración $(MClos(\!| (t\ t', e) |\!), s)$ e intentaremos llegar a la configuración $(MClos(\!| (t'', (t', e) :: e') |\!), s)$, la cual sabemos por (ii) que está relacionada vía $\rightarrowtail$ con v.

$$
\begin{array}{lll}
 & (MClos(\!| (t\ t', e) |\!), & s) \\
= & & \\
(|\!| t\ t' |\!), & MEnv(\!| e |\!), & s) \\
= & & \\
(\mathsf{Push}\ (|\!| t' |\!) \triangleright (|\!| t |\!)), & MEnv(\!| e |\!), & s) \\
\longmapsto & & \\
(|\!| t |\!), & MEnv(\!| e |\!), & (|\!| t' |\!), MEnv(\!| e |\!)) :: s) \\
= & & \\
 & (MClos(\!| (t, e) |\!), & (|\!| t' |\!), MEnv(\!| e |\!)) :: s) \\
\longmapsto^* & & \\
(\mathsf{Grab} \triangleright (|\!| t'' |\!)), & MEnv(\!| e' |\!), & (|\!| t' |\!), MEnv(\!| e |\!)) :: s) \\
\longmapsto & & \\
(|\!| t'' |\!), & (|\!| t' |\!), MEnv(\!| e |\!)) :: MEnv(\!| e' |\!), & s) \\
= & & \\
(|\!| t'' |\!), & MClos(\!| (t', e) |\!) :: MEnv(\!| e' |\!), & s) \\
= & & \\
(|\!| t'' |\!), & MEnv(\!| (t', e) :: e' |\!), & s) \\
= & & \\
 & (MClos(\!| (t, (t', e) :: e') |\!), & s)\ .
\end{array}
$$

Y esto finaliza la prueba para la regla App. Para el resto de las reglas, la demostración es similar. □

Hay una tercera manera de enunciar la corrección del compilador que es un poco más intuitiva: se define la compilación de valores y se demuestra que la compilación de los términos se ejecuta hasta alcanzar la compilación de sus valores. Veremos que el precio a pagar por obtener un enunciado más intuitivo será cambiar una de las reglas de transición.

Definición 42 (Compilación de valores).

$$
\begin{aligned}
Val(\!|\ \_\ |\!) &: Val \to MClos \\
Val(\!|\ (\lambda t, e)\ |\!) &= (\mathsf{Grab} \triangleright (|\!| t |\!), MEnv(\!| e |\!)) \\
Val(\!|\ k\ |\!) &= (\mathsf{Const}\ k, [])
\end{aligned}
$$

La versión alternativa del teorema de corrección se puede formular como sigue:

Enunciado. *Para todo $e \in Env$, $t \in Term$ y $v \in Val$, si $e \vdash t \Rightarrow v$, entonces, para todo $s \in Stack$, $(MClos(\!|\ (t, e)\ |\!), s) \longmapsto^* (Val(\!|\ v\ |\!), s)$.*

La demostración falla para el caso del término $\underline{k}$, ya que $(MClos(\!| (\underline{k},\ e) |\!),\ s)$ no necesariamente alcanza la configuración $(Val(\!| \underline{k} |\!),\ s)$ por el hecho de que el entorno e puede no ser vacío. Si se incorpora la siguiente regla de transición para vaciar el entorno, es posible completar la demostración del teorema.

$$(\texttt{Const}\ k,\ \alpha :: \eta,\ s) \longmapsto (\texttt{Const}\ k,\ [],\ s)\ .$$

Hasta aquí sólo hemos demostrado la corrección de la compilación de términos convergentes. Para completar la prueba de corrección, necesitamos asegurarnos que, cuando la evaluación de un término t diverge, la ejecución de $(\!| t |\!)$ también lo hace. La definición coinductiva de las secuencias infinitas de transiciones se presentó en la Definición 25.

Teorema 5. *Para cualquier $e \in Env$, $t \in Term$, si $e \vdash t \Rightarrow \infty$, entonces se tiene $(MClos(\!| (t,\ e) |\!),\ s) \longmapsto^{\infty}$, para toda pila $s \in Stack$.*

Prueba. Aplicando Lema 3, basta con ver $(MClos(\!| (t,\ e) |\!),\ s) \longmapsto^{+\infty}$ para toda pila $s \in Stack$. Se demuestra usando coinducción, analizando por casos la estructura de la derivación $e \vdash t \Rightarrow \infty$. Como ejemplo, consideremos el caso de la regla App-r:

$$\text{(App-r)}\ \frac{e \vdash\ t \Rightarrow (\lambda\, t'',\ e') \qquad (t',\ e) :: e' \vdash t'' \Rightarrow \infty}{e \vdash t\ t' \Rightarrow \infty}$$

Tomemos $s \in Stack$, debemos demostrar que $(MClos(\!| (t\ t',\ e) |\!),\ s) \longmapsto^{+\infty}$. Es fácil verificar que $(MClos(\!| (t\ \ t',\ e) |\!),\ s) \longmapsto (MClos(\!| (t,\ e) |\!),\ s')$ donde $s' = MClos(\!| (t',\ e) |\!) :: s$.

Tenemos que $(MClos(\!| (t,\ e) |\!),\ s') \longmapsto^{*} (\texttt{Grab} \triangleright (\!| t'' |\!),\ MEnv(\!| e' |\!),\ s')$ por Teorema 4. Luego $(MClos(\!| (t\ t',\ e) |\!),\ s) \longmapsto^{+} (MClos(\!| (t'',\ (t',\ e) :: e') |\!),\ s)$. Por hipótesis coinductiva sabemos $(MClos(\!| (t'',\ (t',\ e) :: e') |\!),\ s) \longmapsto^{+\infty}$. Por lo tanto obtenemos:

$$\text{(∞Step)}\ \frac{\begin{array}{c}(MClos(\!| (t\ t',\ e) |\!),\ s) \longmapsto^{+} (MClos(\!| (t'',\ (t',\ e) :: e') |\!),\ s) \\ (MClos(\!| (t'',\ (t',\ e) :: e') |\!),\ s) \longmapsto^{+\infty}\end{array}}{(MClos(\!| (t\ t',\ e) |\!),\ s) \longmapsto^{+\infty}}$$

□

En el siguiente capítulo seguiremos un enfoque similar pero incorporando características imperativas al lenguaje.

Capítulo 4

Compilación Certificada de un Lenguaje Imperativo

En este capítulo extenderemos el lenguaje del capítulo anterior con construcciones imperativas, tales como la posibilidad de crear, modificar y leer localizaciones de memoria. Adaptaremos la máquina abstracta para poder implementar las extensiones en el lenguaje, y demostraremos la corrección del compilador con respecto a la semántica big-step.

4.1 Sintaxis y semántica

El lenguaje incluye el cálculo lambda, constantes enteras, operadores binarios aritméticos, y varios constructores imperativos. El fragmento imperativo del lenguaje incluye localizaciones (posiciones dentro del estado), un operador de dereferenciación (!), declaraciones de variables (**newvar**), composición (;), asignación (:=) e inacción (**skip**). Como antes, usaremos los índices de De Bruijn para representar variables.

Definición 43 (Términos).

$$
\begin{array}{lll}
t, t' \in \mathit{Term} & ::= & \lambda\, t \mid t\, t' \mid \overline{n} \mid \underline{k} \mid t \oplus t' \quad \textit{(Términos)} \\
& \mid & \underline{\ell} \mid !\, t \mid \textbf{\textit{newvar}}\ t \mid t; t' \mid t := t' \mid \textbf{\textit{skip}} \\
\ell \in \mathit{Loc} = \mathbb{N} & & \textit{(Localizaciones)}
\end{array}
$$

Elegimos como representación concreta de las localizaciones a los números naturales. Esta representación resulta conveniente puesto que de esa forma las localizaciones se pueden usar como índices de una lista de valores, que será

la representación elegida para el estado. Por simplicidad, las localizaciones de memoria de nuestro lenguaje únicamente pueden almacenar valores enteros.

Como un ejemplo de cómo combinar los términos del cálculo lambda con el fragmento imperativo, consideremos el siguiente término:

Ejemplo 9 (Declaración de variable y asignación).

$$\textbf{\textit{newvar}}\ (\lambda\, \overline{0}\,;\, \overline{0})\ (\overline{0} :=!\ \overline{0} \oplus \underline{1})$$

En este ejemplo, **newvar** crea una localización que queda ligada a la variable $\overline{0}$. La expresión $!\ \overline{0} \oplus \underline{1}$ es la aplicación del operador $\oplus$ con el valor de dicha localización y la constante $\underline{1}$. La asignación $\overline{0} :=!\ \overline{0} \oplus \underline{1}$ aparece como argumento de la abstracción $(\lambda\, \overline{0}\,;\, \overline{0})$ cuyo cuerpo es la composición $\overline{0}\,;\, \overline{0}$. Siguiendo las reglas de evaluación que veremos a continuación se puede verificar que la asignación anterior se evalúa dos veces debido a la composición secuencial.

La definición de clausura y entorno se mantiene sin cambios respecto al capítulo anterior.

Definición 44 (Clausuras y entornos).

$$c \in \mathit{Clos} ::= (t,\ e) \qquad e \in \mathit{Env} ::= [] \mid c :: e$$

La evaluación de un término del lenguaje imperativo depende del estado, ya que el mismo contiene el valor de las localizaciones de memoria. Representamos al estado como una lista de valores enteros, y las localizaciones son aquellas posiciones válidas de la misma. Esta representación concreta del estado ayuda a expresar fácilmente en la semántica la creación y liberación de localizaciones de memoria. Por conveniencia, la lista de valores crece hacia la derecha, manteniendo en la última posición la localización más reciente que todavía se encuentra en alcance (scope).

Definición 45 (Estado).

$$\sigma \in \mathit{State} ::= [] \mid \sigma :: k$$

El valor final que se obtiene de la evaluación de un término puede ser una abstracción (junto con su entorno), una constante entera, una localización o un estado.

Definición 46 (Valores).

$$\begin{array}{rcll} v \in \mathit{Val} & ::= & (\lambda\, t,\ e) & \textit{(Abstracciones)} \\ & \mid & k & \textit{(Constantes)} \\ & \mid & \ell & \textit{(Localizaciones)} \\ & \mid & \sigma & \textit{(Estados)} \end{array}$$

Dado que el estado debe formar parte de la evaluación de los términos, necesitamos una nueva notación para las definir las reglas de evaluación. Escribimos $(e,\ \sigma) \vdash t \Rightarrow v$ para denotar que t evalúa a v bajo el entorno e y el estado σ.

Las reglas de evaluación de los términos que no forman parte del fragmento imperativo (cálculo lambda, constantes y operadores) son similares a las presentadas en la Definición 35, excepto que todos los términos se evalúan bajo un estado inicial además del entorno.

Definición 47 (Semántica big-step).

$$\textsc{(Abs)}\ \frac{}{(e,\ \sigma) \vdash \lambda\, t \Rightarrow (\lambda\, t,\ e)} \qquad \textsc{(Const)}\ \frac{}{(e,\ \sigma) \vdash \underline{k} \Rightarrow k}$$

$$\textsc{(App)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow (\lambda\, t'',\ e') \qquad ((t',\ e) :: e',\ \sigma) \vdash t'' \Rightarrow v}{(e,\ \sigma) \vdash t\ t' \Rightarrow v}$$

$$\textsc{(Var)}\ \frac{(e',\ \sigma) \vdash t' \Rightarrow v}{(e,\ \sigma) \vdash \overline{n} \Rightarrow v}\ e \cdot n = (t',\ e')$$

$$\textsc{(BinOp)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow k \qquad (e,\ \sigma) \vdash t' \Rightarrow k'}{(e,\ \sigma) \vdash t \oplus t' \Rightarrow k \oplus k'}$$

Notemos que las reglas App y Var pueden producir un estado como valor final. Esto es necesario para evaluar términos como el presentado en Ejemplo 9, donde se combina el cálculo lambda con el fragmento imperativo. A continuación se muestran las reglas de evaluación para el resto de los términos.

Definición 48 (Semántica big-step, fragmento imperativo).

$$\text{(Loc)}\ \frac{}{(e,\ \sigma) \vdash \underline{\ell} \Rightarrow \ell}\ \ell < |\sigma| \qquad \text{(Deref)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow \ell}{(e,\ \sigma) \vdash !\ t \Rightarrow \sigma(\ell)}$$

$$\text{(Skip)}\ \frac{}{(e,\ \sigma) \vdash \textbf{\textit{skip}} \Rightarrow \sigma}$$

$$\text{(Newvar)}\ \frac{((\underline{\ell},\ e) :: e,\ \sigma :: 0) \vdash t \Rightarrow \sigma' :: k}{(e,\ \sigma) \vdash \textbf{\textit{newvar}}\ t \Rightarrow \sigma'}\ \ell = |\sigma|$$

$$\text{(Comp)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow \sigma' \qquad (e,\ \sigma') \vdash t' \Rightarrow \sigma''}{(e,\ \sigma) \vdash t;\ t' \Rightarrow \sigma''}$$

$$\text{(Assign)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow \ell \qquad (e,\ \sigma) \vdash t' \Rightarrow k}{(e,\ \sigma) \vdash t := t' \Rightarrow \sigma[\ell \mapsto k]}$$

Un término $\underline{\ell}$ evalúa a la localización ℓ (puesto que todas las localizaciones son valores). La condición de que $\ell < |\sigma|$ asegura que ℓ sea una posición válida, esto es, un número natural menor a la longitud del estado. De hecho, se puede demostrar que las localizaciones producidas por la evaluación siempre son válidas respecto al estado inicial:

Lema 15. *Si* $(e,\ \sigma) \vdash t \Rightarrow \ell$ *entonces* $\ell < |\sigma|$.

Otra propiedad de la evaluación es que nunca cambia de tamaño el estado final respecto del estado inicial. Esto es consecuencia de haber diseñado el lenguaje con *disciplina de pila*, liberando las localizaciones al finalizar cada bloque. La disciplina de pila es una característica fundamental de los lenguajes imperativos Algol-like [12, 99].

Lema 16. *Si* $(e,\ \sigma) \vdash t \Rightarrow \sigma'$ *entonces* $|\sigma| = |\sigma'|$.

Para evaluar el término $!\ t$ primero se debe evaluar t obteniendo una localización ℓ, y luego se debe buscar el valor almacenado en esa localización dentro del estado σ (a ese valor lo denotamos $\sigma(\ell)$).

Las últimas cuatro reglas tienen como valor final un estado que resulta de realizar modificaciones en el estado inicial. En particular, el término **skip** devuelve el estado inicial sin modificaciones.

El término **newvar** t crea una nueva localización $\ell = |\sigma|$ donde σ es el estado inicial. El término t se evalúa con un estado extendido $\sigma :: 0$, indicando

que el valor de la posición ℓ es inicialmente igual a 0. El entorno también es extendido, indicando que las ocurrencias libres de la variable $\overline{0}$ en el término t están asociadas con la localización ℓ. La evaluación de t produce un estado de la misma longitud que $\sigma :: 0$, pero con un valor potencialmente distinto en sus localizaciones. El último valor se descarta al finalizar la evaluación de **newvar** t, indicando que la última localización ya no está en alcance. La regla de evaluación de **newvar** t expresa de manera explícita la disciplina de pila del lenguaje: las variables declaradas dentro de un bloque son descartadas al finalizar su ejecución.

Notemos es posible acceder a la nueva localización creada por **newvar** indirectamente a través de la variable $\overline{0}$. Esto sugiere que el constructor $\underline{\ell}$ no es estrictamente necesario en el lenguaje, puesto que se puede conseguir la misma expresividad prescindiendo del mismo.

La composición $t\,;\,t'$ indica la evaluación secuencial del término t seguido de t'. Primero se evalúa t obteniendo un estado σ', que luego se utiliza como estado inicial en la evaluación de t'. A partir de esa última evaluación se obtiene el estado resultante.

La asignación $t := t'$ indica que el término t debe evaluar a una localización ℓ cuyo valor en el estado es reemplazado por el resultado de evaluar t'. Escribimos $\sigma[\ell \mapsto k]$ para denotar la operación que reemplaza el valor almacenado en la localización ℓ por el valor entero k.

Continuamos con la definición de la semántica big-step coinductiva para términos divergentes. Las siguientes reglas describen para cada constructor del lenguaje los diferentes casos en los que la semántica del término puede diverger.

Definición 49 (Semántica big-step coinductiva).

$$\text{(Var)}\ \frac{(e',\,\sigma) \vdash t' \Rightarrow \infty}{(e,\,\sigma) \vdash \overline{n} \Rightarrow \infty}\ e \cdot n = (t',\,e') \qquad \text{(App-l)}\ \frac{(e,\,\sigma) \vdash t \Rightarrow \infty}{(e,\,\sigma) \vdash t\ t' \Rightarrow \infty}$$

$$\text{(App-r)}\ \frac{(e,\,\sigma) \vdash t \Rightarrow (\lambda\, t'',\,e') \qquad ((t',\,e) :: e',\,\sigma) \vdash t'' \Rightarrow \infty}{(e,\,\sigma) \vdash t\ t' \Rightarrow \infty}$$

$$\text{(Bop-fst)}\ \frac{(e,\,\sigma) \vdash t \Rightarrow \infty}{(e,\,\sigma) \vdash t \oplus t' \Rightarrow \infty}$$

$$\text{(Bop-snd)}\ \frac{(e,\,\sigma) \vdash t \Rightarrow k \qquad (e,\,\sigma) \vdash t' \Rightarrow \infty}{(e,\,\sigma) \vdash t \oplus t' \Rightarrow \infty}$$

Definición 50 (Semántica big-step coinductiva, fragmento imperativo).

$$\text{(Deref)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow \infty}{(e,\ \sigma) \vdash\ !\ t \Rightarrow \infty} \qquad \text{(Newvar)}\ \frac{((\underline{\ell},\ e) :: e,\ \sigma :: 0) \vdash t \Rightarrow \infty}{(e,\ \sigma) \vdash \textbf{\textit{newvar}}\ t \Rightarrow \infty}$$

$$\text{(Comp-fst)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow \infty}{(e,\ \sigma) \vdash t;\ t' \Rightarrow \infty}$$

$$\text{(Comp-snd)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow \sigma' \qquad (e,\ \sigma') \vdash t' \Rightarrow \infty}{(e,\ \sigma) \vdash t;\ t' \Rightarrow \infty}$$

$$\text{(Assign-lhs)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow \infty}{(e,\ \sigma) \vdash t := t' \Rightarrow \infty}$$

$$\text{(Assign-rhs)}\ \frac{(e,\ \sigma) \vdash t \Rightarrow \ell \qquad (e,\ \sigma) \vdash t' \Rightarrow \infty}{(e,\ \sigma) \vdash t := t' \Rightarrow \infty}$$

Para ejemplificar cómo se interpretan estas reglas consideremos el caso de la asignación. La evaluación de una asignación $t := t'$ puede diverger por dos razones: cuando el término t diverge (Assign-lhs) o cuando t evalúa a una localización pero el término t' diverge (Assign-rhs).

4.2 La máquina abstracta

Aquí mostraremos que la máquina de Krivine puede extenderse para posibilitar la implementación de las características imperativas del lenguaje. A diferencia de la máquina abstracta del Capítulo 3, aquí se generaliza el tratamiento de los frames de manera que los operadores binarios, la asignación y la dereferenciación pueden implementarse uniformemente. El estado pasa a formar parte de la configuración de la máquina, y se agregan nuevas instrucciones que realizan modificaciones sobre el mismo.

Definición 51 (Instrucciones).

$$\begin{array}{rcll} i,\ i' \in \mathit{Code} & ::= & \texttt{Access}\ n & \textit{(Código)} \\ & | & \texttt{Grab} \triangleright i \mid \texttt{Push}\ i' \triangleright i \mid \texttt{Const}\ v & \\ & | & \texttt{Op} \ominus^n \mid \texttt{Frame} \ominus^n & \\ & | & \texttt{Alloc} \triangleright i \mid \texttt{Dealloc} \mid \texttt{Cont} & \end{array}$$

Definición 52 (Componentes).

$\alpha \in \mathit{MClos}$	$::=$	(i, η)	*(Clausura de máquina)*
$\eta \in \mathit{MEnv}$	$::=$	$[] \mid \alpha :: \eta$	*(Entorno de máquina)*
$\ominus^n \in \mathit{Ops}$	$::=$	$\oplus^2 \mid !^1 \mid :=^2$	*(Operador)*
$v \in \mathit{Arg}$	$::=$	$k \mid \ell$	*(Argumento)*
$\mu \in \mathit{StackVal}$	$::=$	$\alpha \mid \{\ominus^n \bar{v} \bullet \bar{\alpha}\}$	*(Valor de pila)*
$s \in \mathit{Stack}$	$::=$	$[] \mid \mu :: s$	*(Pila)*
$\sigma \in \mathit{State}$	$::=$	$[] \mid \sigma :: k$	*(Estado)*
$w \in \mathit{Conf}$	$::=$	(i, η, σ, s)	*(Configuración)*

Notemos que la pila puede contener clausuras o frames. Un frame es una estructura de datos de la forma $\{\ominus^n \bar{v} \bullet \bar{\alpha}\}$ que contiene: (a) una operador $\ominus^n$ que puede ser una operación aritmética, el operador de dereferenciación o el operador de asignación, (b) una lista $\bar{v}$ que contiene algunos de los argumentos del operador (constantes o localizaciones), y (c) una lista $\bar{\alpha}$ con las clausuras requeridas para computar el resto de los argumentos. El símbolo $\bullet$ indica el lugar donde se almacenará el valor del argumento que se está computando. Si un frame tiene la forma $\{\ominus^n \bar{v}\}$ (donde no aparece el símbolo $\bullet$) se deduce que ya se han sido computados todos los argumentos del operador y por lo tanto $|\bar{v}| = n$.

A continuación se presentan las transiciones de la máquina abstracta:

Definición 53 (Transiciones, continúa en Definición 54).

$(\texttt{Access}\ n, \eta, \sigma, s)$	$\longmapsto$	(i', η', σ, s) *si* $n < \lvert\eta\rvert$ *y* $\eta \cdot n = (i', \eta')$
$(\texttt{Grab} \triangleright i, \eta, \sigma, \alpha :: s)$	$\longmapsto$	$(i, \alpha :: \eta, \sigma, s)$
$(\texttt{Push}\ i' \triangleright i, \eta, \sigma, s)$	$\longmapsto$	$(i, \eta, \sigma, (i', \eta) :: s)$
$(\texttt{Frame} \ominus^n, \eta, \sigma, \alpha' :: \bar{\alpha} :: s)$	$\longmapsto$	$(i', \eta', \sigma, \{\ominus^n \bar{\alpha}\} :: s)$ *si* $\alpha' = (i', \eta')$, $n > 0$ *y* $\lvert\bar{\alpha}\rvert = n - 1$
$(\texttt{Op} \oplus, \eta, \sigma, \{\oplus\ k, k'\} :: s)$	$\longmapsto$	$(\texttt{Const}\ \hat{k}, \eta, \sigma, s)$ *donde* $\hat{k} = k \oplus k'$
$(\texttt{Op} :=, \eta, \sigma, \{:=\ \ell, k\} :: s)$	$\longmapsto$	$(\texttt{Cont}, \eta, \sigma', s)$ *donde* $\sigma' = \sigma[\ell \mapsto k]$
$(\texttt{Op}\ !, \eta, \sigma, \{!\ \ell\} :: s)$	$\longmapsto$	$(\texttt{Const}\ k, \eta, \sigma, s)$ *donde* $k = \sigma(\ell)$

Definición 54 (Transiciones, continuación).

$$
\begin{array}{lll}
(\texttt{Const}\ v,\ \eta,\ \sigma,\ \{\ominus^n\ \overline{v} \bullet \alpha',\ \overline{\alpha}\} :: s) & \longmapsto & (i',\ \eta',\ \sigma,\ \{\ominus^n\ \overline{v},\ v \bullet \overline{\alpha}\} :: s) \\
 & & \textit{donde } \alpha' = (i',\ \eta') \\
(\texttt{Const}\ v,\ \eta,\ \sigma,\ \{\ominus^n\ \overline{v} \bullet\} :: s) & \longmapsto & (\texttt{Op}\ \ominus^n,\ \eta,\ \sigma,\ \{\ominus^n\ \overline{v},\ v\} :: s) \\
(\texttt{Cont},\ \eta,\ \sigma,\ \alpha :: s) & \longmapsto & (i',\ \eta',\ \sigma,\ s) \\
 & & \textit{donde } \alpha = (i',\ \eta') \\
(\texttt{Alloc} \triangleright i,\ \eta,\ \sigma,\ s) & \longmapsto & (i,\ \alpha :: \eta,\ \sigma :: 0,\ s) \\
 & & \textit{donde } \alpha = (\texttt{Const}\ \ell, \eta),\ \ell = |\sigma| \\
(\texttt{Dealloc},\ \eta,\ \sigma :: k,\ s) & \longmapsto & (\texttt{Cont},\ \eta,\ \sigma,\ s)
\end{array}
$$

La instrucción $\texttt{Frame}\ \ominus^n$ espera n clausuras en el tope de la pila, correspondientes a los n argumentos del operador. Se comienza a ejecutar la primera clausura (que corresponde al primer argumento), y el resto se almacena en el tope de la pila dentro de un frame.

La instrucción $\texttt{Const}\ v$ actualiza el frame con el valor v y comienza a ejecutar la siguiente clausura almacenada en el frame, si existe alguna. Una vez que todos los argumentos hayan sido evaluados, se ejecuta la instrucción $\texttt{Op}\ \ominus^n$. Esta instrucción espera un frame con todos los argumentos evaluados, y aplica la operación asociada con el operador $\ominus^n$. Por ejemplo, si $\ominus^n$ es el operador de asignación $(:=)$, entonces $\texttt{Op}\ (:=)$ espera un frame de la forma $\{:=\ \ell,\ k\}$ y actualiza el estado en la localización ℓ con el valor k.

La acción de la instrucción `Cont` consiste simplemente en comenzar la ejecución de la primera clausura de la pila. El entorno actual es reemplazado por el entorno interno de dicha clausura.

Para crear y liberar localizaciones en el estado se utilizan las instrucciones `Alloc` y `Dealloc`, respectivamente. Ambas instrucciones aparecen en la compilación del término **newvar** como se mostrará en la siguiente sección.

4.3 Compilación

A continuación definimos la función de compilación $(\!|\ \_\ |\!) : \mathit{Term} \to \mathit{Code}$ que traduce los términos del lenguaje a las instrucciones de la máquina abstracta. La compilación del cálculo lambda y los operadores aritméticos es similar a la Definición 39, y ahora se incorporan las traducciones para el fragmento imperativo. En la siguiente sección demostraremos la corrección de dicha traducción respecto a la semántica big-step.

Definición 55 (Compilación de términos).

$$
\begin{aligned}
(\!| \lambda\, t |\!) &= \texttt{Grab} \triangleright (\!| t |\!) \\
(\!| t\ t' |\!) &= \texttt{Push}\ (\!| t' |\!) \triangleright (\!| t |\!) \\
(\!| \overline{n} |\!) &= \texttt{Access}\ n \\
(\!| \underline{k} |\!) &= \texttt{Const}\ k \\
(\!| \underline{\ell} |\!) &= \texttt{Const}\ \ell \\
(\!| t \oplus t' |\!) &= \texttt{Push}\ (\!| t' |\!) \triangleright \texttt{Push}\ (\!| t |\!) \triangleright \texttt{Frame}\ (\oplus) \\
(\!| !\ t |\!) &= \texttt{Push}\ (\!| t |\!) \triangleright \texttt{Frame}\ (!) \\
(\!| \textbf{\textit{newvar}}\ t |\!) &= \texttt{Push}\ (\texttt{Dealloc}) \triangleright \texttt{Alloc} \triangleright (\!| t |\!) \\
(\!| t;\ t' |\!) &= \texttt{Push}\ (\!| t' |\!) \triangleright (\!| t |\!) \\
(\!| t := t' |\!) &= \texttt{Push}\ (\!| t' |\!) \triangleright \texttt{Push}\ (\!| t |\!) \triangleright \texttt{Frame}\ (:=) \\
(\!| \textbf{\textit{skip}} |\!) &= \texttt{Cont}
\end{aligned}
$$

El término $\underline{\ell}$ se traduce a la instrucción `Const` ℓ, cuya acción depende del momento en que se ejecuta: puede insertar la localización ℓ en un frame o bien ejecutar la instrucción `Op` $\ominus^n$ que realiza la operación correspondiente al operador $\ominus^n$. La compilación de los términos del fragmento imperativo consiste en una secuencia de instrucciones `Push` cuya acción es insertar en el tope de la pila y en el orden adecuado todas las clausuras necesarias para realizar la operación correspondiente. La compilación del operador binario $\oplus$ comienza con dos instrucciones `Push`, una para cada argumento del operador, y finaliza con la instrucción `Frame` $\oplus$. Las dos instrucciones `Push` insertan los argumentos en el tope de la pila en el orden correcto para que la instrucción `Frame` $\oplus$ construya el frame correspondiente. El término **skip** se compila a la instrucción `Cont`, cuya única acción es ejecutar la siguiente clausura en el tope de la pila sin realizar cambios en el estado.

La función de compilación puede extenderse para clausuras y entornos, como se indica a continuación.

Definición 56 (Compilación de clausuras y entornos).

$$
\begin{aligned}
&MClos(\!|\ \_\ |\!) : Clos \to MClos \\
&MClos(\!|\ (t,\ e)\ |\!) \quad = \quad (\!| t |\!), MEnv(\!|\ e\ |\!)) \\
&MEnv(\!|\ \_\ |\!) : Env \to MEnv \\
&MEnv(\!|\ []\ |\!) \quad = \quad [] \\
&MEnv(\!|\ c :: e\ |\!) \quad = \quad MClos(\!|\ c\ |\!) :: MEnv(\!|\ e\ |\!)
\end{aligned}
$$

Esta definición se mantuvo sin cambios respecto al capítulo anterior: pa-

ra compilar una clausura se deben compilar cada uno de los componentes del par. La compilación del entorno consiste en compilar cada clausura dentro del mismo.

4.4 Corrección del compilador

La prueba de corrección del compilador sigue la misma estrategia que el Capítulo 3: se define una relación $\rightarrowtail \subseteq \mathit{Conf} \times \mathit{Val}$ de *alcanzabilidad* que indica cuál es el valor asociado con una configuración de la máquina y luego se demuestra que la compilación de un término *alcanza* el mismo valor que su evaluación.

Definición 57 (Alcanzabilidad).

$$
\begin{array}{ll}
(\alpha, \sigma, s) \rightarrowtail k & \textit{si y sólo si} \\
\quad (\alpha, \sigma, s) \longmapsto^* (\mathtt{Const}\ k, \eta', \sigma, s) & \textit{para algún } \eta' \in \mathit{MEnv} \\
(\alpha, \sigma, s) \rightarrowtail (\lambda\, t, e) & \textit{si y sólo si} \\
\quad (\alpha, \sigma, s) \longmapsto^* (\mathtt{Grab} \triangleright (\!| t |\!), \mathit{MEnv}(\!| e |\!), \sigma, s) & \\
(\alpha, \sigma, s) \rightarrowtail \ell & \textit{si y sólo si} \\
\quad (\alpha, \sigma, s) \longmapsto^* (\mathtt{Const}\ \ell, \eta', \sigma, s) & \textit{para algún } \eta' \in \mathit{MEnv} \\
(\alpha, \sigma, s) \rightarrowtail \sigma' & \textit{si y sólo si} \\
\quad (\alpha, \sigma, s) \longmapsto^* (\mathtt{Cont}, \eta', \sigma', s) & \textit{para algún } \eta' \in \mathit{MEnv}\ .
\end{array}
$$

Notación: Si $\alpha = (i, \eta)$ escribimos (α, σ, s) a la configuración (i, η, σ, s).

El siguiente teorema establece la corrección del compilador para términos convergentes.

Teorema 6. *Para todo $e \in \mathit{Env}$, $t \in \mathit{Term}$, $\sigma \in \mathit{State}$, $v \in \mathit{Val}$, si $(e, \sigma) \vdash t \Rightarrow v$ entonces, para todo $s \in \mathit{Stack}$, $(\mathit{MClos}(\!| (t, e) |\!), \sigma, s) \rightarrowtail v$.*

Prueba. La prueba se realiza por inducción estructural en la derivación del juicio $(e, \sigma) \vdash t \Rightarrow v$. Ilustraremos la demostración para el caso de la asignación:

$$
(\textsc{Assign})\ \frac{(e, \sigma) \vdash t \Rightarrow \ell \qquad (e, \sigma) \vdash t' \Rightarrow k}{(e, \sigma) \vdash t := t' \Rightarrow \sigma[\ell \mapsto k]}
$$

Tenemos una hipótesis inductiva para cada premisa de la regla:

(i) para todo s', $(\mathit{MClos}(\!| (t, e) |\!), \sigma, s') \rightarrowtail \ell$,

(ii) para todo s', $(\mathit{MClos}(\!| (t', e) |\!), \sigma, s') \rightarrowtail k$.

Por lo tanto, por definición de $\rightarrowtail$, obtenemos:

(iii) para todo $s' \in \mathit{Stack}$, $(\mathit{MClos}\langle\!|\ (t,\ e)\ |\!\rangle,\ \sigma,\ s') \longmapsto^* (\mathtt{Const}\ \ell,\ \eta_1,\ \sigma,\ s')$ para algún $\eta_1 \in \mathit{MEnv}$

(iv) para todo $s' \in \mathit{Stack}$, $(\mathit{MClos}\langle\!|\ (t',\ e)\ |\!\rangle,\ \sigma,\ s') \longmapsto^* (\mathtt{Const}\ k,\ \eta_2,\ \sigma,\ s')$ para algún $\eta_2 \in \mathit{MEnv}$.

Luego podemos obtener la siguiente secuencia de transiciones:

$$
\begin{array}{lllll}
(\mathit{MClos}\langle\!|\ (t := t',\ e)\ |\!\rangle, & & \sigma, & s) & = \\
(\langle\!|\ t := t'\ |\!\rangle, & \mathit{MEnv}\langle\!|\ e\ |\!\rangle, & \sigma, & s) & = \\
(\mathtt{Push}\ \langle\!|\ t'\ |\!\rangle \triangleright \mathtt{Push}\ \langle\!|\ t\ |\!\rangle \triangleright \mathtt{Frame}\ (:=), & \mathit{MEnv}\langle\!|\ e\ |\!\rangle, & \sigma, & s) & \longmapsto \\
(\mathtt{Push}\ \langle\!|\ t\ |\!\rangle \triangleright \mathtt{Frame}\ (:=), & \mathit{MEnv}\langle\!|\ e\ |\!\rangle, & \sigma, & s_0) & \longmapsto \\
\text{donde } s_0 = (\langle\!|\ t'\ |\!\rangle, \mathit{MEnv}\langle\!|\ e\ |\!\rangle) :: s & & & & \\
(\mathtt{Frame}\ (:=), & \mathit{MEnv}\langle\!|\ e\ |\!\rangle, & \sigma, & s_1) & \longmapsto \\
\text{donde } s_1 = (\langle\!|\ t\ |\!\rangle, \mathit{MEnv}\langle\!|\ e\ |\!\rangle) :: s_0 & & & & \\
(\langle\!|\ t\ |\!\rangle, & \mathit{MEnv}\langle\!|\ e\ |\!\rangle, & \sigma, & s_2) & = \\
\text{donde } s_2 = \{:= \bullet\ (\langle\!|\ t'\ |\!\rangle, \mathit{MEnv}\langle\!|\ e\ |\!\rangle)\} :: s & & & & \\
(\mathit{MClos}\langle\!|\ (t,\ e)\ |\!\rangle, & & \sigma, & s_2) & \longmapsto^* \\
(\mathtt{Const}\ \ell, & \eta_1, & \sigma, & s_2) & \longmapsto \\
(\langle\!|\ t'\ |\!\rangle, & \mathit{MEnv}\langle\!|\ e\ |\!\rangle, & \sigma, & s_3) & = \\
\text{donde } s_3 = \{:= \ell \bullet\} :: s & & & & \\
(\mathit{MClos}\langle\!|\ (t',\ e)\ |\!\rangle, & & \sigma, & s_3) & \longmapsto^* \\
(\mathtt{Const}\ k, & \eta_2, & \sigma, & s_3) & \longmapsto \\
(\mathtt{Op}\ (:=), & \eta_2, & \sigma, & s_4) & \longmapsto \\
\text{donde } s_4 = \{:=\ \ell,\ k\} :: s & & & & \\
(\mathtt{Cont} & \eta_2, & \sigma[\ell \mapsto k], & s) & .
\end{array}
$$

Con eso hemos probado que $(\mathit{MClos}\langle\!|\ (t := t',\ e)\ |\!\rangle,\ \sigma,\ s) \rightarrowtail \sigma[\ell \mapsto k]$. El resto de los casos son similares. □

Es posible demostrar usando coinducción que cuando un término t diverge, la ejecución del código $\langle\!|\ t\ |\!\rangle$ también lo hace.

Teorema 7. *Para todo $e \in \mathit{Env}$, $t \in \mathit{Term}$, $\sigma \in \mathit{State}$, si $(e,\ \sigma) \vdash t \Rightarrow \infty$, entonces $(\mathit{MClos}\langle\!|\ (t,\ e)\ |\!\rangle,\ \sigma,\ s) \longmapsto^\infty$ para todo $s \in \mathit{Stack}$.*

Prueba. Se demuestra usando coinducción, analizando por casos la estructura de la derivación $(e,\ \sigma) \vdash t \Rightarrow \infty$. □

A diferencia de otras formalizaciones de la semántica big-step de lenguajes imperativos (e.g [21, 77]), en este capítulo se modela un lenguaje de alto orden

que combina construcciones imperativas con aplicativas y las reglas de evaluación describen de manera explícita la disciplina de pila manteniendo un alcance local para todas las variables del programa. En comparación con Leroy [78] donde se utiliza semántica coinductiva para un lenguaje funcional estricto, aquí hemos usado evaluación normal y hemos incorporando reglas de evaluación coinductiva para las construcciones imperativas.

Con este capítulo concluye nuestro análisis de la semántica big-step, a partir del siguiente capítulo utilizaremos *realizabilidad* y semántica denotacional para estructurar las pruebas de corrección.

Capítulo 5

Realizabilidad de Krivine

En este capítulo introduciremos la noción de *realizabilidad* aplicada sobre la máquina abstracta de Krivine. Como veremos, el esquema de realizabilidad puede adaptarse para obtener la demostración de la corrección del compilador.

El concepto general de realizabilidad consiste en la interpretación de fórmulas o términos de un sistema formal por medio de objetos matemáticos con cierto contenido computacional. La construcción del objeto matemático provee evidencia de la veracidad de la fórmula asociada, obteniendo en general más información de la que se adquiere mediante una prueba meramente sintáctica o axiomática. A este conjunto de objetos se los denomina *realizadores* de la fórmula o el término, según el sistema formal que se considera.

La noción de realizabilidad surgió en 1945 con Kleene [66], que utilizó funciones computables como realizadores de las fórmulas de la aritmética de Heyting. Luego Kreisel [69] utilizó los términos del cálculo lambda tipado como realizadores, esta vez de la aritmética de Heyting de alto orden. Algunos autores consideran ambos trabajos como formalizaciones de una idea anterior denominada *interpretación de Brouwer–Heyting–Kolmogorov* (por los autores que la describieron) donde aparece por primera vez la noción de función de A en B como prueba de una implicación $A \to B$. En la famosa correspondencia de Curry-Howard [45, 60], los términos del cálculo lambda son realizadores de la lógica intuisionista, los tipos del cálculo se corresponden con las fórmulas, y los términos con su demostración. Más recientemente, a partir del descubrimiento por parte de Griffin [55] de la correspondencia entre el operador de control `call/cc` y la ley de Pierce, Krivine [72] formalizó la noción de realizabilidad para la lógica clásica y la teoría de conjuntos de Zermelo-Fraenkel. Para conocer de manera más detallada la evolución histórica de la realizabilidad se refiere al lector a [116]. Introduciremos primero la noción de realizabilidad aplicada por Krivine [72] y modificada por Jaber [61], y a partir de la Sección 5.3 presenta-

mos una interpretación propia del esquema de realizabilidad que resulta más adecuada para obtener la prueba de corrección del compilador que se mostrará en el Capítulo 6.

5.1 Clausuras como realizadores de tipos

En la realizabilidad de Krivine, los términos del cálculo lambda son realizadores de los tipos del lenguaje. Aquí mostraremos una modificación propuesta por Jaber donde los realizadores son clausuras (en lugar de los términos), lo que permite trabajar con términos abiertos. En esta sección usaremos la máquina abstracta de la Definición 26, que tiene únicamente las instrucciones necesarias para evaluar al cálculo lambda.

La idea básica es que queremos asociarle un tipo a una clausura de máquina, de acuerdo a si esa clausura se comporta o no como un valor de ese tipo. El comportamiento que se analiza es la secuencia de transiciones que puede realizar esa clausura cuando forma parte de una configuración. Para cada tipo θ se define un conjunto de *tests* $\mathcal{T}(\theta) \subseteq \mathit{Stack}$, que contiene las pilas con las cuales se formaran esas configuraciones a analizar. La clausura debe "satisfacer" cada uno de los tests en $\mathcal{T}(\theta)$ para ser considerada un realizador de tipo θ.

Para formalizar esta idea, tomamos un conjunto $\perp\!\!\!\perp$ de configuraciones cerrado por anti-ejecución. Esto es, si $w \in \perp\!\!\!\perp$ y $w' \longmapsto w$ entonces se cumple $w' \in \perp\!\!\!\perp$. A $\perp\!\!\!\perp \subseteq \mathit{Conf}$ se lo denomina conjunto de *observaciones*. Una clausura α es un realizador de tipo θ, que escribimos $\alpha \in \mathcal{R}(\theta)$, si al combinarla con cada uno de los tests $s \in \mathcal{T}(\theta)$ se cumple $(\alpha,\ s) \in \perp\!\!\!\perp$. Esto se puede expresar como sigue:

Definición 58 (Realizadores de tipos).

$$\mathcal{R}(\theta) = \{\ \alpha \mid \textit{para todo } s \in \mathcal{T}(\theta) \textit{ se cumple } (\alpha,\ s) \in \perp\!\!\!\perp\ \}$$

Cuando $s \in \mathcal{T}(\theta)$, se dice que α *satisface* el test s si $(\alpha,\ s) \in \perp\!\!\!\perp$. El conjunto $\mathcal{T}(\theta)$ se define por inducción en el tipo θ. La definición de los tests para el tipo básico b va depender de la elección concreta del tipo, y de las constantes que posea el cálculo. Más adelante, en este capítulo, consideraremos el tipo **unit** como el tipo básico. Para los tipos funcionales, el conjunto de test de define de la siguiente manera:

Definición 59 (Tests para tipos funcionales).

$$\mathcal{T}(\theta \to \theta') = \{\ \alpha :: s \mid \alpha \in \mathcal{R}(\theta),\ s \in \mathcal{T}(\theta')\ \}$$

La idea intuitiva de esta definición es la siguiente: si una clausura realiza un tipo funcional $\theta \to \theta'$ debe "aceptar como argumento" un realizador de tipo θ y "producir como resultado" un realizador de tipo θ' (que por definición satisface todos los tests de tipo θ'). En particular, la clausura $(\texttt{Grab} \triangleright i,\ \eta)$ es un realizador de tipo $\theta \to \theta'$ si $(i,\ \alpha :: \eta)$ es un realizador de tipo θ' para cualquier clausura $\alpha \in \mathcal{R}(\theta)$.

Lema 17. $(\texttt{Grab} \triangleright i,\ \eta) \in \mathcal{R}(\theta \to \theta')$ *si para todo* $\alpha \in \mathcal{R}(\theta)$, $(i,\ \alpha :: \eta) \in \mathcal{R}(\theta')$.

Prueba. Para ver que $(\texttt{Grab} \triangleright i,\ \eta) \in \mathcal{R}(\theta \to \theta')$, por definición de $\mathcal{R}(\theta \to \theta')$ debemos tomar una pila $\alpha :: s \in \mathcal{T}(\theta \to \theta')$ (donde $\alpha \in \mathcal{R}(\theta)$ y $s \in \mathcal{T}(\theta')$) y demostrar $(\texttt{Grab} \triangleright i,\ \eta,\ \alpha :: s) \in \perp\!\!\!\perp$.

Por hipótesis sabemos que vale $(i,\ \alpha :: \eta) \in \mathcal{R}(\theta')$, y como $s \in \mathcal{T}(\theta')$ obtenemos $(i,\ \alpha :: \eta,\ s) \in \perp\!\!\!\perp$. Como $\perp\!\!\!\perp$ es cerrado por anti-ejecución, y además $(\texttt{Grab} \triangleright i,\ \eta,\ \alpha :: s) \longmapsto (i,\ \alpha :: \eta,\ s)$, se tiene $(\texttt{Grab} \triangleright i,\ \eta,\ \alpha :: s) \in \perp\!\!\!\perp$ como queríamos demostrar. □

Así como las clausuras son realizadores de tipos, podemos ver a los entornos como realizadores de contextos, simplemente extendiendo punto a punto la definición de realizador de tipos, como se indica a continuación.

Definición 60 (Realizadores de contextos).

$$\eta \in \mathcal{R}(\pi) \ \text{ si y sólo si } \ \eta \cdot n \in \mathcal{R}(\pi \cdot n) \ \text{ para todo } n < |\pi| \ .$$

El siguiente lema establece que, cuando un término tiene tipo θ, se puede obtener un realizador de tipo θ mediante la compilación de ese término. La función de compilación es la misma que en la Definición 28. Repetimos las reglas de tipado del cálculo lambda (Definición 19) para facilitar la lectura de la demostración.

Definición 61 (Reglas de tipado).

$$(\textsc{Abs})\ \frac{\theta :: \pi \vdash t : \theta'}{\pi \vdash \lambda t : \theta \to \theta'} \qquad (\textsc{Var})\ \frac{}{\pi \vdash \overline{n} : \theta}\ \pi \cdot n = \theta$$

$$(\textsc{App})\ \frac{\pi \vdash t : \theta \to \theta' \qquad \pi \vdash t' : \theta}{\pi \vdash t\, t' : \theta'}$$

Lema 18. *Si* $\eta \in \mathcal{R}(\pi)$ *y* $\pi \vdash t : \theta$ *entonces* $(\!|\, t \,|\!),\ \eta) \in \mathcal{R}(\theta)$.

Prueba. La prueba se realiza por inducción estructural en la derivación del juicio $\pi \vdash t : \theta$. Construiremos la prueba por casos:

- Caso (Abs). Sea $\alpha \in \mathcal{R}(\theta)$. Como $\eta \in \mathcal{R}(\pi)$ se tiene $\alpha :: \eta \in \mathcal{R}(\theta :: \pi)$. Luego por hipótesis inductiva obtenemos $(\!| t |\!),\ \alpha :: \eta) \in \mathcal{R}(\theta')$. Como esto vale para cualquier $\alpha \in \mathcal{R}(\theta)$, podemos usar el Lema 17 para obtener $(\mathtt{Grab} \triangleright (\!| t |\!),\ \eta) = ((\!| \lambda\, t |\!),\ \eta) \in \mathcal{R}(\theta \to \theta')$.

- Caso (Var). Debemos demostrar $((\!| \bar{n} |\!),\ \eta) = (\mathtt{Access}\ n,\ \eta) \in \mathcal{R}(\theta)$. Sea $s \in \mathcal{T}(\theta)$, debemos ver $(\mathtt{Access}\ n,\ \eta,\ s) \in \perp\!\!\!\perp$. Como tenemos $\eta \in \mathcal{R}(\pi)$, y $\pi \cdot n = \theta$, sabemos que $\eta \cdot n \in \mathcal{R}(\theta)$. Por lo tanto $(\eta \cdot n,\ s) \in \perp\!\!\!\perp$. Como $\perp\!\!\!\perp$ es cerrado por ejecución se sigue que $(\mathtt{Access}\ n,\ \eta,\ s) \in \perp\!\!\!\perp$.

- Caso (App). Debemos ver $((\!| t\ t' |\!),\ \eta) = (\mathtt{Push}\ (\!| t' |\!) \triangleright (\!| t |\!),\ \eta) \in \mathcal{R}(\theta')$. Sea $s \in \mathcal{T}(\theta')$, debemos ver $(\mathtt{Push}\ (\!| t' |\!) \triangleright (\!| t |\!),\ \eta,\ s) \in \perp\!\!\!\perp$. Por anti-ejecución, alcanza con ver $((\!| t |\!),\ \eta,\ ((\!| t' |\!),\ \eta) :: s) \in \perp\!\!\!\perp$. Por hipótesis inductiva sabemos $((\!| t' |\!),\ \eta) \in \mathcal{R}(\theta)$, luego $((\!| t' |\!),\ \eta) :: s \in \mathcal{T}(\theta \to \theta')$. Tenemos, por hipótesis inductiva, que $((\!| t |\!),\ \eta) \in \mathcal{R}(\theta \to \theta')$, por lo tanto podemos concluir $((\!| t |\!),\ \eta,\ ((\!| t' |\!),\ \eta) :: s) \in \perp\!\!\!\perp$ como queríamos.

□

El Lema 18 está muy relacionado con la prueba de corrección del compilador, como veremos en el Capítulo 6. La idea básica es que la clausura $(\!| t |\!),\ \eta)$, en lugar de ser un realizador del tipo θ, será un realizador de un valor denotacional $d \in [\![\theta]\!]$ que describe la semántica del término.

Como indica la Definición 58, los realizadores de un tipo θ son las clausuras α tales que *para toda* pila $s \in \mathcal{T}(\theta)$, se cumple $(\alpha,\ s) \in \perp\!\!\!\perp$. Es posible abstraer esta cuantificación sobre el conjunto $\mathcal{T}(\theta)$ utilizando un operador $(\_)^\top : \mathcal{P}(\mathit{Stack}) \to \mathcal{P}(\mathit{MClos})$, como se muestra a continuación:

$$\mathcal{R}(\theta) = \mathcal{T}(\theta)^\top$$
$$X^\top = \{\, \alpha \mid \text{para todo } s \in X,\ (\alpha,\ s) \in \perp\!\!\!\perp \,\} \ .$$

Aquí usamos $\mathcal{P}(A)$ para denotar las partes de A, es decir, el conjunto de todos los subconjuntos de A. Las propiedades del operador $(\_)^\top$ se pueden analizar de manera general, como lo haremos en la siguiente sección.

5.2 Biortogonalidad

Biortogonalidad es un concepto muy general que se ha utilizado en distintas áreas de las ciencias de la computación tales como equivalencia de programas [94], realizabilidad [70] y corrección de compiladores [15, 62], entre otras áreas [25, 96]. La idea fundamental puede explicarse como sigue.

Sean E y T dos conjuntos, y $\models \subseteq E \times T$ una relación entre esos conjuntos. Si pensamos a T como un conjunto de *tests*, y a $\models$ como una relación de satisfacibilidad, entonces $\epsilon \models \tau$ establece que $\epsilon \in E$ *satisface* el test $\tau \in T$. Si T_0 es un subconjunto de T, entonces denotamos $T_0^{\top}$ al conjunto de elementos que satisfacen todos los tests en T_0.

$$T_0^{\top} = \{\epsilon \in E \mid \text{para todo } \tau \in T_0,\ \epsilon \models \tau\} \ .$$

Como un ejemplo concreto, si T son la fórmulas y E son los modelos de una lógica particular, entonces $T_0^{\top}$ es el conjunto de modelos que satisfacen todas las fórmulas en T_0. Podemos definir una operación dual para obtener el conjunto de tests que son satisfechos por todos los elementos de un subconjunto $E_0 \subseteq E$:

$$E_0^{\perp} = \{\tau \in T \mid \text{para todo } \epsilon \in E_0,\ \epsilon \models \tau\} \ .$$

Las funciones $(\_)^{\perp} : \mathcal{P}(E) \to \mathcal{P}(T)$ y $(\_)^{\top} : \mathcal{P}(T) \to \mathcal{P}(E)$ se denominan operadores *ortogonales*. Para examinar las propiedades de estos operadores se puede analizar su acción en los posets $\langle \mathcal{P}(E), \subseteq \rangle$ y $\langle \mathcal{P}(T), \subseteq \rangle$. Utilizamos la notación $\langle P, \leq_P \rangle$ para denotar al poset formado por el conjunto P y el orden parcial $\leq_P$. En particular, el par de operadores ortogonales es una conexión de Galois antítona [26, 89], cuya definición mostramos a continuación.

Definición 62 (Conexión de Galois antítona). *Sean $\langle P, \leq_P \rangle$ y $\langle Q, \leq_Q \rangle$ posets. Supongamos $f : P \to Q$ y $g : Q \to P$ son un par de funciones tal que para todo $p \in P$ y para todo $q \in Q$ se cumple,*

$$q \leq_Q f(p) \ \text{ si y sólo si } \ p \leq_P g(q) \ . \tag{Gal}$$

Entonces el par (f, g) es una conexión de Galois entre los posets $\langle P, \leq_P \rangle$ y $\langle Q, \leq_Q \rangle$.

Lema 19. *El par de operadores $((\_)^{\perp}, (\_)^{\top})$ es una conexión de Galois entre los posets $\langle \mathcal{P}(E), \subseteq \rangle$ y $\langle \mathcal{P}(T), \subseteq \rangle$.*

Prueba. Debemos tomar $E_0 \subseteq E$ y $T_0 \subseteq T$ y demostrar

$$T_0 \subseteq E_0^{\perp} \ \text{ si y sólo si } \ E_0 \subseteq T_0^{\top} \ .$$

Supongamos $T_0 \subseteq E_0^{\perp}$ y sea $\epsilon \in E_0$. Queremos ver que $\epsilon \in T_0^{\top}$, esto es, mostrar que $\epsilon \models \tau$ para todo $\tau \in T_0$. Dado $\tau \in T_0$, tenemos que $\tau \in E_0^{\perp}$, y como $\epsilon \in E_0$ se tiene $\epsilon \models \tau$ por definición de $(\_)^{\perp}$. La otra implicación se demuestra de manera simétrica. □

Notar que la Definición 62 es simétrica y por lo tanto los roles de f y g son intercambiables. Es decir, si (f, g) es una conexión de Galois entre $\langle P, \leq_P \rangle$ y $\langle Q, \leq_Q \rangle$, entonces (g, f) es una conexión de Galois entre $\langle Q, \leq_Q \rangle$ y $\langle P, \leq_P \rangle$. Continuamos con la siguiente propiedad de las conexiones de Galois:

Lema 20. *Si (f, g) es una conexión de Galois entre $\langle P, \leq_P \rangle$ y $\langle Q, \leq_Q \rangle$ entonces:*

1. *para todo $p \in P$, $p \leq_P g(f(p))$ (análogamente, $q \leq_Q f(g(q))$ para todo $q \in Q$),*
2. *f es antítona: si $p \leq_P p'$ entonces $f(p') \leq_Q f(p)$ (análogamente, g es antítona),*
3. *$f \circ g \circ f = f$ (análogamente, $g \circ f \circ g = g$).*

Prueba. (1) Sea $p \in P$, entonces $f(p) \leq_Q f(p)$ por reflexividad. Por lo tanto obtenemos $p \leq_P g(f(p))$ aplicando (Gal) de la Definición 62.

(2) Como $p \leq_P p'$ se tiene, por transitividad, que $p \leq_P g(f(p'))$. Luego por (Gal) se obtiene $f(p') \leq_Q f(p)$.

(3) Sea $p \in P$. Queremos ver $f(g(f(p))) = f(p)$; usando antisimetría de $\leq_Q$ demostraremos la desigualdad para ambos lados. Como $p \leq_P g(f(p))$ y f es antítona, tenemos $f(g(f(p))) \leq_Q f(p)$. La otra desigualdad es una aplicación de (1) con $q = f(p)$, es decir, $f(p) \leq_Q f(g(f(p)))$. □

Cuando se componen los dos operadores ortogonales se obtiene una función $(\_)^{\perp\top} : \mathcal{P}(E) \to \mathcal{P}(E)$ que forma un *operador de clausura* del poset $\langle \mathcal{P}(E), \subseteq \rangle$. Mostraremos que una conexión de Galois entre $\langle P, \leq_P \rangle$ y $\langle Q, \leq_Q \rangle$ da lugar a un operador de clausura del poset $\langle P, \leq_P \rangle$.

Definición 63 (Operador de clausura). *Un operador de clausura del poset $\langle P, \leq_P \rangle$ es una función $h : P \to P$ tal que, para todo $p, p' \in P$,*

- *$p \leq_P h(p)$ (h es extensiva).*
- *si $p \leq_P p'$, entonces $h(p) \leq_P h(p')$ (h es monótona).*
- *$h(h(p)) = h(p)$ (h es idempotente).*

Lema 21. *Supongamos (f, g) es una conexión de Galois entre los posets $\langle P, \leq_P \rangle$ y $\langle Q, \leq_Q \rangle$, entonces $g \circ f$ es un operador de clausura para $\langle P, \leq_P \rangle$.*

Prueba. Sea $h = g \circ f$, entonces:

- h es extensiva: Directo por Lema 20 (1).

- h es monótona: Sea $p \leq_P p'$. Por Lema 20 (2) sabemos que tanto f como g son antítonas. Luego $f(p') \leq_Q f(p)$, y por lo tanto tenemos

$$h(p) = g(f(p)) \leq_P g(f(p')) = h(p') \ .$$

- h es idempotente: Tenemos $h = g \circ f = g \circ (f \circ g \circ f) = h \circ h$, aplicando el Lema 20 (3).

□

Con estos resultados podemos concluir que $(\_)^{\perp\top} : \mathcal{P}(E) \to \mathcal{P}(E)$ es un operador de clausura para el poset $\langle \mathcal{P}(E), \subseteq \rangle$, y que tanto $(\_)^{\perp}$ y $(\_)^{\top}$ son funciones antítonas respecto a la inclusión.

La utilidad principal de la biortogonalidad es que para un conjunto E_0 podemos obtener un conjunto $E_0^{\perp\top}$ que es una extensión de E_0 y satisface todos los tests en $E_0^{\perp}$. Esto es, el operador de clausura permite extender E_0 sin "perder" ningún test y por lo tanto mantener la relación de satisfacibilidad. En la siguiente sección, presentamos un uso concreto de biortogonalidad que será útil para demostrar la corrección de un compilador.

5.3 Un enfoque alternativo

Aquí presentamos una modificación del enfoque de la Sección 5.1 donde aplicamos el concepto de realizabilidad en la máquina de Krivine. Esta modificación resulta más adecuada para obtener la demostración de la corrección del compilador que veremos en el siguiente capítulo, debido fundamentalmente a que el uso de los operadores ortogonales otorga mayor flexibilidad en el diseño de la máquina abstracta. Presentaremos la nueva definición de realizadores y tests, utilizando un lenguaje y una máquina abstracta ligeramente diferentes a las usadas en la Sección 5.1.

El lenguaje que usaremos es el cálculo lambda con una constante $\diamond$ (denominada *unit*). El sistema de tipos contiene el tipo **unit** y los tipos funcionales.

Definición 64 (Términos).

$$t,\ t' \in \mathit{Term} ::= \lambda\, t \mid \overline{n} \mid t\ t' \mid \diamond \quad \textit{(Términos)}$$
$$\theta,\ \theta' \in \mathit{Type} ::= \mathbf{unit} \mid \theta \to \theta' \quad \textit{(Tipos)}$$

Las reglas de tipado son las mismas de la Definición 61, con el agregado de una regla nueva para la constante $\diamond$, que tiene tipo **unit** bajo cualquier contexto.

Definición 65 (Regla de tipado de la constante $\diamond$).

$$(\textsc{Unit})\ \frac{}{\pi \vdash \diamond : \boldsymbol{unit}}$$

Utilizaremos la misma máquina abstracta que en la Definición 26, excepto que agregaremos una nueva instrucción para implementar la constante $\diamond$.

Definición 66 (Código).

$$\begin{array}{rcl} i \in \mathit{Code} & ::= & \mathtt{Access}\ n \\ & | & \mathtt{Grab} \triangleright i \\ & | & \mathtt{Push}\ i' \triangleright i \\ & | & \mathtt{Unit} \end{array}$$

Las transiciones serán las mismas que en la Definición 27. La instrucción `Unit` no tendrá regla de transición, puesto que necesitamos que si la ejecución llega una configuración de la forma $(\mathtt{Unit},\ \eta,\ s)$, entonces la máquina se detenga en ese punto (indicando que la clausura es un realizador de tipo **unit**, como veremos en esta sección).

Definimos ahora los operadores ortogonales que utilizaremos para definir los realizadores. La relación de satisfacibilidad (en el sentido de la sección anterior) se define en términos de un conjunto $\perp\!\!\!\perp \subseteq \mathit{Conf}$ cerrado por anti-ejecución.

Definición 67 (Operadores ortogonales).

$$X^{\perp} = \{\ \alpha \mid \textit{para todo } s \in X \textit{ se cumple } (\alpha,\ s) \in \perp\!\!\!\perp\ \}$$
$$Y^{\top} = \{\ s \mid \textit{para todo } \alpha \in Y \textit{ se cumple } (\alpha,\ s) \in \perp\!\!\!\perp\ \}$$

Continuamos con la definición del conjunto de realizadores de tipo θ, que se denota $\mathcal{R}(\theta)$. Se define también un conjunto $\mathcal{R}_P(\theta)$ de clausuras llamadas *realizadores primitivos* de tipo θ, que son aquellas clausuras que por representar un valor (o forma canónica) del tipo θ se los considera naturalmente realizadores del mismo tipo. Por ejemplo, una clausura $(\mathtt{Unit},\ \eta)$ se considera un realizador primitivo de tipo **unit** pues representa al valor $\diamond$.

Definición 68 (Realizadores).

$$\begin{aligned}\mathcal{R}_P(\textbf{\textit{unit}}) &= \{\,(\texttt{Unit},\ \eta)\ \mid\ \eta \in \mathit{MEnv}\,\}\\ \mathcal{R}_P(\theta \to \theta') &= \{(\texttt{Grab} \triangleright i,\ \eta)\ \mid\ \textit{para todo } \alpha \in \mathcal{R}(\theta)\\ &\qquad\qquad \textit{se cumple } (i,\ \alpha :: \eta) \in \mathcal{R}(\theta')\}\\ \mathcal{R}(\theta) &= \mathcal{R}_P(\theta)^{\perp\top}\end{aligned}$$

Los realizadores primitivos de tipo $\theta \to \theta'$ son aquellas clausuras de la forma $(\texttt{Grab} \triangleright i,\ \eta)$ tales que $(i,\ \alpha :: \eta)$ es un realizador de tipo θ' para todo realizador α de tipo θ. Esto es coherente con la intuición de que los realizadores de tipos funcionales $\theta \to \theta'$ deben ser capaces de "aceptar como argumento" realizadores de tipo θ y "producir como resultado" realizadores de tipo θ'. El hecho de que la clausura α sea un realizador que no necesariamente es primitivo refleja el orden de evaluación normal que posee la máquina abstracta. Es decir, si exigiéramos $\alpha \in \mathcal{R}_P(\theta)$ estaríamos forzando que α represente un valor de tipo θ; en cambio, $\alpha \in \mathcal{R}(\theta)$ indica que la clausura realiza el tipo θ pero no necesariamente se ha computado aún el valor que representa.

El conjunto $\mathcal{T}(\theta)$ contiene aquellas pilas que combinadas con todos los realizadores de tipo θ (formando una configuración) pertenecen al conjunto de observaciones $\perp\!\!\!\perp$. Esto se puede expresar como se indica a continuación usando el operador ortogonal $(\_)^{\perp} : \mathcal{P}(\mathit{MClos}) \to \mathcal{P}(\mathit{Stack})$.

Definición 69 (Tests).

$$\mathcal{T}(\theta) = \mathcal{R}(\theta)^{\perp}$$

De la extensividad del operador $(\_)^{\perp\top}$ de desprende que $\mathcal{R}_P(\theta) \subseteq \mathcal{R}(\theta)$. Más aún, el conjunto $\mathcal{R}(\theta)$ es una extensión de $\mathcal{R}_P(\theta)$ que satisface todos los tests en $\mathcal{R}_P(\theta)^{\perp}$. En efecto, se puede demostrar el siguiente resultado.

Lema 22. $\mathcal{T}(\theta) = \mathcal{R}_P(\theta)^{\perp}$.

Prueba. Por definición se tiene $\mathcal{T}(\theta) = \mathcal{R}(\theta)^{\perp} = (\mathcal{R}_P(\theta)^{\perp\top})^{\perp}$. Por lo tanto obtenemos $\mathcal{T}(\theta) = h(\mathcal{R}_P(\theta))$ con $h = (\_)^{\perp} \circ (\_)^{\top} \circ (\_)^{\perp}$. Por Lema 20 (3) sabemos $h = (\_)^{\perp}$ como queríamos demostrar. □

Notemos que por lema anterior se tiene $\mathcal{R}(\theta) = \mathcal{R}_P(\theta)^{\perp\top} = \mathcal{T}(\theta)^{\top}$. En efecto, para demostrar que $\alpha \in \mathcal{R}(\theta)$ usualmente se aplica la definición del operador $(\_)^{\top}$, es decir, se toma una pila $s \in \mathcal{T}(\theta)$ cualquiera y se demuestra que $(\alpha,\ s) \in \perp\!\!\!\perp$. Esa estrategia de prueba no tiene pérdida de generalidad ya que, si no existe $s \in \mathcal{T}(\theta)$, entonces $\mathcal{T}(\theta) = \emptyset$ y $\mathcal{R}(\theta) = \mathcal{T}(\theta)^{\top} = \emptyset^{\top} = \mathit{MClos}$, con lo cual la afirmación $\alpha \in \mathcal{R}(\theta)$ es trivialmente verdadera.

Como sucedía en la Sección 5.1, los tests de tipos funcionales $\theta \to \theta'$ se obtienen combinando un realizador α de tipo θ con un test s de tipo θ', formando la pila $\alpha :: s$.

Lema 23. *Si $\alpha \in \mathcal{R}(\theta)$ y $s \in \mathcal{T}(\theta')$, entonces $\alpha :: s \in \mathcal{T}(\theta \to \theta')$.*

Prueba. Para demostrar $\alpha :: s \in \mathcal{T}(\theta \to \theta')$, podemos aplicar Lema 22 y tomar $\alpha' \in \mathcal{R}_p(\theta \to \theta')$ y luego probar $(\alpha', \alpha :: s) \in \perp\!\!\!\perp$. Por definición de $\mathcal{R}_p(\theta \to \theta')$, la clausura α' tiene la forma $(\mathtt{Grab} \triangleright i, \eta)$ para algún $i \in \mathit{Code}$ y $\eta \in \mathit{MEnv}$. Más aún, como $\alpha \in \mathcal{R}(\theta)$ se tiene $(i, \alpha :: \eta) \in \mathcal{R}(\theta')$. Luego, como $s \in \mathcal{T}(\theta')$, se tiene $(i, \alpha :: \eta, s) \in \perp\!\!\!\perp$. Se obtiene entonces por anti-ejecución que $(\alpha', \alpha :: s) \in \perp\!\!\!\perp$. □

Un entorno η puede verse como realizador de un contexto π (es decir, $\eta \in \mathcal{R}(\pi)$) si en cada posición n del entorno, se tiene $\eta \cdot n \in \mathcal{R}(\pi \cdot n)$. Alternativamente, se puede definir $\mathcal{R}(\pi)$ con las siguientes reglas inductivas:

Definición 70 (Realizadores de contextos).

$$\frac{}{[] \in \mathcal{R}([])} \qquad \frac{\alpha \in \mathcal{R}(\theta) \qquad \eta \in \mathcal{R}(\pi)}{\alpha :: \eta \in \mathcal{R}(\theta :: \pi)}$$

El siguiente resultado (denominado *adecuación*, análogo al Lema 18) relaciona realizabilidad con el sistema de tipos del lenguaje.

Lema 24. *Si $\eta \in \mathcal{R}(\pi)$ y $\pi \vdash t : \theta$, entonces $(\!| t |\!), \eta) \in \mathcal{R}(\theta)$.*

Prueba. Se demuestra por inducción en la derivación del juicio $\pi \vdash t : \theta$. Procedemos caso por caso según la última regla de tipado aplicada en la derivación.

- Caso (Abs). Queremos ver que $((\!| \lambda t |\!), \eta) \in \mathcal{R}(\theta \to \theta')$. Probaremos en su lugar que $((\!| \lambda t |\!), \eta) = (\mathtt{Grab} \triangleright (\!| t |\!), \eta) \in \mathcal{R}_p(\theta \to \theta')$, y para ello aplicaremos la definición de $\mathcal{R}_p(\theta \to \theta')$. Sea $\alpha \in \mathcal{R}(\theta)$, como $\eta \in \mathcal{R}(\pi)$ se tiene que $\alpha :: \eta \in \mathcal{R}(\theta :: \pi)$. Luego aplicando hipótesis inductiva se obtiene $((\!| t |\!), \alpha :: \eta) \in \mathcal{R}(\theta')$. Como esto vale para cualquier $\alpha \in \mathcal{R}(\theta)$, concluimos por definición de $\mathcal{R}_p(\theta \to \theta')$ que $((\!| \lambda t |\!), \eta) \in \mathcal{R}_p(\theta \to \theta')$.

- Caso (Var). Debemos ver $((\!| \bar{n} |\!), \eta) = (\mathtt{Access}\ n, \eta) \in \mathcal{R}(\theta)$. Como $\mathcal{R}(\theta) = \mathcal{T}(\theta)^\top$ aplicaremos la definición del operador ortogonal $(\_)^\top$. Sea $s \in \mathcal{T}(\theta)$, debemos ver $(\mathtt{Access}\ n, \eta, s) \in \perp\!\!\!\perp$. Como tenemos $\eta \in \mathcal{R}(\pi)$, y $\pi \cdot n = \theta$, sabemos que $\eta \cdot n \in \mathcal{R}(\theta)$. Por lo tanto $(\eta \cdot n, s) \in \perp\!\!\!\perp$. Como $\perp\!\!\!\perp$ es cerrado por ejecución se sigue que $(\mathtt{Access}\ n, \eta, s) \in \perp\!\!\!\perp$.

- Caso (App). Debemos ver $(\!|\, t\ t' \,|\!), \eta) = (\mathsf{Push}\ (\!|\, t' \,|\!) \triangleright (\!|\, t \,|\!), \eta) \in \mathcal{R}(\theta')$. Sea $s \in \mathcal{T}(\theta')$, veamos $(\mathsf{Push}\ (\!|\, t' \,|\!) \triangleright (\!|\, t \,|\!), \eta, s) \in \perp\!\!\!\perp$. Por anti-ejecución alcanza con ver $((\!|\, t \,|\!), \eta, \alpha :: s) \in \perp\!\!\!\perp$ donde $\alpha = ((\!|\, t' \,|\!), \eta)$. Por hipótesis inductiva se tiene $\alpha \in \mathcal{R}(\theta)$. Luego, por Lema 23, $\alpha :: s \in \mathcal{T}(\theta \to \theta')$. Dado que, por hipótesis inductiva, $((\!|\, t \,|\!), \eta) \in \mathcal{R}(\theta \to \theta')$ concluimos $((\!|\, t \,|\!), \eta, \alpha :: s) \in \perp\!\!\!\perp$.

- Caso (Unit). Este caso es trivial ya que $((\!|\, \diamond \,|\!), \eta) = (\mathtt{Unit}, \eta) \in \mathcal{R}_p(\mathbf{unit})$.

□

Se debe notar que el resultado anterior vale para cualquier elección del conjunto $\perp\!\!\!\perp$ de observaciones, cuyo único requerimiento es que sea cerrado por anti-ejecución. A partir de ese resultado, se puede demostrar lo que se denomina *corrección operacional* para términos cerrados de tipo **unit**:

Lema 25. *Si* $[] \vdash t : \mathbf{unit}$, *entonces* $((\!|\, t \,|\!), [], []) \longmapsto^* (\mathtt{Unit}, \eta, [])$, *para algún entorno* $\eta \in MEnv$.

Prueba. Definimos $\perp\!\!\!\perp = \{ w \mid w \longmapsto^* (\mathtt{Unit}, \eta, []),\ \text{para algún } \eta \in Env \}$. Es fácil ver que el conjunto $\perp\!\!\!\perp$ definido de esa manera es cerrado por anti-ejecución. Por lo tanto podemos aplicar el Lema 24 y obtener $((\!|\, t \,|\!), []) \in \mathcal{R}(\mathbf{unit})$.

Por otro lado, es claro que $[] \in \mathcal{T}(\mathbf{unit}) = \mathcal{R}_p(\mathbf{unit})^\top$. Concluimos entonces que $((\!|\, t \,|\!), [], []) \in \perp\!\!\!\perp$, que es lo que queríamos demostrar. □

Inclusión del punto fijo

Hasta aquí hemos analizado el concepto de realizabilidad aplicado sobre un lenguaje tipado. Sin embargo, el sistema de tipos suprime gran parte de la expresividad que posee el cálculo lambda puro, de hecho, todos los términos bien tipados pueden reducirse a una forma canónica (el lenguaje es fuertemente normalizable) y por lo tanto no hay posibilidad de escribir, por ejemplo, un término divergente (en el sentido del Capítulo 2). Cuando se requiere un lenguaje con mayor expresividad, resulta conveniente agregar un operador de punto fijo a los términos del mismo. En lo que resta de este capítulo analizaremos cómo aplicar realizabilidad en el cálculo lambda con este operador incorporado.

Definición 71 (Operador de punto fijo).

$$t \in \mathit{Term} ::= \ldots \mid \mathit{rec}\ t$$

La regla de tipado de este operador se puede expresar como sigue:

Definición 72 (Regla de tipado).

$$(\textsc{Rec})\ \frac{\pi \vdash t : \theta \to \theta}{\pi \vdash \mathit{rec}\ t : \theta}$$

Incorporamos una nueva instrucción a la máquina abstracta, que permitirá simular la recursión introducida por el operador de punto fijo.

Definición 73 (La instrucción `Fix` y su regla de transición).

$$i \in \mathit{Code} ::= \ldots \mid \texttt{Fix} \triangleright i$$
$$(\texttt{Fix} \triangleright i,\ \eta,\ s) \longmapsto (i,\ \eta,\ (\texttt{Fix} \triangleright i,\ \eta) :: s)\ .$$

En principio podríamos mantener intacta la definición de realizadores y tests, pero con esa definición no se puede completar la prueba del lema de adecuación, al menos cuando se intenta demostrarlo por inducción en la derivación del juicio de tipado.

Enunciado. *Si* $\eta \in \mathcal{R}(\pi)$ *y* $\pi \vdash t : \theta$*, entonces* $(\!|\, t \,|\!),\ \eta) \in \mathcal{R}(\theta)$.

Intento de demostración. Caso (Rec). Sea $\alpha = (\texttt{Fix} \triangleright (\!|\, t \,|\!),\ \eta)$, debemos ver que $\alpha \in \mathcal{R}(\theta)$. Tomemos $s \in \mathcal{T}(\theta)$ y veamos $(\alpha,\ s) \in \perp\!\!\!\perp$. Como $\perp\!\!\!\perp$ es cerrado por anti-ejecución alcanza con ver $((\!|\, t \,|\!),\ \eta,\ \alpha :: s) \in \perp\!\!\!\perp$.

Para ello, podemos probar $((\!|\, t \,|\!),\ \eta) \in \mathcal{R}(\theta \to \theta)$ y $\alpha :: s \in \mathcal{T}(\theta \to \theta)$. Lo primero es válido por hipótesis inductiva. Sin embargo, si se intenta aplicar el Lema 23 para demostrar $\alpha :: s \in \mathcal{T}(\theta \to \theta)$, nos encontramos con que el lema requiere la hipótesis $\alpha \in \mathcal{R}(\theta)$ que es justamente lo que queremos demostrar.

En lo que resta del capítulo veremos cómo adaptar la definición de los realizadores y los tests para poder obtener un lema similar al de adecuación en la presencia del operador de punto fijo.

5.4 Relaciones indexadas

Hemos observado que la incorporación del operador de punto fijo rompe el esquema de realizabilidad aplicado sobre la máquina de Krivine. No obstante, es posible aplicar un esquema similar utilizando relaciones indexadas (en inglés, *Step-indexing* relations) que se utilizan usualmente para enfrentar las dificultades que introduce en el razonamiento la incorporación de operadores recursivos. Este tipo de relaciones se han utilizado (a veces en combinación con biortogonalidad) para obtener pruebas de corrección de compiladores [16, 62],

equivalencia de programas [7, 51], proof-carrying code [11], entre otros tópicos [27, 25]. Mostraremos cómo se puede aplicar este método para adaptar el esquema de realizabilidad sobre el cálculo lambda tipado con operador de punto fijo.

En esta sección introducimos las relaciones indexadas y mostramos algunas propiedades que surgen al combinarlas con operadores ortogonales. Una relación indexada sobre un conjunto X es simplemente una secuencia infinita de subconjuntos de X.

Definición 74 (Relación indexada). *Una relación indexada sobre X es una secuencia infinita R_r de conjuntos tal que $R_r \subseteq X$ para todo $r \in \mathbb{N}$.*

Esa secuencia de subconjuntos es además *decreciente* si se cumple

$$R_0 \supseteq R_1 \supseteq R_2 \supseteq R_3 \supseteq \dots$$

o equivalentemente:

Definición 75 (Relación indexada decreciente). *Una relación indexada sobre el conjunto X es decreciente si para todo $r \in \mathbb{N}$, $R_{r+1} \subseteq R_r$.*

Un ejemplo de relación indexada decreciente sobre el conjunto de configuraciones de la máquina de Krivine se obtiene al fijar cada R_r como el conjunto de configuraciones a partir de las cuales se pueden realizar al menos r pasos de transición.

Retomemos el esquema general de biortogonalidad donde teníamos un conjunto de elementos E y un conjunto de tests T sobre los cuales se definieron operadores ortogonales. Dada una relación indexada $\Vdash_r$ sobre $E \times T$, definimos la relación binaria $\Vdash \subseteq \widehat{E} \times \widehat{T}$ sobre elementos indexados $\widehat{E} = \mathbb{N} \times E$ y tests indexados $\widehat{T} = \mathbb{N} \times T$ de la siguiente manera.

Definición 76 (Relación de satisfacibilidad). *Sea $\Vdash_r$ una relación indexada sobre $E \times T$, entonces $\Vdash \subseteq \widehat{E} \times \widehat{T}$ queda definida como sigue:*

$$(u,\ \epsilon)\ \Vdash\ (r,\ \tau) \qquad \textit{si y sólo si} \qquad \epsilon \Vdash_{min(u,\,r)} \tau\ .$$

Cuando se cumple $(u,\ \epsilon)\ \Vdash\ (r,\ \tau)$ y la relación indexada $\Vdash_r$ es decreciente, se tiene que $\epsilon \Vdash_{u'} \tau$ para todo u' tal que $u' \leq r$ y a la vez $u' \leq u$. Intuitivamente, los índices u y r definen una secuencia finita de relaciones $\Vdash_0, \dots, \Vdash_{min(u,\,r)}$ donde el elemento ϵ satisface el test τ.

Definimos el operador ortogonal $(\_)^{\perp} : \mathcal{P}(\hat{E}) \to \mathcal{P}(\hat{T})$ donde la relación de satisfacibilidad se define en términos de $\Vdash \subseteq \hat{E} \times \hat{T}$:

$$X^{\perp} = \{ (r,\ \tau) \mid \text{para todo } (u,\ \epsilon) \in X, \text{ se cumple } (u,\ \epsilon) \Vdash (r,\ \tau) \} \ .$$

Por definición, para demostrar que $(r,\ \tau) \in X^{\perp}$, se debe verificar que *todo* elemento $(u,\ \epsilon)$ en el conjunto X está relacionado con $(r,\ \tau)$ vía $\Vdash$. Sin embargo, cuando $\Vdash_r$ es indexada decreciente, se puede dar una definición más simple de $X^{\perp}$ que permita verificar solamente un subconjunto de los elementos en X. Para llegar a esa definición necesitamos analizar cómo se comporta el operador $(\_)^{\perp}$ para *conjuntos completos.*

Definición 77 (Conjunto completo). *Para cualquier conjunto E, decimos que $X \subseteq \hat{E}$ es completo cuando $(u,\ \epsilon) \in X$ y $u' \leq u$ implica $(u',\ \epsilon) \in X$.*

Es posible demostrar que el conjunto $X^{\perp}$ es completo cuando $\Vdash_r$ es decreciente:

Lema 26. *Supongamos $\Vdash_r$ es decreciente. Entonces, para todo $X \subseteq \hat{E}$, $X^{\perp}$ es completo. Análogamente, para todo $Y \subseteq \hat{T}$, $Y^{\top}$ es completo.*

Prueba. Sea $(r,\ \tau) \in X^{\perp}$ y sea $r' \leq r$. Veamos $(r',\ \tau) \in X^{\perp}$. Para ello tomemos $(u,\ \epsilon) \in X$. Como $(r,\ \tau) \in X^{\perp}$, se tiene $\epsilon \Vdash_{min(u,\ r)} \tau$. Además, como $r' \leq r$, es fácil ver que $min(u,\ r') \leq min(u,\ r)$. Dado que $\Vdash_r$ es decreciente, se tiene que la relación $\Vdash_{min(u,\ r)}$ está incluida en $\Vdash_{min(u,\ r')}$. Luego concluimos $\epsilon \Vdash_{min(u,\ r')} \tau$. □

En particular, el Lema 26 establece que cuando $\Vdash_r$ es decreciente, se tiene que $X^{\perp\top}$ es siempre completo.

La importancia de los conjuntos completos radica en que facilitan la verificación de qué elementos están en el conjunto ortogonal. Si X es completo y se quiere demostrar $(r,\ \tau) \in X^{\perp}$, sólo se deben verificar aquellos elementos $(u,\ \epsilon) \in X$ tales que $u \leq r$, como muestra el siguiente lema.

Lema 27. *Si $X \subseteq \hat{E}$ es completo, entonces*

$$X^{\perp} = \{ (r,\ \tau) \mid \textit{para todo } (u,\ \epsilon) \in X,\ u \leq r \textit{ implica } \epsilon \Vdash_u \tau \} \ .$$

Análogamente, si $Y \subseteq \hat{T}$, entonces

$$Y^{\top} = \{ (u,\ \epsilon) \mid \textit{para todo } (r,\ \tau) \in Y,\ r \leq u \textit{ implica } \epsilon \Vdash_u \tau \} \ .$$

Prueba. Demostraremos $X^{\perp} \subseteq \overline{X}$, y $X^{\perp} \supseteq \overline{X}$ donde $\overline{X}$ se define como el conjunto $\{ (r,\ \tau) \mid \text{para todo } (u,\ \epsilon) \in X,\ u \leq r \text{ implica } \epsilon \Vdash_u \tau \}$.

- $X^{\perp} \subseteq \overline{X}$. Sea $(r, \tau) \in X^{\perp}$, veamos $(r, \tau) \in \overline{X}$. Tomemos $(u, \epsilon) \in X$ con $u \leq r$. Por definición de $(\_)^{\perp}$ se tiene $\epsilon \Vdash_{min(u, r)} \tau$. Como $min(u, r) = u$, se tiene $\epsilon \Vdash_{u} \tau$.
- $X^{\perp} \supseteq \overline{X}$. Sea $(r, \tau) \in \overline{X}$, veamos $(r, \tau) \in X^{\perp}$. Sea $(u, \epsilon) \in X$. Si $u \leq r$, entonces se tiene $\epsilon \Vdash_{u} \tau$, y como $min(u, r) = u$ obtenemos $\epsilon \Vdash_{min(u, r)} \tau$. Supongamos ahora $u > r$. Como X es completo, $(r, \epsilon) \in X$. Luego, por definición de $\overline{X}$, se tiene $\epsilon \Vdash_{r} \tau$. Por lo tanto, como $min(u, r) = r$, se obtiene $\epsilon \Vdash_{min(u, r)} \tau$.

□

5.5 Realizabilidad usando indexación

Usando relaciones indexadas se puede definir un esquema de realizabilidad compatible con el cálculo lambda tipado y un operador de punto fijo. De la misma manera como en la Sección 5.3 trabajamos con un conjunto $\perp\!\!\!\perp$ de configuraciones, ahora tomaremos $\perp\!\!\!\perp_r$ como una relación indexada decreciente sobre *Conf* con las siguientes propiedades:

Definición 78 (Observaciones). *$\perp\!\!\!\perp_r$ es una relación indexada decreciente sobre Conf tal que:*

1. $\perp\!\!\!\perp_0 = Conf$ *(conjunto total de configuraciones)*
2. $w' \in \perp\!\!\!\perp_r$ *y* $w \longmapsto w'$ *implica* $w \in \perp\!\!\!\perp_{r+1}$.

Es fácil ver que con esas propiedades cada conjunto $\perp\!\!\!\perp_r$ es cerrado por anti-ejecución. Un ejemplo de relación indexada decreciente que cumple (1) y (2) es el siguiente:

$$\perp\!\!\!\perp_r = \{ (\alpha, s) \mid \text{se pueden realizar al menos } r \text{ transiciones a partir de } (\alpha, s) \} \ .$$

Aprovechando que los operadores ortogonales se aplicarán siempre a conjuntos completos, podemos usar el Lema 27 para definirlos de la siguiente manera:

Definición 79 (Operadores ortogonales).

$$X^{\perp} = \{ (r, s) \mid \textit{para todo } (u, \alpha) \in X, u \leq r \textit{ implica } (\alpha, s) \in \perp\!\!\!\perp_u \}$$
$$Y^{\top} = \{ (u, \alpha) \mid \textit{para todo } (r, s) \in Y, r \leq u \textit{ implica } (\alpha, s) \in \perp\!\!\!\perp_r \}$$

Como antes, el conjunto de realizadores $\mathcal{R}(\theta)$ se define a partir del conjunto de realizadores primitivos $\mathcal{R}_P(\theta)$.

Definición 80 (Realizadores).

$$\begin{array}{rcl} \mathcal{R}_P(\textbf{\textit{unit}}) & = & \{\,(u,\,(\texttt{Unit},\,\eta))\;\mid\; u\in\mathbb{N},\,\eta\in \mathit{MEnv}\,\} \\ \mathcal{R}_P(\theta\to\theta') & = & \{\,(u,\,(\texttt{Grab}\triangleright i,\,\eta))\;\mid\; \textit{para todo } u'\le u, \\ & & (u',\,\alpha)\in\mathcal{R}(\theta),\ \textit{se cumple}\ (u',\,(i,\,\alpha::\eta))\in\mathcal{R}(\theta')\,\} \\ \mathcal{R}(\theta) & = & \mathcal{R}_P(\theta)^{\perp\,\top} \end{array}$$

El conjunto de tests $\mathcal{T}(\theta)\subseteq\mathbb{N}\times \mathit{Stack}$ se define en términos del operador ortogonal $(\_)^{\perp}:\mathcal{P}(\mathbb{N}\times \mathit{MClos})\to\mathcal{P}(\mathbb{N}\times \mathit{Stack})$.

Definición 81 (Tests).

$$\mathcal{T}(\theta)=\mathcal{R}(\theta)^{\perp}$$

Es fácil ver que el Lema 22 sigue siendo válido para la nueva definición de realizadores y test, por lo tanto $\mathcal{T}(\theta)=\mathcal{R}_P(\theta)^{\perp}$. Como consecuencia, también se cumple $\mathcal{R}(\theta)=\mathcal{R}_P(\theta)^{\perp\,\top}=\mathcal{T}(\theta)^{\top}$.

La definición de los realizadores de contextos es una simple extensión punto a punto de los realizadores de tipos:

Definición 82 (Realizadores de contextos).

$$(u,\,\eta)\in\mathcal{R}(\pi)\ \ \textit{si y sólo si}\ \ (u,\,\eta\cdot n)\in\mathcal{R}(\pi\cdot n)\ \ \textit{para todo}\ n<|\pi|\ \ .$$

La Definición 79 es válida siempre que los operadores ortogonales se apliquen a conjuntos completos. Ahora demostraremos que los conjuntos $\mathcal{R}_P(\theta)$ y $\mathcal{R}(\theta)$ son completos, y eso nos permitirá usar esa definición cuando necesitemos demostrar que cierto elemento $(u,\,\epsilon)$ pertenece al conjunto $\mathcal{R}(\theta)$; para ello sólo será necesario verificar que satisface todos los tests $(r,\,\tau)\in\mathcal{T}(\theta)$ tal que $r\le u$.

Lema 28. *El conjunto $\mathcal{R}_P(\theta)$ es completo.*

Prueba. Se demuestra por casos en el tipo θ.

- Caso **unit**. Es claro que $\mathcal{R}_P(\textbf{unit})$ es completo, puesto que $(u,\,(\texttt{Unit},\,\eta))$ pertenece a $\mathcal{R}_P(\textbf{unit})$ para todo $u\in\mathbb{N}$.
- Caso $\theta\to\theta'$. Tomemos $(u,\,(\texttt{Grab}\triangleright i,\,\eta))\in\mathcal{R}_P(\theta\to\theta')$ y $u'\le u$, y veamos $(u',\,(\texttt{Grab}\triangleright i,\,\eta))\in\mathcal{R}_P(\theta\to\theta')$. Sea $u''\le u'$ y $(u'',\,\alpha)\in\mathcal{R}(\theta)$. Como $u''\le u$, y $(u,\,(\texttt{Grab}\triangleright i,\,\eta))\in\mathcal{R}_P(\theta\to\theta')$, entonces obtenemos $(u'',\,(i,\,\alpha::\eta))\in\mathcal{R}(\theta')$.

□

El Lema 26 establece que el conjunto de realizadores también es completo.

Lema 29. *El conjunto $\mathcal{R}(\theta)$ es completo.*

Es fácil ver que el conjunto $\mathcal{R}(\pi)$ es también completo, pues $\mathcal{R}(\pi)$ se define posicionalmente en términos de $\mathcal{R}(\theta)$.

Otra propiedad de $\mathcal{R}(\theta)$ es que siempre incluye a los pares de la forma $(0,\ \alpha)$ para cualquier $\alpha \in \mathit{MClos}$, esto es válido como consecuencia de la propiedad (1) de la relación $\perp\!\!\!\perp_r$.

Lema 30. $(0,\ \alpha) \in \mathcal{R}(\theta)$.

Prueba. Como $\mathcal{R}(\theta) = \mathcal{T}(\theta)^\top$, debemos ver que para todo $(r,\ s) \in \mathcal{T}(\theta)$ con $r \leq 0$ se cumple $(\alpha,\ s) \in \perp\!\!\!\perp_r$. Esto fuerza que $r = 0$, y como $\perp\!\!\!\perp_0 = \mathit{Conf}$ se tiene $(\alpha,\ s) \in \perp\!\!\!\perp_0$. □

De manera similar al Lema 23, se puede construir un test de tipo funcional $\theta \to \theta'$ combinando un realizador de tipo θ con un test de tipo θ'.

Lema 31. *Si $(u,\ \alpha) \in \mathcal{R}(\theta)$ y $(u,\ s) \in \mathcal{T}(\theta')$, entonces $(u,\ \alpha :: s) \in \mathcal{T}(\theta \to \theta')$.*

Prueba. Para ver $(u,\ \alpha :: s) \in \mathcal{T}(\theta \to \theta')$, tomemos $(u',\ \widehat{\alpha}) \in \mathcal{R}_P(\theta \to \theta')$ tal que $u' \leq u$, y veamos $(\widehat{\alpha},\ \alpha :: s) \in \perp\!\!\!\perp_{u'}$. Por definición de $\mathcal{R}_P(\theta \to \theta')$ se tiene que $\widehat{\alpha}$ tiene la forma $(u',\ (\mathtt{Grab} \triangleright i,\ \eta))$ para algún $i \in \mathit{Code}$ y $\eta \in \mathit{MEnv}$. Como $\mathcal{R}(\theta)$ y $\mathcal{T}(\theta')$ son completos, se tiene $(u',\ \alpha) \in \mathcal{R}(\theta)$ y $(u',\ s) \in \mathcal{T}(\theta')$. Por definición de $\mathcal{R}_P(\theta \to \theta')$ se tiene que $(u',\ (i,\ \alpha :: \eta)) \in \mathcal{R}(\theta')$. Por lo tanto obtenemos $(i,\ \alpha :: \eta,\ s) \in \perp\!\!\!\perp_{u'}$, y como $\perp\!\!\!\perp_{u'}$ es cerrado por anti-ejecución, podemos concluir que $(\widehat{\alpha},\ \alpha :: s) = (\mathtt{Grab} \triangleright i,\ \eta,\ \alpha :: s) \in \perp\!\!\!\perp_{u'}$. □

Finalmente podemos demostrar un lema similar al de adecuación, excepto que los realizadores y los tests están indexados con números naturales. La demostración para el caso del operador de punto fijo se logra haciendo inducción en el índice, usando el Lema 30 en el caso base (cuando el índice es igual a 0).

Lema 32. *Si $(u,\ \eta) \in \mathcal{R}(\pi)$ y $\pi \vdash t : \theta$, se tiene $(u,\ (\!|\, t \,|\!),\ \eta)) \in \mathcal{R}(\theta)$.*

Prueba. Procedemos por inducción en la derivación de $\pi \vdash t : \theta$.

- Caso (Abs). Usando que $\mathcal{R}_P(\theta \to \theta') \subseteq \mathcal{R}(\theta \to \theta')$ demostraremos que se cumple $(u,\ (\!|\, \lambda\, t \,|\!),\ \eta)) = (u,\ (\mathtt{Grab} \triangleright (\!|\, t \,|\!),\ \eta)) \in \mathcal{R}_P(\theta \to \theta')$. Sean $u' \leq u$ y $(u',\ \alpha) \in \mathcal{R}(\theta)$, debemos ver $(u',\ (\!|\, t \,|\!),\ \alpha :: \eta)) \in \mathcal{R}(\theta')$. Como $\mathcal{R}(\pi)$ es completo, se tiene $(u',\ \eta) \in \mathcal{R}(\pi)$. Por lo tanto, por definición de $\mathcal{R}(\theta :: \pi)$, se tiene $(u',\ \alpha :: \eta) \in \mathcal{R}(\theta :: \pi)$. Luego, por hipótesis inductiva, obtenemos $(u',\ (\!|\, t \,|\!),\ \alpha :: \eta)) \in \mathcal{R}(\theta')$ como queríamos.

- Caso (Var). Debemos ver $(u, (\!| \bar{n} |\!), \eta)) = (u, (\mathtt{Access}\ n, \eta)) \in \mathcal{R}(\theta)$, donde $\pi \cdot n = \theta$. Como $\mathcal{R}(\theta) = \mathcal{T}(\theta)^\top$, debemos tomar $(u', s) \in \mathcal{T}(\theta)$ tal que $u' \leq u$ y demostrar $(\mathtt{Access}\ n, \eta, s) \in \perp\!\!\!\perp_{u'}$. Como $\perp\!\!\!\perp_{u'}$ es cerrada por anti-ejecución, basta con verificar que $(\eta \cdot n, s) \in \perp\!\!\!\perp_{u'}$. Dado que $\mathcal{R}(\pi)$ es completo, se tiene $(u', \eta) \in \mathcal{R}(\pi)$, y por lo tanto $(u', \eta \cdot n) \in \mathcal{R}(\theta)$. Como $(u', s) \in \mathcal{T}(\theta)$, se tiene $(\eta \cdot n, s') \in \perp\!\!\!\perp_{u'}$.

- Caso (App). Debemos ver que $(u, (\mathtt{Push}\ (\!| t' |\!) \triangleright (\!| t |\!), \eta)) \in \mathcal{R}(\theta')$. Tomamos $(u', s) \in \mathcal{T}(\theta')$ tal que $u' \leq u$ y demostremos que se cumple $(\mathtt{Push}\ (\!| t' |\!) \triangleright (\!| t |\!), \eta, s) \in \perp\!\!\!\perp_{u'}$. Por anti-ejecución, basta con ver $((\!| t |\!), \eta, \alpha :: s) \in \perp\!\!\!\perp_{u'}$ donde $\alpha = ((\!| t' |\!), \eta)$. Como $(u', \eta) \in \mathcal{R}(\pi)$, se tiene $(u', ((\!| t |\!), \eta)) \in \mathcal{R}(\theta \to \theta')$ por hipótesis inductiva. Se obtiene también por hipótesis inductiva que $(u', \alpha) \in \mathcal{R}(\theta)$. Luego, como $(u', s) \in \mathcal{T}(\theta')$, se tiene por Lema 31 que $(u', \alpha :: s) \in \mathcal{T}(\theta \to \theta')$. Concluimos entonces que $((\!| t |\!), \eta, \alpha :: s) \in \perp\!\!\!\perp_{u'}$.

- Caso (Unit). Trivial, ya que $(u, ((\!| \diamond |\!), \eta)) = (u, (\mathtt{Unit}, \eta)) \in \mathcal{R}_P(\mathbf{unit})$ por definición de $\mathcal{R}_P(\mathbf{unit})$.

- Caso (Rec). Veamos $(u, ((\!| rec\ t |\!), \eta)) = (u, (\mathtt{Fix} \triangleright (\!| t |\!), \eta)) \in \mathcal{R}(\theta)$. Procedemos por inducción en el índice u.

  - Caso 0. Directo por Lema 30.
  - Caso $u+1$. Sea $\alpha = (\mathtt{Fix} \triangleright (\!| t |\!), \eta)$, queremos ver $(u+1, \alpha) \in \mathcal{R}(\theta)$ asumiendo $(u, \alpha) \in \mathcal{R}(\theta)$. Como $\mathcal{R}(\theta) = \mathcal{T}(\theta)^\top$, debemos tomar $(u', s) \in \mathcal{T}(\theta)$ tal que $u' \leq u+1$, y demostrar $(\alpha, s) \in \perp\!\!\!\perp_{u'}$. Si $u' \leq u$, se obtiene directamente $(\alpha, s) \in \perp\!\!\!\perp_{u'}$ pues se asumió $(u, \alpha) \in \mathcal{R}(\theta)$. Supongamos ahora $u' = u+1$, debemos ver $(\alpha, s) \in \perp\!\!\!\perp_{u+1}$. Tenemos $(u+1, s) \in \mathcal{T}(\theta)$, luego $(u, s) \in \mathcal{T}(\theta)$ por ser $\mathcal{T}(\theta)$ completo, y por lo tanto por Lema 31 tenemos $(u, \alpha :: s) \in \mathcal{R}(\theta \to \theta)$. Por hipótesis inductiva tenemos $(u, ((\!| t |\!), \eta)) \in \mathcal{R}(\theta \to \theta)$, y por lo tanto obtenemos $((\!| t |\!), \eta, \alpha :: s) \in \perp\!\!\!\perp_u$. Como $(\alpha, s) \longmapsto ((\!| t |\!), \eta, \alpha :: s)$, aplicamos la propiedad 2 de la Definición 78 para obtener $(\alpha, s) \in \perp\!\!\!\perp_{u+1}$.

□

A partir del capítulo siguiente aplicaremos este enfoque para demostrar la corrección de la compilación. En el Capítulo 6 lo aplicamos sobre un lenguaje con evaluación normal y la máquina de Krivine, mientras que en el Capítulo 7 usaremos la máquina de Sestoft modelando el orden de evaluación lazy.

Capítulo 6

Compilación Certificada Usando Semántica Denotacional

Hemos visto que la semántica de un lenguaje se puede describir operacionalmente a través de reglas que especifican cómo evaluar un término hasta obtener un valor. Una manera alternativa de definir semántica es asignar a cada término del lenguaje un objeto matemático (llamado *denotación*) que describa su significado. La semántica denotacional provee una representación abstracta de los programas que puede resultar más natural que una definición operacional sobre todo en lenguajes funcionales de alto nivel.

En este capítulo aplicaremos un esquema de realizabilidad sobre la máquina de Krivine para obtener una prueba de corrección de la compilación con respecto a la semántica denotacional del lenguaje fuente. En lugar de definir realizadores de tipos, como en el Capítulo 5, esta vez tendremos realizadores de objetos denotacionales.

Usaremos relaciones lógicas indexadas (*step-indexing logical relations* [7, 11]) y biortogonalidad [94] para capturar la noción de corrección de una manera composicional y modular. Esas dos técnicas han sido utilizadas de manera combinada para obtener pruebas de corrección de compiladores [15, 16, 62] y también en otras áreas como equivalencia de programas [51].

A diferencia del trabajo realizado por Benton et al. [15, 16], que utiliza step-indexing y biortogonalidad sobre lenguajes estrictos y diferentes variantes de la máquina SECD [73], en este trabajo nos enfocamos en un lenguaje con evaluación normal y utilizamos la máquina de Krivine [71] como entorno de ejecución.

El enfoque que utilizamos se basa en el esquema de realizabilidad presentado en el Capítulo 5, con las modificaciones necesarias para demostrar la corrección respecto a la semántica denotacional.

Comenzaremos con la definición del lenguaje fuente y su semántica denota-

cional, luego presentaremos la máquina abstracta, el compilador y finalmente introduciremos las relaciones lógicas y la prueba de corrección.

6.1 Sintaxis

El lenguaje fuente que utilizaremos incluye los términos del cálculo lambda junto un operador de punto fijo, constantes enteras, operadores aritméticos de aridad finita, pares, proyecciones y un condicional.

Definición 83 (Términos).

$$\begin{array}{rcl} t \in \mathit{Term} & ::= & \lambda\, t \mid \bar{n} \mid t\, t' \mid \mathit{rec}\, t \mid \underline{k} \mid \ominus^n (t_1, \ldots, t_n) \\ & \mid & (t_0, t_1) \mid \mathit{fst}\, t \mid \mathit{snd}\, t \mid \mathit{ifz}\ t\ .\ t' \end{array}$$

El último constructor $\mathit{ifz}\ t\ .\ t'$ es una proyección condicional, cuya semántica se puede describir informalmente como sigue: de acuerdo a si el valor numérico de t es igual 0 o distinto de 0 se elije respectivamente la primera o la segunda componente del par que resulta de evaluar t'. Elegimos esta forma de condicional por conveniencia; un constructor más tradicional del condicional como $\mathit{ifz}\, t\ \mathit{then}\, t_1\, \mathit{else}\, t_2$ se puede expresar como $\mathit{ifz}\ t\ .\ (t_1, t_2)$. Escribimos $\ominus^n$ para representar cualquier operador aritmético con aridad $n > 0$, los operadores se escriben de manera prefija y no pueden ser parcialmente aplicados. Nuevamente utilizamos los índices de De Bruijn para representar las variables del lenguaje.

El sistema de tipos consta del tipo **int** (que por simplicidad será el único tipo básico), tipos funcionales $\theta \to \theta'$ y tipos de producto $\theta \times \theta'$.

Definición 84 (Tipos).

$$\theta, \theta' \in \mathit{Type} \quad ::= \quad \mathbf{int} \mid \theta \to \theta' \mid \theta \times \theta'$$

Si queremos conocer el tipo de un término, primero es necesario determinar el tipo de cada una de sus variables libres. Como ya hemos observado, los contextos son útiles para ese propósito, manteniendo en la n-ésima posición el tipo de la variable libre $\bar{n}$ en el término.

Definición 85 (Contextos).

$$\pi \in \mathit{Ctx} \quad ::= \quad [] \mid \theta :: \pi$$

Como antes, utilizaremos juicios de tipado de la forma $\pi \vdash t : \theta$ que establecen que el término t tiene tipo θ bajo el contexto π. Las reglas de tipado indican cómo construir derivaciones correctas de un juicio. Para el cálculo lambda y el operador de punto fijo, las reglas se mantienen sin cambios respecto a las que utilizamos en el Capítulo 5. Se incorporan nuevas reglas para tipar los operadores aritméticos, los pares, las proyecciones y los condicionales.

Definición 86 (Reglas de tipado).

$$\text{(Abs)}\ \frac{\theta :: \pi \vdash t : \theta'}{\pi \vdash \lambda\, t : \theta \to \theta'} \qquad \text{(App)}\ \frac{\pi \vdash t : \theta \to \theta' \qquad \pi \vdash t' : \theta}{\pi \vdash t\ t' : \theta'}$$

$$\text{(Var)}\ \frac{}{\pi \vdash \bar{n} : \theta}\ \pi \cdot n = \theta \qquad \text{(Rec)}\ \frac{\pi \vdash t : \theta \to \theta}{\pi \vdash \mathit{rec}\ t : \theta}$$

$$\text{(Const)}\ \frac{}{\pi \vdash \underline{k} : \mathbf{int}} \qquad \text{(Op)}\ \frac{\pi \vdash t_j : \mathbf{int},\ \forall j \in \{1, \ldots, n\}}{\pi \vdash \ominus^n (t_1, \ldots, t_n) : \mathbf{int}}$$

$$\text{(Fst)}\ \frac{\pi \vdash t : \theta \times \theta'}{\pi \vdash \mathit{fst}\ t : \theta} \qquad \text{(Snd)}\ \frac{\pi \vdash t : \theta \times \theta'}{\pi \vdash \mathit{snd}\ t : \theta'}$$

$$\text{(Pair)}\ \frac{\pi \vdash t_0 : \theta \qquad \pi \vdash t_1 : \theta'}{\pi \vdash (t_0, t_1) : \theta \times \theta'}$$

$$\text{(Cond)}\ \frac{\pi \vdash t : \mathbf{int} \qquad \pi \vdash t' : \theta \times \theta}{\pi \vdash \mathit{ifz}\ t\ .\ t' : \theta}$$

Notemos que la regla Op del operador aritmético $\ominus^n (t_1, \ldots, t_n)$ exige que cada término t_r tenga tipo **int**. En el condicional $\mathit{ifz}\ t\ .\ t'$, el término t' debe tener tipo $\theta \times \theta$, esto es, ambos elementos del par deben tener el mismo tipo. El tipo *bool* no fue incorporado al lenguaje, pero podemos modelar las comparaciones booleanas como operadores aritméticos que devuelven 0 o 1 según su valor de verdad. Con el condicional y el operador de punto fijo contamos con la expresividad suficiente para definir funciones recursivas no triviales.

Es importante observar que estas reglas de tipado permiten construir derivaciones distintas para el mismo juicio de tipado (ver Ejemplo 2) y a su vez permiten asociar más de un tipo al mismo término (pensar por ejemplo en el término identidad $\lambda\, \bar{0}$ que tiene infinitos tipos).

6.2 Semántica

Definiciones elementales de teoría de dominios

La semántica del lenguaje se define usando *dominios*, que son básicamente conjuntos parcialmente ordenados (posets) con cierta estructura adicional que garantiza, entre otras cosas, la existencia de puntos fijos de funciones continuas y que permite la interpretación de operadores de recursión. Si bien no profundizaremos en teoría de dominios, repasaremos las definiciones básicas que son relevantes en este capítulo. Para una introducción más completa a la teoría referimos al lector a [5, 54, 100]. Comenzaremos con la definición de *cadena*.

Definición 87 (Cadena). *Una cadena sobre el poset $\langle P, \sqsubseteq \rangle$, es una secuencia infinita p_r de elementos de P tales que*

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \cdots \ ,$$

es decir, $p_r \sqsubseteq p_{r+1}$ para todo $r \in \mathbb{N}$.

El límite (o supremo) de una cadena p_r sobre $\langle P, \sqsubseteq \rangle$ es un elemento $q \in P$ tal que

- q es cota superior de p_r, esto es, para todo $r \in \mathbb{N}$, se cumple $p_r \sqsubseteq q$.
- para todo $p' \in P$, si p' es cota superior de p_r, entonces $q \sqsubseteq p'$.

Es decir, el límite es la menor cota superior de la cadena p_r. Cuando el límite de la cadena p_r existe, se lo denota $\bigsqcup_{r \in \mathbb{N}}^{\infty} p_r$.

Definición 88 (Predominio y dominio). *Un poset $\langle P, \sqsubseteq \rangle$ es un predominio si toda cadena sobre $\langle P, \sqsubseteq \rangle$ tiene límite. Si el predominio $\langle P, \sqsubseteq \rangle$ tiene un elemento distinguido $\bot \in P$ tal que $\bot \sqsubseteq p$ para todo $p \in P$, entonces se lo llama dominio.*

Como ejemplos consideremos:

- El poset $\langle \mathbb{N}, \leq \rangle$ (donde $\leq$ es el orden usual de los números naturales) no es un predominio pues no toda cadena tiene límite.
- El poset $\langle \mathbb{N}, \geq \rangle$ es un predominio pero no es un dominio.
- El poset $\langle X, \sqsubseteq \rangle$ donde X es un conjunto finito (y $\sqsubseteq$ es un orden parcial sobre X) es un dominio.

- El poset $\langle \mathcal{P}(\mathbb{N}),\ \subseteq \rangle$ (el conjunto de todos los subconjuntos de $\mathbb{N}$ junto con la inclusión) es un dominio.
- El poset $\langle A, \sqsubseteq \rangle$ donde $a \sqsubseteq a'$ si y sólo si $a = a'$ es un predominio (llamado *predominio discreto*).

Otro ejemplo de dominio es el denominado *dominio llano*:

Definición 89 (Dominio llano). *Dado un conjunto P, el dominio llano es el poset $\langle P_\perp, \sqsubseteq \rangle$ donde $P_\perp = \{ \iota_\uparrow p \mid p \in P \} \cup \{ \perp \}$, y el orden $\sqsubseteq$ se define por 1. $\iota_\uparrow p \sqsubseteq \iota_\uparrow p'$ si y sólo si $p = p'$, 2. $\perp \sqsubseteq \iota_\uparrow p$ para todo $p \in P$, 3. $\perp \sqsubseteq \perp$.*

El producto cartesiano de dominios, donde el orden se define punto a punto, también es un dominio:

Definición 90 (Producto). *Si $\langle P, \sqsubseteq_P \rangle$ y $\langle Q, \sqsubseteq_Q \rangle$ son dominios, se define el dominio $\langle P \times Q, \sqsubseteq \rangle$ donde*

$$(p,\ q) \sqsubseteq (p',\ q') \quad \text{si y sólo si} \quad p \sqsubseteq_P p' \ \ y \ \ q \sqsubseteq_Q q' \ .$$

Una función f de P a Q es monótona (con respecto a $\sqsubseteq_P$ y $\sqsubseteq_Q$) si para todo $p,\ p' \in P$, $p \sqsubseteq_P p'$ implica $fp \sqsubseteq_Q fp'$). Cuando p_r es una cadena sobre $\langle P, \sqsubseteq_P \rangle$ y f es monótona, se tiene que $q_r = fp_r$ es una cadena sobre $\langle Q, \sqsubseteq_Q \rangle$. Una función monótona es además continua si preserva límites de cadenas:

Definición 91 (Continuidad). *Si $\langle P, \sqsubseteq_P \rangle$ y $\langle Q, \sqsubseteq_Q \rangle$ son predominios, y la función $f : P \to Q$ es monótona respecto a $\sqsubseteq_P$ y $\sqsubseteq_Q$, entonces f es continua si se cumple*

$$f(\bigsqcup_{r \in \mathbb{N}}^{\infty} p_r) = \bigsqcup_{r \in \mathbb{N}}^{\infty} fp_r \ ,$$

para toda cadena p_r sobre $\langle P, \sqsubseteq_P \rangle$.

El conjunto de funciones continuas entre predominios también forma un predominio, donde el orden de las funciones se define de manera extensional.

Definición 92 (Espacio de funciones continuas). *Si $\langle P, \sqsubseteq_P \rangle$ y $\langle Q, \sqsubseteq_Q \rangle$ son predominios, entonces $\langle [P \to Q], \sqsubseteq \rangle$ es el predominio de funciones continuas*

de P a Q, donde el orden $\sqsubseteq$ queda definido como:

$$f \sqsubseteq g \quad \textit{si y sólo si} \quad \textit{para todo } p \in P, f\,p \sqsubseteq_Q g\,p \ .$$

Si $\langle Q, \leq_Q \rangle$ es un dominio, entonces $[P \to Q]$ también lo es: el menor elemento es la función que asigna $\bot \in Q$ a todo elemento de P (esto es $\bot(p) = \bot$ para todo $p \in P$).

El conocido teorema del punto fijo establece que todas las funciones continuas tienen un punto fijo:

Teorema 8. *Si $\langle D, \sqsubseteq \rangle$ es un dominio, y f es una función continua de D en D, entonces*

$$d = \bigsqcup_{r \in \mathbb{N}}^{\infty} f^r \bot, \quad \textit{donde } f^0 = \bot \textit{ y } f^{r+1} = f^r \circ f,$$

es el menor punto fijo de f (es decir $f\,d = d$ y si $f\,d' = d'$ entonces $d \sqsubseteq d'$).

Denotamos con $\mathrm{Y}_D f$ al menor punto fijo de f en el dominio D (el subíndice D se omite si queda claro del contexto donde se usa). El punto fijo de una función es útil para definir la semántica de funciones recursivas, como veremos a continuación.

Semántica del lenguaje

La semántica denotacional del lenguaje se define de manera intrínseca (en el sentido de Reynolds [101]), esto es, se define la semántica de las derivaciones de juicios de tipado en lugar de los términos del lenguaje, y esto implica que sólo tienen semántica los términos bien tipados. Comenzamos asignando un dominio a cada tipo del lenguaje:

Definición 93 (Semántica de tipos).

$$\begin{aligned} [\![\mathbf{int}]\!] &= \mathbb{Z}_\bot \\ [\![\theta \to \theta']\!] &= [[\![\theta]\!] \to [\![\theta']\!]] \\ [\![\theta \times \theta']\!] &= [\![\theta]\!] \times [\![\theta']\!] \end{aligned}$$

La semántica del tipo **int** es el dominio llano de los números enteros junto con un elemento distinguido $\bot$ (ver Definición 89), la semántica del tipo $\theta \to \theta'$ es el espacio de funciones continuas entre $[\![\theta]\!]$ y $[\![\theta']\!]$, y por último, la semántica del tipo $\theta \times \theta'$ es el producto cartesiano $[\![\theta]\!] \times [\![\theta']\!]$.

Es posible también asignar a cada contexto π un dominio $[\![\pi]\!]$, simplemente tomando el producto cartesiano de todos los dominios $[\![\pi \cdot n]\!]$ para todo natural

$n < |\pi|$. Esto se puede definir por inducción estructural en el contexto como sigue:

Definición 94 (Semántica de contextos).

$$\begin{aligned} [\![\, [\,] \,]\!] &= \{\, \varnothing \,\} \\ [\![\, \theta :: \pi \,]\!] &= [\![\, \theta \,]\!] \times [\![\, \pi \,]\!] \end{aligned}$$

Recordemos que los elementos $\rho \in [\![\, \pi \,]\!]$ se denominan ambientes. Utilizamos $\varnothing$ para denotar al ambiente vacío (correspondiente a una o-tupla).

La semántica de una derivación de tipos con conclusión $\pi \vdash t : \theta$ es una función continua de $[\![\, \pi \,]\!]$ en $[\![\, \theta \,]\!]$. Dicha función se define inducción en la estructura de la derivación, especificando una ecuación por cada regla de tipado.

Definición 95 (Semántica denotacional).

$$\begin{aligned}
&[\![\, \lambda\, t \,]\!]_{\pi,\, \theta \to \theta'}\, \rho && = \hat{\lambda}\, d\,.\, [\![\, t \,]\!]_{\theta :: \pi,\, \theta'}\, (d,\, \rho) \\
&[\![\, \overline{n} \,]\!]_{\pi,\, \theta}\, \rho && = \rho \swarrow n \\
&[\![\, t\ t' \,]\!]_{\pi,\, \theta'}\, \rho && = ([\![\, t \,]\!]_{\pi,\, \theta \to \theta'}\, \rho)\ ([\![\, t' \,]\!]_{\pi,\, \theta}\, \rho) \\
&[\![\, rec\ t \,]\!]_{\pi,\, \theta}\, \rho && = Y_{[\![\, \theta \,]\!]}\, ([\![\, t \,]\!]_{\pi,\, \theta \to \theta}\, \rho) \\
&[\![\, \underline{k} \,]\!]_{\pi,\, \mathbf{int}}\, \rho && = \iota_{\uparrow}\, k \\
&[\![\, \ominus^n\, (t_1, \ldots, t_n) \,]\!]_{\pi,\, \mathbf{int}}\, \rho && = \ominus^n_{\perp}\, (d_1, \ldots, d_n) \\
& && \textit{donde}\ d_j = [\![\, t_j \,]\!]_{\pi,\, \mathbf{int}}\, \rho \\
&[\![\, fst\ t \,]\!]_{\pi,\, \theta}\, \rho && = d_0 \\
& && \textit{donde}\ (d_0,\, d_1) = [\![\, t \,]\!]_{\pi,\, \theta \times \theta'}\, \rho \\
&[\![\, snd\ t \,]\!]_{\pi,\, \theta'}\, \rho && = d_1 \\
& && \textit{donde}\ (d_0,\, d_1) = [\![\, t \,]\!]_{\pi,\, \theta \times \theta'}\, \rho \\
&[\![\, (t_0,\, t_1) \,]\!]_{\pi,\, \theta \times \theta'}\, \rho && = ([\![\, t_0 \,]\!]_{\pi,\, \theta}\, \rho,\, [\![\, t_1 \,]\!]_{\pi,\, \theta'}\, \rho) \\
&[\![\, ifz\ t\ .\ t' \,]\!]_{\pi,\, \theta}\, \rho && = IFZ_{\theta}\ d\ (d_0,\, d_1) \\
& && \textit{donde}\ \ d = [\![\, t \,]\!]_{\pi,\, \mathbf{int}}\, \rho\ y\ (d_0,\, d_1) = [\![\, t' \,]\!]_{\pi,\, \theta \times \theta}\, \rho
\end{aligned}$$

Una propiedad de la semántica es que depende únicamente de la conclusión de la derivación, esto es, dos derivaciones con la misma conclusión tienen la misma semántica. Esta propiedad se denomina *coherencia*, y se ha demostrado para lenguajes incluso más grandes que el que usamos en este capítulo [97]. Debido a esa propiedad podemos escribir $[\![\, t \,]\!]_{\pi,\, \theta}$ sin ambigüedad para denotar a la semántica de cualquier derivación con conclusión $\pi \vdash t : \theta$.

Hemos omitido aquí la demostración de que $[\![\, t \,]\!]_{\pi,\, \theta}$ es efectivamente una función continua. La prueba consiste en mostrar que $[\![\, t \,]\!]_{\pi,\, \theta}$ puede expresarse como composición de funciones continuas, lo cual puede observarse en la formalización en Coq que acompaña a este capítulo.

Para los términos del cálculo lambda, la semántica se define de la misma manera como vimos en el Capítulo 1. La semántica del operador de punto fijo $[\![rec\ \ t]\!]_{\pi,\,\theta}\rho$ es justamente el menor punto fijo de la función $[\![t]\!]_{\pi,\,\theta\to\theta}\rho$, que siempre existe por el Teorema 8.

Para cada operador $\ominus^n$ se asume la existencia de una función $\ominus^n : \mathbb{Z}^n \to \mathbb{Z}$ (que justamente representa la operación asociada con $\ominus^n$). Además, para cada una de esas funciones, se define la función $\ominus^n_\perp : \mathbb{Z}^n_\perp \to \mathbb{Z}_\perp$ como sigue:

Definición 96 (Aplicación de un operador n-ario).

$$\ominus^n_\perp(d_1, \ldots, d_n) = \begin{cases} \perp & \text{si existe } r \leq n \text{ tal que } d_r = \perp \\ \ominus^n(k_1, \ldots, k_n) & \text{si } d_r = \iota_\uparrow k_r \text{ para todo } r \leq n. \end{cases}$$

Notemos que la aplicación del operador se hace de manera *estricta*, esto es, para obtener el valor de la aplicación se debe conocer la constante k_r para todos los argumentos d_r del operador, y si alguno de esos argumentos es igual a $\perp \in \mathbb{Z}_\perp$, la aplicación completa del operador también lo es.

Para definir la semántica de la proyección condicional hemos utilizado una función $IFZ_\theta : \mathbb{Z}_\perp \to [\![\theta \times \theta]\!] \to [\![\theta]\!]$ definida de la siguiente manera.

Definición 97 (Proyección condicional).

$$IFZ_\theta\ d\ (d_0, d_1) = \begin{cases} \perp & si\ d = \perp \\ d_0 & si\ d = \iota_\uparrow 0 \\ d_1 & si\ d = \iota_\uparrow k,\ k \neq 0 \ . \end{cases}$$

La función selecciona una de las componentes del par (d_0, d_1) de acuerdo al valor de $d \in \mathbb{Z}_\perp$, esto es, es igual a $\perp$ si $d = \perp$, igual a d_0 si $d = \iota_\uparrow 0$, o bien igual a d_1 en otro caso. P

Para cada derivación con conclusión $\pi \vdash t : \theta$, definiremos una cadena $[\![t]\!]^r_{\pi,\,\theta}$ en el espacio de funciones $[[\![\pi]\!] \to [\![\theta]\!]]$. El primer elemento de la cadena $[\![t]\!]^0_{\pi,\,\theta}$ es igual a $\perp$, para cualquier derivación. Para $r > 0$ se define por inducción en la derivación del juicio, como se indica a continuación.

Definición 98 (Cadena semántica).

$$
\begin{array}{lcl}
[\![\, t \,]\!]^0_{\pi,\,\theta}\,\rho & = & \bot,\ \textit{para todo } t \in \textit{Term} \\
[\![\, \lambda\, t \,]\!]^{r+1}_{\pi,\,\theta\to\theta'}\,\rho & = & \widehat{\lambda}\, d\,.\, [\![\, t \,]\!]^{r}_{\theta::\,\pi,\,\theta'}\,(d,\,\rho) \\
[\![\, \bar{n} \,]\!]^{r+1}_{\pi,\,\theta}\,\rho & = & \rho \;\swarrow\; n \\
[\![\, t\ t' \,]\!]^{r+1}_{\pi,\,\theta'}\,\rho & = & ([\![\, t \,]\!]^{r}_{\pi,\,\theta\to\theta'}\,\rho)\ ([\![\, t' \,]\!]^{r}_{\pi,\,\theta}\rho) \\
[\![\, \textit{rec}\ t \,]\!]^{r+1}_{\pi,\,\theta}\,\rho & = & ([\![\, t \,]\!]^{r}_{\pi,\,\theta\to\theta}\,\rho)\ ([\![\, \textit{rec}\ t \,]\!]^{r}_{\pi,\,\theta}\,\rho) \\
[\![\, \underline{k} \,]\!]^{r+1}_{\pi,\,\mathbf{int}}\,\rho & = & \iota_{\uparrow}\, k \\
[\![\, \ominus^n\,(t_1,\ldots,t_n) \,]\!]^{r+1}_{\pi,\,\mathbf{int}}\,\rho & = & \ominus^n_{\bot}\,(d_1,\ \ldots\ ,d_n) \\
 & & \textit{donde } d_j = [\![\, t_j \,]\!]^{r}_{\pi,\,\mathbf{int}}\,\rho \\
[\![\, \textit{fst}\ t \,]\!]^{r+1}_{\pi,\,\theta}\,\rho & = & d_0 \\
 & & \textit{donde } (d_0,\,d_1) = [\![\, t \,]\!]^{r}_{\pi,\,\theta\,\times\,\theta'}\,\rho \\
[\![\, \textit{snd}\ t \,]\!]^{r+1}_{\pi,\,\theta'}\,\rho & = & d_1 \\
 & & \textit{donde } (d_0,\,d_1) = [\![\, t \,]\!]^{r}_{\pi,\,\theta\,\times\,\theta'}\,\rho \\
[\![\, (t_0,\,t_1) \,]\!]^{r+1}_{\pi,\,\theta\,\times\,\theta'}\,\rho & = & ([\![\, t_0 \,]\!]^{r}_{\pi,\,\theta}\rho,\ [\![\, t_1 \,]\!]^{r}_{\pi,\,\theta'}\,\rho) \\
[\![\, \textit{ifz}\ t\ .\ t' \,]\!]^{r+1}_{\pi,\,\theta}\,\rho & = & \mathit{IFZ}_{\theta}\ d\ (d_0,\,d_1) \\
 & & \textit{donde}\ \ d = [\![\, t \,]\!]^{r}_{\pi,\,\mathbf{int}}\,\rho\ y\ (d_0,\,d_1) = [\![\, t' \,]\!]^{r}_{\pi,\,\theta\,\times\,\theta}\rho
\end{array}
$$

Como el espacio de funciones $[[\![\, \pi \,]\!] \to [\![\, \theta \,]\!]]$ es un dominio, se tiene que la cadena $[\![\, t \,]\!]^{r}_{\pi,\,\theta}$ tiene límite. Además, se puede demostrar que ese límite es igual a la semántica denotacional de la derivación.

Lema 33. *Para toda derivación de* $\pi \vdash t : \theta$ *se tiene* $\bigsqcup_{r\in\mathbb{N}}^{\infty} [\![\, t \,]\!]^{r}_{\pi,\,\theta} = [\![\, t \,]\!]_{\pi,\,\theta}$.

Omitimos la demostración de que $[\![\, t \,]\!]^{r}_{\pi,\,\theta}$ es efectivamente una cadena, y que su límite es $[\![\, t \,]\!]_{\pi,\,\theta}$. La prueba para el caso del término *rec t* utiliza el principio de inducción de punto fijo de Scott (e.g [117, página 166]). El lector puede verificar la prueba completa en la formalización en Coq, junto con todos los resultados auxiliares necesarios para completarla. La utilidad de la cadena semántica quedará clara cuando definamos las relaciones lógicas que establecen la corrección del compilador.

6.3 La máquina abstracta

Nuevamente utilizaremos una variante de la máquina abstracta de Krivine como entorno de ejecución. Además de las instrucciones que corresponden al cálculo lambda, se incorporan nuevas instrucciones para implementar el operador de punto fijo, operadores aritméticos, pares y condicionales. A continuación se definen los distintos componentes que integran la máquina abstracta.

Definición 99 (Componentes).

$$
\begin{array}{lcll}
i,\ i' \in \mathit{Code} & ::= & \texttt{Access}\ n & \textit{(Código)} \\
& \mid & \texttt{Grab} \triangleright i \mid \texttt{Push}\ i' \triangleright i \mid \texttt{Fix} \triangleright i & \\
& \mid & \texttt{Const}\ k \mid \texttt{Frame} \ominus^n & \\
& \mid & \texttt{Pair}\ (i, i') \mid \texttt{Fst} \mid \texttt{Snd} & \\
\alpha \in \mathit{MClos} & ::= & (i,\ \eta) & \textit{(Clausuras de máquina)} \\
\eta \in \mathit{MEnv} & ::= & [] \mid \alpha :: \eta & \textit{(Entornos de máquina)} \\
\ominus^n \in \mathit{Ops} & & & \textit{(Operadores n-arios)} \\
\mu \in \mathit{StackVal} & ::= & \alpha \mid \{\ominus^n \bar{k} \bullet \bar{\alpha}\} \mid \langle \alpha,\ \alpha' \rangle & \textit{(Valores de pila)} \\
s \in \mathit{Stack} & ::= & [] \mid \mu :: s & \textit{(Pilas)} \\
w \in \mathit{Conf} & ::= & (i,\ \eta,\ s) & \textit{(Configuraciones)}
\end{array}
$$

Respecto a las máquinas abstractas que hemos usado anteriormente, la definición de clausura y entorno de máquina se mantienen intactas. Por otro lado, tenemos tres clases de valores de pila: clausuras, frames y pares.

Los frames los habíamos utilizado antes en el Capítulo 3, y su funcionamiento en esta nueva versión es similar. El frame $\{\ominus^n \bar{k} \bullet \bar{\alpha}\}$ almacena los argumentos del operador $\ominus^n$, el primer vector $\bar{k}$ contiene aquellos argumentos ya computados (es decir, se conoce su valor) y el segundo vector $\bar{\alpha}$ contiene las clausuras necesarias para computar el resto de los argumentos. Como invariante de ejecución se cumple que $|\bar{k}| + |\bar{\alpha}| = n - 1$, el argumento restante (para completar los n argumentos necesarios) es el que se está computando en el momento en que el frame se encuentra en la pila (y se utiliza el símbolo $\bullet$ para indicar el lugar donde se alojará el valor del mismo).

El valor de pila $\langle \alpha,\ \alpha' \rangle$ contiene las clausuras necesarias para computar las componentes de un par. Los pares no son estrictos, esto es, sólo se ejecutará el código de la clausura α si es necesario proyectar el par en la primer componente, y de la misma forma con la clausura α' y la segunda componente.

A continuación mostramos las reglas de transición que definen el comportamiento de la máquina. Estas reglas definen una relación $\longmapsto \subseteq \mathit{Conf} \times \mathit{Conf}$, y se puede demostrar que esa relación es determinista.

Definición 100 (Transiciones).

$$
\begin{array}{ll}
(\texttt{Access}\ n,\ \eta,\ s) & \longmapsto\ (i',\ \eta',\ s) \\
 & \textit{si } n < |\,\eta\,| \textit{ y } \eta \cdot n = (i',\ \eta') \\
(\texttt{Grab} \triangleright i,\ \eta,\ \alpha :: s) & \longmapsto\ (i,\ \alpha :: \eta,\ s) \\
(\texttt{Push}\ i' \triangleright i,\ \eta,\ s) & \longmapsto\ (i,\ \eta,\ (i',\ \eta) :: s) \\
(\texttt{Fix} \triangleright i,\ \eta,\ s) & \longmapsto\ (i,\ \eta,\ \alpha :: s) \\
 & \textit{donde } \alpha = (\texttt{Fix} \triangleright i,\ \eta) \\
(\texttt{Pair}\ (i_0, i_1),\ \eta,\ \alpha :: s) & \longmapsto\ (i',\ \eta',\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \\
 & \textit{si } \alpha = (i',\ \eta') \textit{ y } \alpha_r = (i_r,\ \eta) \\
(\texttt{Fst},\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) & \longmapsto\ (i_0,\ \eta_0,\ s) \\
 & \textit{si } \alpha_0 = (i_0,\ \eta_0) \\
(\texttt{Snd},\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) & \longmapsto\ (i_1,\ \eta_1,\ s) \\
 & \textit{si } \alpha_1 = (i_1,\ \eta_1) \\
(\texttt{Frame}\ \ominus^n,\ \eta,\ \alpha_1 :: \bar{\alpha} :: s) & \longmapsto\ (i',\ \eta',\ \{\ominus^n \bullet \bar{\alpha}\} :: s) \\
 & \textit{si } \alpha_1 = (i',\ \eta') \textit{ y } |\,\bar{\alpha}\,| = n - 1 \\
(\texttt{Const}\ k_0,\ \eta,\ \{\ominus^n \bar{k} \bullet \alpha',\ \bar{\alpha}\} :: s) & \longmapsto\ (i',\ \eta',\ \{\ominus^n \bar{k},\ k_0 \bullet \bar{\alpha}\} :: s) \\
 & \textit{donde } \alpha' = (i',\ \eta') \\
(\texttt{Const}\ k_0,\ \eta,\ \{\ominus^n \bar{k} \bullet\} :: s) & \longmapsto\ (\texttt{Const}\ k',\ \eta,\ s) \\
 & \textit{donde } k' = \ominus^n (\bar{k},\ k_0) \textit{ y } |\,\bar{k}\,| = n - 1 \\
(\texttt{Const}\ k_0,\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) & \longmapsto\ (i_u,\ \eta_u,\ s) \\
 & \textit{si } \alpha_r = (i_r,\ \eta) \textit{ y } u = \lfloor\, k_0 \neq 0 \,\rfloor
\end{array}
$$

La instrucción $\texttt{Pair}\ (i_0, i_1)$ comienza la ejecución de la clausura α que se encuentra en el tope de la pila, e inserta en su lugar el par $\langle (i_0,\ \eta),\ (i_1,\ \eta) \rangle$, donde $\eta \in \mathit{MEnv}$ es el entorno de máquina en el momento de la ejecución. Las instrucciones `Fst` y `Snd` se utilizan para ejecutar respectivamente la primera y la segunda componente del par ubicado en el tope de la pila.

La instrucción $\texttt{Frame}\ \ominus^n$ es la encargada de crear un nuevo frame en el tope de la pila, colocando en el mismo el código necesario para computar el valor de todos los argumentos del operador, excepto el primer argumento, cuya clausura comienza a ejecutarse de forma inmediata. Esas clausuras serán las que completarán el frame con los valores de los argumentos cuando se haya determinado su valor.

La instrucción $\texttt{Const}\ k_0$ tiene tres reglas de transición que dependen del valor que se encuentre en el tope de la pila. En la primera regla, el frame ubicado en el tope contiene al menos una clausura α' que todavía no ha sido ejecutada

(correspondiente a uno de los argumentos del operador). En ese caso la constante k_0 se almacena en el frame, y se continúa con la ejecución de la clausura α'. En la segunda regla, la constante k_0 corresponde con el valor del último argumento que faltaba computar. Como se dispone de todos los argumentos, se aplica el operador con el valor de los mismos y se ejecuta una nueva instrucción `Const` k', donde k' es el resultado de esa aplicación. Finalmente, en la tercera regla, la constante k_0 se utiliza para elegir cuál de las dos clausuras del par que está en el tope se debe continuar ejecutando. Se denota con $\lfloor\, k_0 \neq 0 \,\rfloor$ al número cuyo es valor es 1 o 0 dependiendo respectivamente si la expresión $k_0 \neq 0$ es verdadera o falsa. Ese número es el que determina la siguiente clausura a ejecutar.

6.4 Compilación

Se define a continuación la función de traducción de los términos del lenguaje a las instrucciones de la máquina abstracta. Una diferencia respecto a las definiciones anteriores de la función de compilación es que aquí compilamos las derivaciones de tipos y no los términos. Se define de esa manera por compatibilidad con la semántica intrínseca, esto es, necesitamos que los términos que se compilan tengan semántica asociada. De todas formas, es claro que las instrucciones que genera la función dependen solamente del término involucrado en la derivación.

Definición 101 (Compilación).

$$
\begin{array}{lcl}
(\!|\, \bar{n} \,|\!)_{\pi,\,\theta} & = & \texttt{Access}\ n \\
(\!|\, \lambda\, t \,|\!)_{\pi,\,\theta\to\theta'} & = & \texttt{Grab} \triangleright (\!|\, t \,|\!)_{\theta::\pi,\,\theta'} \\
(\!|\, t\ t' \,|\!)_{\pi,\,\theta'} & = & \texttt{Push}\ ((\!|\, t' \,|\!)_{\pi,\,\theta}) \triangleright (\!|\, t \,|\!)_{\pi,\,\theta\to\theta'} \\
(\!|\, \mathit{rec}\ t \,|\!)_{\pi,\,\theta} & = & \texttt{Fix} \triangleright ((\!|\, t \,|\!)_{\pi,\,\theta\to\theta}) \\
(\!|\, \underline{k} \,|\!)_{\pi,\,\mathbf{int}} & = & \texttt{Const}\ k \\
(\!|\, \ominus^n\ (t_1, \ldots, t_n) \,|\!)_{\pi,\,\mathbf{int}} & = & \texttt{Push}\ i_n \triangleright \ldots \triangleright \texttt{Push}\ i_1 \triangleright \texttt{Frame}\ \ominus^n \\
 & & \textit{donde}\ i_r = (\!|\, t_r \,|\!)_{\pi,\,\mathbf{int}}\ \textit{con}\ r \in \{\, 1, \ldots, n \,\} \\
(\!|\, (t_0,\, t_1) \,|\!)_{\pi,\,\theta\times\theta'} & = & \texttt{Pair}\ ((\!|\, t_0 \,|\!)_{\pi,\,\theta},\ (\!|\, t_1 \,|\!)_{\pi,\,\theta'}) \\
(\!|\, \mathit{fst}\ t \,|\!)_{\pi,\,\theta} & = & \texttt{Push Fst} \triangleright (\!|\, t \,|\!)_{\pi,\,\theta\times\theta'} \\
(\!|\, \mathit{snd}\ t \,|\!)_{\pi,\,\theta'} & = & \texttt{Push Snd} \triangleright (\!|\, t \,|\!)_{\pi,\,\theta\times\theta'} \\
(\!|\, \mathit{ifz}\ t\ .\ t' \,|\!)_{\pi,\,\theta} & = & \texttt{Push}\ (\!|\, t \,|\!)_{\pi,\,\mathbf{int}} \triangleright (\!|\, t' \,|\!)_{\pi,\,\theta\times\theta}
\end{array}
$$

En la compilación del operador n-ario se debe observar que el orden de los argumentos aparece invertido, esto se hace para lograr que la ejecución de cada una de las instrucciones `Push` termine por insertar en la pila cada clausura en el

orden correcto. La instrucción `Frame` que se agrega al final se encarga de crear en el tope de la pila el frame que contiene el código de los argumentos.

6.5 Relaciones de aproximación

En el Capítulo 5 definimos el conjunto de *realizadores* de tipos, que básicamente está compuesto por clausuras que interpretan, en cierto sentido operacional, a los tipos del lenguaje fuente. Por ejemplo, la clausura $(\texttt{Grab} \triangleright i, \eta)$ *realiza* un tipo funcional $\theta \to \theta'$ si para cualquier clausura α que realiza a θ se tiene que $(i, \alpha :: \eta)$ realiza a θ'. Se puede aplicar una idea similar para capturar la noción de corrección del compilador: el código generado por la compilación de un término se considera correcto cuando interpreta operacionalmente a la semántica del mismo. Para describir rigurosamente esta noción de corrección se definirán relaciones lógicas entre las clausuras y los elementos del dominio semántico del término involucrado. Específicamente, tendremos dos relaciones lógicas, la primera llamada "aproximación denotacional" que describe cómo los elementos del dominio aproximan a las clausuras, y la segunda, llamada "aproximación operacional" describe cómo las clausuras aproximan a los elementos del dominio.

$$\begin{array}{lcll} \blacktriangleright^{\theta} & \subseteq & \mathit{MClos} \times [\![\, \theta \,]\!] & \text{(Aproximación denotacional)} \\ \blacktriangleleft^{\theta}_{r} & \subseteq & \mathit{MClos} \times [\![\, \theta \,]\!] & \text{(Aproximación operacional)} \end{array}$$

El orden $\sqsubseteq$ del dominio $[\![\, \theta \,]\!]$ determina, de alguna forma, la precisión de la aproximación (si $\alpha \blacktriangleright^{\theta} d$ y $d \sqsupseteq d'$, entonces d' es una aproximación "menos precisa" de α con respecto a d). Operacionalmente, sin embargo, la precisión estará determinada por el índice $r \in \mathbb{N}$, un mayor índice indica mayor precisión. Esquemáticamente, tendremos la siguiente situación:

$$\begin{array}{l} \alpha \blacktriangleright^{\theta} d \sqsupseteq d' \sqsupseteq \ldots \sqsupseteq \bot \\ \blacktriangleleft^{\theta}_{r} \subseteq \blacktriangleleft^{\theta}_{r-1} \subseteq \ldots \subseteq \blacktriangleleft^{\theta}_{0} \ . \end{array}$$

(Esto muestra que $\blacktriangleleft^{\theta}_{r}$ es una relación indexada decreciente, cómo veremos en la Sección 6.5). A continuación comenzamos con la definición formal de estas relaciones.

Aproximación denotacional

Para definir la relación de aproximación denotacional aplicaremos un esquema de realizabilidad similar al del Capítulo 5. Esta vez no tendremos realizadores *de tipos*, sino que serán realizadores de objetos denotacionales (elementos

de un dominio). Esto es, para cada $d \in [\![\, \theta \,]\!]$, definiremos un conjunto $\mathcal{R}^{\theta}(d)$ de clausuras (llamadas realizadores de d con tipo θ).

Asumiremos un conjunto $\perp\!\!\!\perp \subseteq \mathit{Conf}$ de observaciones cerrado por anti-ejecución; además requerimos la existencia de una clausura "excluida", esto es, $\alpha \in \mathit{MClos}$ tal que $(\alpha, s) \notin \perp\!\!\!\perp$ para toda $s \in \mathit{Stack}$. Es fácil ver que $\perp\!\!\!\perp$ puede ser el conjunto de configuraciones que terminan (alcanzan una configuración bloqueante) o también las que divergen. En el primer caso, la clausura excluida puede ser, por ejemplo, $(i, [])$ donde $i = (\!|\, \mathit{rec}\ \lambda\, \bar{0}\ |\!)_{[],\,\mathbf{int}}$. En el segundo caso, la clausura excluida puede ser $(\texttt{Access 0}, [])$, que combinado con cualquier pila da lugar a una configuración bloqueante. Los realizadores se definen usando biortogonalidad. El conjunto $\mathcal{R}^{\theta}(d)$ se define en términos del conjunto $\mathcal{R}^{\theta}_{P}(d)$ (realizadores primitivos) usando el operador de clausura.

Definición 102 (Realizadores).

$$
\begin{array}{rcl}
\mathcal{R}^{\theta}_{P}(\bot) & = & \mathit{MClos} \\
\mathcal{R}^{\mathbf{int}}_{P}(\iota_{\uparrow}\, k) & = & \{\, (\texttt{Const}\ k,\ \eta) \mid \eta \in \mathit{MEnv} \,\} \\
\mathcal{R}^{\theta \to \theta'}_{P}(f) & = & \{\, (\texttt{Grab} \triangleright i,\ \eta) \mid \textit{para todo } \alpha \in \mathcal{R}^{\theta}(d), \\
 & & \textit{se cumple } (i,\ \alpha :: \eta) \in \mathcal{R}^{\theta'}(f\, d) \,\} \\
\mathcal{R}^{\theta \times \theta'}_{P}((d_0,\ d_1)) & = & \{\, (\texttt{Pair}\ (i_0, i_1),\ \eta) \mid (i_0,\ \eta) \in \mathcal{R}^{\theta}(d_0), \\
 & & (i_1,\ \eta) \in \mathcal{R}^{\theta'}(d_1) \,\}
\end{array}
$$

Los operadores ortogonales están definidos de la misma manera que en la Definición 67. El conjunto de tests para $d \in [\![\, \theta \,]\!]$ con tipo θ se define $\mathcal{T}^{\theta}(d) = \mathcal{R}^{\theta}(d)^{\perp}$, pero como antes, se puede demostrar que $\mathcal{T}^{\theta}(d) = \mathcal{R}^{\theta}_{P}(d)^{\perp}$ (ver Lema 22). La relación de aproximación denotacional se define directamente en términos del conjunto de realizadores.

Definición 103 (Aproximación denotacional).

$$\alpha \blacktriangleright^{\theta} d \qquad \textit{si y sólo si} \qquad \alpha \in \mathcal{R}^{\theta}(d)\ .$$

Aquí $\alpha \blacktriangleright^{\theta} d$ se lee "d aproxima a α con tipo θ". Por la extensividad del operador de clausura, se tiene $\mathcal{R}^{\theta}_{P}(d) \subseteq \mathcal{R}^{\theta}(d)$, por lo tanto, si $\alpha \in \mathcal{R}^{\theta}_{P}(d)$ tenemos que $\alpha \blacktriangleright^{\theta} d$. Con la primera igualdad $\mathcal{R}^{\theta}_{P}(\bot) = \mathit{MClos}$ establecemos que $\bot$ aproxima cualquier clausura (aunque esa aproximación resulte, intuitivamente, poco "precisa"). La constante entera $\iota_{\uparrow}\, k$ aproxima a $(\texttt{Const}\ k,\ \eta)$ donde $\eta \in \mathit{MEnv}$ es cualquier entorno de máquina. La función $f \in [\![\, \theta \to \theta' \,]\!]$ aproxima a $(\texttt{Grab} \triangleright i,\ \eta)$ si para cualquier argumento $d \in [\![\, \theta \,]\!]$, cuando d aproxima a α, se cumple que la aplicación $f\, d \in [\![\, \theta' \,]\!]$ aproxima a $(i,\ \alpha :: \eta)$. Por otro lado, el par $(d_0,\ d_1)$ aproxima a la clausura $(\texttt{Pair}\ (i_0, i_1),\ \eta)$ si d_0 aproxima a $(i_0,\ \eta)$ y d_1 aproxima a $(i_1,\ \eta)$.

Por supuesto, al tomar la extensión usando el operador de clausura, existen más aproximaciones por cada tipo que las determinadas por los realizadores primitivos.

El hecho de que $\alpha \blacktriangleright^{\theta} \bot$ para cualquier clausura α se sigue de la extensividad del operador de clausura:

Lema 34. *Para todo $\alpha \in MClos$, $\alpha \blacktriangleright^{\theta} \bot$ (es decir, $\mathcal{R}^{\theta}(\bot) = MClos$).*

Prueba. Como $\mathcal{R}^{\theta}_{P}(\bot) = MClos$ por definición, y $\mathcal{R}^{\theta}_{P}(\bot) \subseteq \mathcal{R}^{\theta}(\bot)$, se tiene $\mathcal{R}^{\theta}(\bot) = MClos$. □

De la existencia de la clausura excluida en el conjunto de observaciones se sigue que el conjunto de tests para $\bot$ con cualquier tipo θ es siempre igual al conjunto vacío.

Lema 35. $\mathcal{T}^{\theta}(\bot) = \emptyset$.

Prueba. Como $\mathcal{R}^{\theta}_{P}(\bot) = MClos$, entonces $\mathcal{T}^{\theta}(\bot) = MClos^{\bot}$. Luego, si vale que $s \in \mathcal{T}^{\theta}(\bot)$, se debe cumplir $(\alpha,\ s) \in \perp\!\!\!\perp$ para cualquier α, pero eso es imposible ya que si $\widehat{\alpha}$ es una clausura excluida, se tiene $(\widehat{\alpha},\ s) \notin \perp\!\!\!\perp$. Luego, concluimos $\mathcal{T}^{\theta}(\bot) = \emptyset$. □

Recordando la idea intuitiva de que $\sqsubseteq$ debe indicar la exactitud de la aproximación, debemos demostrar que cuando $d \in [\![\, \theta \,]\!]$ aproxima a una clausura α, y además $d' \sqsubseteq d$, entonces d' es también una aproximación de α. Primero demostraremos que esa propiedad es también válida sobre los realizadores primitivos.

Lema 36. *Si $d' \sqsubseteq d$ entonces $\mathcal{R}^{\theta}_{P}(d) \subseteq \mathcal{R}^{\theta}_{P}(d')$.*

Prueba. Se procede por inducción en el tipo θ. En la prueba usaremos que el operador de clausura es monótona respecto a la inclusión, y por lo tanto se tiene que $\mathcal{R}^{\theta}_{P}(d) \subseteq \mathcal{R}^{\theta}_{P}(d')$ implica $\mathcal{R}^{\theta}(d) \subseteq \mathcal{R}^{\theta}(d')$.

- Caso **int**: Notemos que si $d = d'$ la inclusión es verdadera por reflexividad de la inclusión. Si $d = \bot$, se tiene $d' = \bot$. Si $d = \iota_{\uparrow}\, k$, puede darse $d = d'$ o $d' = \bot$ (por definición de $\sqsubseteq$ sobre $[\![\, \mathbf{int} \,]\!]$). En el caso $d' = \bot$ se tiene $\mathcal{R}^{\theta}_{P}(d') = MClos$ y la inclusión es trivialmente verdadera.

- Caso $\theta \to \theta'$: Supongamos $(\texttt{Grab} \triangleright i,\ \eta) \in \mathcal{R}^{\theta \to \theta'}_{P}(f)$ y $g \sqsubseteq f$, demostremos $(\texttt{Grab} \triangleright i,\ \eta) \in \mathcal{R}^{\theta \to \theta'}_{P}(g)$. Sea $\alpha \in \mathcal{R}^{\theta}(d)$, por definición de $\mathcal{R}^{\theta \to \theta'}_{P}(f)$ tenemos $(i,\ \alpha :: \eta) \in \mathcal{R}^{\theta'}(f\,d)$. Por definición de $\sqsubseteq$ tenemos $g\,d \sqsubseteq f\,d$. Por hipótesis inductiva (y la monotonía del operador de clausura) se tiene $\mathcal{R}^{\theta'}(f\,d) \subseteq \mathcal{R}^{\theta'}(g\,d)$. Luego, $(i,\ \alpha :: \eta) \in \mathcal{R}^{\theta'}(g\,d)$. Por lo tanto, $(\texttt{Grab} \triangleright i,\ \eta) \in \mathcal{R}^{\theta \to \theta'}_{P}(g)$.

- Caso $\theta \times \theta'$. Supongamos $(\texttt{Pair}\ (i_0, i_1), \eta) \in \mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$ y tomemos $(d_0', d_1') \sqsubseteq (d_0, d_1)$. Sabemos $(i_0, \eta) \in \mathcal{R}_P^{\theta}(d_0)$ y $(i_1, \eta) \in \mathcal{R}_P^{\theta'}(d_1)$ por definición de $\mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$. Por hipótesis inductiva vale la inclusión $\mathcal{R}^{\theta}(d_0) \subseteq \mathcal{R}^{\theta}(d_0')$, luego $(i_0, \eta) \in \mathcal{R}_P^{\theta}(d_0')$. De la misma forma obtenemos $(i_1, \eta) \in \mathcal{R}_P^{\theta}(d_1')$, y por lo tanto $(\texttt{Pair}\ (i_0, i_1), \eta) \in \mathcal{R}_P^{\theta \times \theta'}((d_0', d_1'))$.

□

Del resultado anterior se desprende la propiedad que queríamos demostrar para la aproximación denotacional.

Lema 37. *Si* $\alpha \blacktriangleright^{\theta} d$ *y* $d' \sqsubseteq d$ *entonces* $\alpha \blacktriangleright^{\theta} d'$.

Prueba. Consecuencia directa del Lema 36 y la monotonía del operador de clausura. □

Si conocemos que $\alpha \blacktriangleright^{\theta} d$, y queremos demostrar que $(\alpha, s) \in \perp\!\!\!\perp$, una forma de hacerlo es probar que $s \in \mathcal{T}^{\theta}(d)$. Para ello es útil contar con resultados que permitan combinar tanto tests como realizadores para construir otros tests de tipo θ. El Lema 38 permite obtener tests para tipos funcionales, los Lemas 39 y 41 para tipos de productos, y los Lemas 40 y 42 para el tipo **int**. Presentamos esos resultados a continuación.

Lema 38. *Si* $\alpha \blacktriangleright^{\theta} d$ *y* $s \in \mathcal{T}^{\theta'}(f\,d)$ *entonces* $\alpha :: s \in \mathcal{T}^{\theta \to \theta'}(f)$.

Prueba. Probaremos que para todo $\hat{\alpha} \in \mathcal{R}_P^{\theta \to \theta'}(f)$, se cumple $(\hat{\alpha}, \alpha :: s) \in \perp\!\!\!\perp$. Notemos que $\mathcal{T}^{\theta'}(f\,d) \neq \emptyset$ por hipótesis, luego $f\,d \neq \bot$, y por lo tanto $f \neq \bot$. Por definición de $\mathcal{R}_P^{\theta \to \theta'}(f)$, y como $f \neq \bot$, se tiene $\hat{\alpha} = (\texttt{Grab} \triangleright i, \eta)$ para algún $i \in \mathit{Code}$, $\eta \in \mathit{MEnv}$. Más aún, como $\alpha \blacktriangleright^{\theta} d$, se tiene $(i, \alpha :: \eta) \blacktriangleright^{\theta'} f\,d$. Como $(\hat{\alpha}, \alpha :: s) \longmapsto (i, \alpha :: \eta, s) \in \perp\!\!\!\perp$, concluimos $(\hat{\alpha}, \alpha :: s) \in \perp\!\!\!\perp$ por anti-ejecución. □

Lema 39. *Si* $s \in \mathcal{T}^{\theta}(d_0)$ *entonces* $(\texttt{Fst}, \eta) :: s \in \mathcal{T}^{\theta \times \theta'}((d_0, d_1))$. *Similarmente, si* $s \in \mathcal{T}^{\theta'}(d_1)$ *entonces* $(\texttt{Snd}, \eta) :: s \in \mathcal{T}^{\theta \times \theta'}((d_0, d_1))$.

Prueba. Notemos que $d_0 \neq \bot$ pues $\mathcal{T}^{\theta}(d_0) \neq \emptyset$ por hipótesis; luego tenemos $(d_0, d_1) \neq \bot$. Sea $\hat{\alpha} \in \mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$, queremos ver $(\hat{\alpha}, (\texttt{Fst}, \eta) :: s) \in \perp\!\!\!\perp$. Por definición de $\mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$, se tiene que $\hat{\alpha} = (\texttt{Pair}\ (i_0, i_1), \eta')$ donde $i_0, i_1 \in \mathit{Code}$, $\eta' \in \mathit{MEnv}$. Más aún, se tiene $(i_0, \eta') \blacktriangleright^{\theta} d_0$ y $(i_1, \eta') \blacktriangleright^{\theta'} d_1$. Por lo tanto, como

$$\begin{array}{llll} (\texttt{Pair}\ (i_0, i_1), & \eta', & (\texttt{Fst}, \eta) :: s) & \longmapsto \\ (\texttt{Fst}, & \eta, & \langle (i_0, \eta'), (i_1, \eta') \rangle :: s) & \longmapsto \\ (i_0, & \eta', & s) & \in \perp\!\!\!\perp\ , \end{array}$$

se tiene $(\texttt{Pair}\ (i_0, i_1),\ \eta',\ (\texttt{Fst},\ \eta) :: s) \in \perp\!\!\!\perp$ por anti-ejecución. □

Lema 40. *Suponiendo que* $\alpha_0 \blacktriangleright^\theta d_0$, $\alpha_1 \blacktriangleright^\theta d_1$ *y* $s \in \mathcal{T}^\theta(IFZ_\theta\ d\ (d_0,\ d_1))$, *entonces se cumple* $\langle \alpha_0,\ \alpha_1 \rangle :: s \in \mathcal{T}^{\mathbf{int}}(d)$.

Prueba. Sea $d' = IFZ_\theta\ d\ (d_0,\ d_1)$. Por hipótesis sabemos $\mathcal{T}^\theta(d')$ no es vacío, por lo tanto $d' \neq \bot$. Luego, $d \neq \bot$ y en consecuencia $d = \iota_\uparrow k$ con $k \in \mathbb{Z}$. Para demostrar $\langle \alpha_0,\ \alpha_1 \rangle :: s \in \mathcal{T}^{\mathbf{int}}(\iota_\uparrow k)$ debemos tomar $(\texttt{Const}\ k,\ \eta) \in \mathcal{R}_P^{\mathbf{int}}(\iota_\uparrow k)$ y demostrar $(\texttt{Const}\ k,\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \in \perp\!\!\!\perp$.

- Si $k = 0$. Tenemos $d' = d_0$ y $(\texttt{Const}\ k,\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \longmapsto (\alpha_0,\ s) \in \perp\!\!\!\perp$,
- Si $k \neq 0$, Tenemos $d' = d_1$ y $(\texttt{Const}\ k,\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \longmapsto (\alpha_1,\ s) \in \perp\!\!\!\perp$.

En ambos casos se obtiene $(\texttt{Const}\ k,\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \in \perp\!\!\!\perp$ como queríamos. □

Lema 41. *Si* $\alpha \blacktriangleright^{\mathbf{int}} d$ *y* $s \in \mathcal{T}^\theta(IFZ_\theta\ d\ (d_0,\ d_1))$, *entonces* $\alpha :: s \in \mathcal{T}^{\theta \times \theta}((d_0,\ d_1))$.

Prueba. Sea $\hat{\alpha} \in \mathcal{R}_P^{\theta \times \theta}((d_0,\ d_1))$, debemos demostrar $(\hat{\alpha},\ \alpha :: s) \in \perp\!\!\!\perp$. Definimos $d' = IFZ_\theta\ d\ (d_0,\ d_1)$; por hipótesis sabemos $\mathcal{T}^\theta(d')$ no es vacío, por lo tanto $d' \neq \bot$. Luego $d_0 \neq \bot$, $d_1 \neq \bot$ y en consecuencia $(d_0,\ d_1) \neq \bot$. Por lo tanto $\hat{\alpha}$ tiene la forma $(\texttt{Pair}\ (i_0, i_1),\ \eta)$ con $i_0,\ i_1 \in \mathit{Code}$, $\eta \in \mathit{MEnv}$. Por definición de $\mathcal{R}_P^{\theta \times \theta}((d_0,\ d_1))$, se tiene $(i_0,\ \eta) \blacktriangleright^\theta d_0$ y $(i_1,\ \eta) \blacktriangleright^\theta d_1$. Por Lema 40 obtenemos $\langle \alpha_0,\ \alpha_1 \rangle :: s \in \mathcal{T}^{\mathbf{int}}(d)$, donde $\alpha_0 = (i_0,\ \eta)$ y $\alpha_1 = (i_1,\ \eta)$. Luego, como $(\texttt{Pair}\ (i_0, i_1),\ \eta,\ \alpha :: s) \longmapsto (\alpha,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \in \perp\!\!\!\perp$, se obtiene $(\hat{\alpha},\ \alpha :: s) \in \perp\!\!\!\perp$ como queríamos demostrar. □

Lema 42. *Supongamos* $\ominus^n$ *es un operador aritmético de aridad n. Entonces para todo* $\bar{k},\ \bar{\alpha},\ \bar{d}$ *tales que*

$$(1)\ |\bar{k}| + |\bar{\alpha}| = n - 1, \quad (2)\ |\bar{\alpha}| = |\bar{d}|, \quad (3)\ \alpha_r \blacktriangleright^{\mathbf{int}} d_r \text{ para todo } r < |\bar{\alpha}|,$$

y para todo $d' \in [\![\, \mathbf{int} \,]\!]$, $s \in \mathcal{T}^{\mathbf{int}}((\ominus^n \bar{k})_\perp\ (d',\ \bar{d}))$ *se cumple*

$$\{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s \in \mathcal{T}^{\mathbf{int}}(d')\ .$$

Prueba. Para mostrar que $\{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s \in \mathcal{T}^{\mathbf{int}}(d')$ tomamos $\hat{\alpha} \in \mathcal{R}_P^{\mathbf{int}}(d')$ y demostramos que $(\hat{\alpha},\ \{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s) \in \perp\!\!\!\perp$. Sea $d'' = (\ominus^n \bar{k})_\perp\ (d',\ \bar{d})$; dado que $\mathcal{T}^{\mathbf{int}}(d'')$ es no vacío, tenemos $d'' \neq \bot$ y en consecuencia $d' \neq \bot$. Por lo tanto, $d' = \iota_\uparrow k_0$ para alguna constante $k_0 \in \mathbb{Z}$, y $d'' = (\ominus^n (\bar{k},\ k_0))_\perp\ \bar{d}$. Además, $\hat{\alpha}$ tiene la forma $(\texttt{Const}\ k_0,\ \eta)$ para algún $\eta \in \mathit{MEnv}$. Veremos que se cumple $(\hat{\alpha},\ \{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s) \in \perp\!\!\!\perp$ por inducción en el tamaño del vector $\bar{\alpha}$.

En el caso $\bar{\alpha} = []$ obtenemos $\bar{d} = []$ y $|\bar{k}| = n - 1$. Por lo tanto se tiene $d'' = \iota_\uparrow k'$ donde $k' = \ominus^n (\bar{k},\ k_0)$. Es fácil ver que $(\texttt{Const}\ k',\ \eta) \blacktriangleright^{\mathbf{int}} d''$, y dado

que $(\hat{\alpha},\ \{\ominus^n \bar{k} \bullet\} :: s) \longmapsto (\texttt{Const}\ k',\ \eta,\ s) \in \perp\!\!\!\perp$ tenemos $(\hat{\alpha},\ \{\ominus^n \bar{k} \bullet\} :: s) \in \perp\!\!\!\perp$ como queríamos.

Ahora asumamos $\bar{\alpha} = (\alpha_0,\ \bar{\alpha_1})$. Tenemos $\bar{d} = (d_0,\ \bar{d_1})$ para algún $d_0 \in \mathbb{Z}_\perp$ tal que $\alpha_0 \blacktriangleright^{\text{int}} d_0$. Como $d'' = (\ominus^n (\bar{k},\ k_0))_\perp (d_0,\ \bar{d_1})$ podemos aplicar hipótesis inductiva para obtener $\{\ominus^n \bar{k},\ k_0 \bullet \bar{\alpha_1}\} :: s \in \mathcal{T}^{\text{int}}(d_0)$. Por lo tanto, dado que vale $(\hat{\alpha},\ \{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s) \longmapsto (\alpha_0,\ \{\ominus^n \bar{k},\ k_0 \bullet \bar{\alpha_1}\} :: s)$ y luego $(\hat{\alpha},\ \{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s) \in \perp\!\!\!\perp$ como queríamos demostrar. □

Definiremos una extensión de la relación de aproximación denotacional $\blacktriangleright^\pi \subseteq [\![\pi]\!] \times MEnv$, que describe cuándo un ambiente ρ es una aproximación de un entorno de máquina η, bajo un contexto π. Esta relación se define punto a punto: la n-ésima proyección del ambiente (esto es, $\rho \swarrow n$) aproxima a la clausura ubicada en la n-ésima posición del entorno η (es decir, $\eta \cdot n$) con el tipo $\pi \cdot n$. Esto se puede definir por inducción estructural en el contexto π, como se indica a continuación.

Definición 104 (Aproximación denotacional para entornos).

$$[]\ \blacktriangleright^{[]}\ \varnothing$$

$$\alpha :: \eta \blacktriangleright^{\theta :: \pi} (d,\ \rho) \qquad \text{si y sólo si} \qquad \alpha \blacktriangleright^\theta d \quad y \quad \eta \blacktriangleright^\pi \rho\ .$$

Si $\eta \blacktriangleright^\pi \rho$, entonces la proyección $(\texttt{Access}\ n,\ \eta)$ es una aproximación de la clausura $\rho \swarrow n$, como se demuestra a continuación.

Lema 43. *Si* $\eta \blacktriangleright^\pi \rho$ *y* $n < |\pi|$, *entonces* $(\texttt{Access}\ n,\ \eta) \blacktriangleright^{\pi.n} \rho \swarrow n$.

Prueba. Sea $\theta = \pi \cdot n$ y $s \in \mathcal{T}^\theta(\rho \swarrow n)$. Como $\eta \blacktriangleright^\pi \rho$ y $n < |\pi|$ se tiene $(\eta \cdot n,\ \eta) \blacktriangleright^\theta \rho \swarrow n$. Luego $(\texttt{Access}\ n,\ \eta,\ s) \longmapsto (\eta \cdot n,\ s) \in \perp\!\!\!\perp$. Por lo tanto $(\texttt{Access}\ n,\ \eta,\ s) \in \perp\!\!\!\perp$. Como esto vale para todo $s \in \mathcal{T}^\theta(\rho \swarrow n)$, se tiene $(\texttt{Access}\ n,\ \eta) \blacktriangleright^{\pi.n} \rho \swarrow n$. □

Supongamos que $\pi \vdash t : \theta$ es un juicio válido y que se tiene $\eta \blacktriangleright^\pi \rho$. Si se cumple $(i,\ \eta) \blacktriangleright^\theta [\![t]\!]_{\pi,\theta} \rho$, entonces por Lema 43 se puede demostrar que $(\texttt{Access}\ 0,\ (i,\ \eta) :: \eta) \blacktriangleright^\theta [\![t]\!]_{\pi,\theta} \rho$. De esa forma, hemos construimos una aproximación de la semántica del término t a partir de otra aproximación. Observemos que i no necesariamente es código generado por un compilador, lo que sugiere que la noción de aproximación puede usarse para analizar el comportamiento del código (con respecto a un objeto denotacional) sin hablar de un compilador específico.

En efecto, aplicar el Lema 43 es sólo una de varias maneras de construir nuevas aproximaciones a partir de otras, tal como indican los siguientes resultados.

Lema 44. *Si* $(i,\ \eta) \blacktriangleright^{\theta \to \theta'} f\ y\ (i',\ \eta) \blacktriangleright^\theta d$, *entonces* $(\texttt{Push}\ i' \triangleright i,\ \eta) \blacktriangleright^{\theta'} f d$.

Prueba. Sea $s \in \mathcal{T}^{\theta'}(f\,d)$. Tenemos $(i', \eta) :: s \in \mathcal{T}^{\theta \to \theta'}(f)$ por Lema 38. Luego $(\mathtt{Push}\ i' \triangleright i, \eta, s) \longmapsto (i, \eta, (i', \eta) :: s) \in \mathbb{1}$ y $(\mathtt{Push}\ i' \triangleright i, \eta, s) \in \mathbb{1}$. Como esto vale para todo $s \in \mathcal{T}^{\theta'}(f\,d)$, se tiene $(\mathtt{Push}\ i' \triangleright i, \eta) \blacktriangleright^{\theta'} f\,d$. □

Lema 45. *Si* $(i, \eta) \blacktriangleright^{\theta \to \theta} f\,y$ $(\mathtt{Fix} \triangleright i, \eta) \blacktriangleright^{\theta} d$, *entonces* $(\mathtt{Fix} \triangleright i, \eta) \blacktriangleright^{\theta} f\,d$.

Prueba. Sea $\alpha = (\mathtt{Fix} \triangleright i, \eta)$ y $s \in \mathcal{T}^{\theta}(f\,d)$. Por Lema 38 vale $\alpha :: s \in \mathcal{T}^{\theta \to \theta}(f)$. Luego $(\mathtt{Fix} \triangleright i, \eta, s) \longmapsto (i, \eta, \alpha :: s) \in \mathbb{1}$ y $(\mathtt{Fix} \triangleright i, \eta, s) \in \mathbb{1}$. Como esto vale para todo $s \in \mathcal{T}^{\theta}(f\,d)$, se obtiene $(\mathtt{Fix} \triangleright i, \eta) \blacktriangleright^{\theta} f\,d$. □

Lema 46. *Suponiendo que* $(i_r, \eta) \blacktriangleright^{\mathbf{int}} d_r$ *para todo* $r \in \{1, \ldots, n\}$, *entonces se cumple* $(\mathtt{Push}\ i_n \triangleright \ldots \triangleright \mathtt{Push}\ i_1 \triangleright \mathtt{Frame}\ \ominus^n, \eta) \blacktriangleright^{\mathbf{int}} \ominus^n_{\perp}\ (d_1, \ldots, d_n)$.

Prueba. Sea $s \in \mathcal{T}^{\mathbf{int}}(\ominus^n_{\perp}\,(d_1, \ldots, d_n))$, $\alpha_r = (i_r, \eta)$ con $r \in \{1, \ldots, n\}$ y sea $\bar{\alpha} = (\alpha_2, \ldots, \alpha_n)$. Por Lema 42 se tiene $\{\ominus^n \bullet \bar{\alpha}\} :: s \in \mathcal{T}^{\mathbf{int}}(d_1)$. Luego, como

$$\begin{array}{llll} (\mathtt{Push}\ i_n \triangleright \ldots \triangleright \mathtt{Push}\ i_1 \triangleright \mathtt{Frame}\ \ominus^n, & \eta, & s) & \longmapsto^* \\ (\mathtt{Frame}\ \ominus^n, & \eta, & \alpha_1 :: \alpha_2 :: \ldots :: \alpha_n :: s) & \longmapsto \\ (i_1, & \eta, & \{\ominus^n \bullet \bar{\alpha}\} :: s) & \in \mathbb{1}\ , \end{array}$$

se obtiene $(\mathtt{Push}\ i_n \triangleright \ldots \triangleright \mathtt{Push}\ i_1 \triangleright \mathtt{Frame}\ \ominus^n, \eta, s) \in \mathbb{1}$. □

Lema 47. *Si* $(i, \eta) \blacktriangleright^{\theta \times \theta'} (d_0, d_1)$, *entonces se tiene* $(\mathtt{Push}\ \mathtt{Fst} \triangleright i, \eta) \blacktriangleright^{\theta} d_0$ *y además* $(\mathtt{Push}\ \mathtt{Snd} \triangleright i, \eta) \blacktriangleright^{\theta'} d_1$.

Prueba. Sea $s \in \mathcal{T}^{\theta}(d_0)$. Se tiene $(\mathtt{Fst}, \eta) :: s \in \mathcal{T}^{\theta \times \theta'}((d_0, d_1))$ por Lema 39. Luego $(\mathtt{Push}\ \mathtt{Fst} \triangleright i, \eta, s) \longmapsto (i, \eta, (\mathtt{Fst}, \eta) :: s) \in \mathbb{1}$. Obtenemos entonces $(\mathtt{Push}\ \mathtt{Fst} \triangleright i, \eta, s) \in \mathbb{1}$ para toda $s \in \mathcal{T}^{\theta}(d_0)$. Por lo tanto concluimos $(\mathtt{Push}\ \mathtt{Fst} \triangleright i, \eta) \blacktriangleright^{\theta} d_0$. La prueba de $(\mathtt{Push}\ \mathtt{Snd} \triangleright i, \eta) \blacktriangleright^{\theta'} d_1$ es análoga. □

Lema 48. *Suponiendo que valen* $(i, \eta) \blacktriangleright^{\theta \times \theta} (d_0, d_1)$ *y* $(i', \eta) \blacktriangleright^{\mathbf{int}} d$, *entonces se cumple* $(\mathtt{Push}\ i' \triangleright i, \eta) \blacktriangleright^{\theta} IFZ_{\theta}\ d\ (d_0, d_1)$.

Prueba. Sea $s \in \mathcal{T}^{\theta}(IFZ_{\theta}\ d\ (d_0, d_1))$, entonces $(i', \eta) :: s \in \mathcal{T}^{\theta \times \theta}((d_0, d_1))$ aplicando el Lema 41. Como $(\mathtt{Push}\ i' \triangleright i, \eta, s) \longmapsto (i, \eta, (i', \eta) :: s) \in \mathbb{1}$ obtenemos $(\mathtt{Push}\ i' \triangleright i, \eta, s) \in \mathbb{1}$. Como esto es válido para toda $s \in \mathcal{T}^{\theta}(IFZ_{\theta}\ d\ (d_0, d_1))$, se obtiene $(\mathtt{Push}\ i' \triangleright i, \eta) \blacktriangleright^{\theta} IFZ_{\theta}\ d\ (d_0, d_1)$. □

Suponiendo $\pi \vdash t : \theta$, podríamos esperar que si $\eta \blacktriangleright^{\pi} \rho$ entonces $[\![\, t \,]\!]_{\pi,\theta}\,\rho$ es una aproximación de $((\!|\, t \,|\!)_{\pi,\theta}, \eta)$. Sin embargo, no se puede demostrar esa afirmación directamente, sino que tendremos que usar la cadena semántica (ver Definición 98). Demostraremos que cada miembro de la cadena semántica $[\![\, t \,]\!]^r_{\pi,\theta}\,\rho$ es una aproximación de $((\!|\, t \,|\!)_{\pi,\theta}, \eta)$. Por el Lema 33, para aquellas cadenas semánticas que contengan su propio límite (por ejemplo, todas las cadenas del

dominio $[\![\,\mathbf{int}\,]\!]$), es posible probar que $\bigsqcup_{r\in\mathbb{N}}^{\infty} [\![\,t\,]\!]^{r}_{\pi,\,\theta}\rho = [\![\,t\,]\!]_{\pi,\,\theta}\rho$ aproxima a $(\!|\,t\,|\!)_{\pi,\,\theta},\ \eta)$, como queríamos originalmente.

El siguiente resultado relaciona la compilación de un término con todos los miembros de su cadena semántica.

Teorema 9. *Si* $\eta \blacktriangleright^{\pi} \rho$ *y* $\pi \vdash t : \theta$, *entonces* $((\!|\,t\,|\!)_{\pi,\,\theta},\ \eta) \blacktriangleright^{\theta} [\![\,t\,]\!]^{r}_{\pi,\,\theta}\rho$ *para todo índice* $r \in \mathbb{N}$.

Prueba. Se procede por inducción en la derivación de $\pi \vdash t : \theta$. A su vez, en cada caso, se realiza una inducción en el índice r. Cuando $r = 0$ la prueba es directa por Lema 34, puesto que $[\![\,t\,]\!]^{0}_{\pi,\,\theta}\rho = \bot$. Nos enfocamos en el caso $r = r' + 1$.

- Caso (Abs). Sea $f = [\![\,\lambda\, t\,]\!]^{r'+1}_{\pi,\,\theta\to\theta'}\rho$, debemos ver $((\!|\,\lambda\, t\,|\!)_{\pi,\,\theta\to\theta'},\ \eta) \blacktriangleright^{\theta\to\theta'} f$. Mostraremos que $(\mathtt{Grab} \triangleright (\!|\,t\,|\!)_{\theta::\pi,\,\theta'},\ \eta) = ((\!|\,\lambda\, t\,|\!)_{\pi,\,\theta\to\theta'},\ \eta) \in \mathcal{R}_P^{\theta\to\theta'}(f)$. Para ello tomemos $\alpha \in \mathcal{R}^{\theta}(d)$. Como $\eta \blacktriangleright^{\pi} \rho$, obtenemos $\alpha :: \eta \blacktriangleright^{\theta::\pi} (d,\ \rho)$. Luego, por hipótesis inductiva, tenemos

  $$(\!|\,t\,|\!)_{\theta::\pi,\,\theta'},\ \alpha :: \eta) \blacktriangleright^{\theta'} [\![\,t\,]\!]^{r'}_{\theta::\pi,\,\theta'}\,(d,\ \rho) = [\![\,\lambda\, t\,]\!]^{r'+1}_{\pi,\,\theta\to\theta'}\,\rho\, d = f\, d\ .$$

  Como esto vale para todo $\alpha \in \mathcal{R}^{\theta}(d)$, por definición de $\mathcal{R}_P^{\theta\to\theta'}(f)$ se obtiene $(\mathtt{Grab} \triangleright (\!|\,t\,|\!)_{\theta::\pi,\,\theta'},\ \eta) \in \mathcal{R}_P^{\theta\to\theta'}(f) \subseteq \mathcal{R}^{\theta\to\theta'}(f)$.

- Caso (Var). Debemos demostrar

  $$((\!|\,\bar{n}\,|\!)_{\pi,\,\theta},\ \eta) = (\mathtt{Access}\ n,\ \eta) \blacktriangleright^{\theta} [\![\,\bar{n}\,]\!]^{r'+1}_{\pi,\,\theta}\,\rho = \rho \swarrow n\ .$$

  Como $\pi \cdot n = \theta$, se obtiene $(\mathtt{Access}\ n,\ \eta) \blacktriangleright^{\theta} [\![\,\bar{n}\,]\!]^{r'+1}_{\pi,\,\theta}\,\rho$ directamente del Lema 43.

- Caso (App). Queremos ver $((\!|\,t\ t'\,|\!)_{\pi,\,\theta'},\ \eta) \blacktriangleright^{\theta'} [\![\,t\ t'\,]\!]^{r'+1}_{\pi,\,\theta'}\,\rho$. Por hipótesis inductiva, $((\!|\,t\,|\!)_{\pi,\,\theta\to\theta'},\ \eta) \blacktriangleright^{\theta'} [\![\,t\,]\!]^{r'}_{\pi,\,\theta\to\theta'}\,\rho$ y $((\!|\,t'\,|\!)_{\pi,\,\theta},\ \eta) \blacktriangleright^{\theta} [\![\,t'\,]\!]^{r'}_{\pi,\,\theta}\rho$. Luego, por Lema 44, concluimos

  $$\begin{array}{lll} (\mathtt{Push}\ (\!|\,t'\,|\!)_{\pi,\,\theta} \triangleright (\!|\,t\,|\!)_{\pi,\,\theta\to\theta'},\ \eta) & \blacktriangleright^{\theta'} & ([\![\,t\,]\!]^{r'}_{\pi,\,\theta\to\theta'}\,\rho)\ ([\![\,t'\,]\!]^{r'}_{\pi,\,\theta}\rho) \\ & = & [\![\,t\ t'\,]\!]^{r'+1}_{\pi,\,\theta'}\,\rho\ . \end{array}$$

- Caso (Rec). Sea $\alpha = ((\!|\,rec\ t\,|\!)_{\pi,\,\theta},\ \eta)$, $d_u = [\![\,rec\ t\,]\!]^{u}_{\pi,\,\theta}\rho$, queremos demostrar $\alpha \blacktriangleright^{\theta} d_{r'+1}$. Como hemos realizado inducción sobre r, podemos suponer $\alpha \blacktriangleright^{\theta} d_{r'}$. Por hipótesis inductiva, $((\!|\,t\,|\!)_{\pi,\,\theta\to\theta},\ \eta) \blacktriangleright^{\theta\to\theta} [\![\,t\,]\!]^{r'}_{\pi,\,\theta\to\theta}\rho$. Luego, por Lema 45, concluimos

  $$\alpha = (\mathtt{Fix} \triangleright (\!|\,t\,|\!)_{\pi,\,\theta\to\theta'},\ \eta) \blacktriangleright^{\theta} ([\![\,t\,]\!]^{r'}_{\pi,\,\theta\to\theta}\rho)\ d_{r'} = [\![\,rec\ t\,]\!]^{r'+1}_{\pi,\,\theta}\,\rho = d_{r'+1}\ .$$

- Caso (Const). Como $(\texttt{Const}\ k,\ \eta) \in \mathcal{R}_P^{\mathbf{int}}(\iota_\uparrow k)$, se tiene

$$(\!|\ \underline{k}\ |\!)_{\pi,\,\mathbf{int}},\ \eta) = (\texttt{Const}\ k,\ \eta) \blacktriangleright^{\mathbf{int}} \iota_\uparrow k = [\![\ \underline{k}\ ]\!]^{r'+1}_{\pi,\,\mathbf{int}}\,\rho\ .$$

- Caso (Op). Sea $d_u = [\![\ t_u\ ]\!]^{r'}_{\pi,\,\mathbf{int}}\,\rho$ y $i_u = (\!|\ t_u\ |\!)_{\pi,\,\mathbf{int}}$ con $u \in \{1,\ \ldots\ , n\}$, queremos ver $((\!|\ \ominus^n\ (t_1, \ldots, t_n)\ |\!)_{\pi,\,\mathbf{int}},\ \eta) \blacktriangleright^{\mathbf{int}} \ominus^n_\perp\ (d_1,\ \ldots\ , d_n)$. Por hipótesis inductiva tenemos $(i_u,\ \eta) = ((\!|\ t_u\ |\!)_{\pi,\,\mathbf{int}},\ \eta) \blacktriangleright^{\mathbf{int}} d_u$. Luego, por Lema 46, obtenemos

$$\begin{array}{ll} ((\!|\ \ominus^n\ (t_1, \ldots, t_n)\ |\!)_{\pi,\,\mathbf{int}},\ \eta) & = \\ (\texttt{Push}\ i_n \triangleright \ldots \triangleright \texttt{Push}\ i_1 \triangleright \texttt{Frame}\ \ominus^n,\ \eta) & \blacktriangleright^{\mathbf{int}} \\ \ominus^n_\perp\ (d_1,\ \ldots\ , d_n) & = [\![\ \ominus^n\ (t_1, \ldots, t_n)\ ]\!]^{r'+1}_{\pi,\,\mathbf{int}}\,\rho\ . \end{array}$$

- Caso (Pair). Para abreviar definamos $d_0 = [\![\ t_0\ ]\!]^{r'}_{\pi,\,\theta}\rho$, $d_1 = [\![\ t_1\ ]\!]^{r'}_{\pi,\,\theta'}\,\rho$ y además $i_0 = (\!|\ t_0\ |\!)_{\pi,\,\theta}$ y $i_1 = (\!|\ t_1\ |\!)_{\pi,\,\theta'}$. Se tiene $(i_0,\ \eta) \blacktriangleright^{\theta} d_0$ y $(i_1,\ \eta) \blacktriangleright^{\theta'} d_1$ por hipótesis inductiva, luego concluimos

$$(\!|\ (t_0,\ t_1)\ |\!)_{\pi,\,\theta\,\times\,\theta'},\ \eta) = (\texttt{Pair}\ (i_0, i_1),\ \eta) \in \mathcal{R}_P^{\theta\,\times\,\theta'}((d_0,\ d_1))\ .$$

  Como $(d_0,\ d_1) = [\![\ (t_0,\ t_1)\ ]\!]^{r'+1}_{\pi,\,\theta\,\times\,\theta'}\,\rho$, queda demostrado este caso.

- Caso (Fst). Sea $(d_0,\ d_1) = [\![\ t\ ]\!]^{r'}_{\pi,\,\theta\,\times\,\theta'}\,\rho$, por hipótesis inductiva tenemos $(\!|\ t\ |\!)_{\pi,\,\theta\,\times\,\theta'},\ \eta) \blacktriangleright^{\theta\,\times\,\theta'} (d_0,\ d_1)$, por lo tanto, por Lema 47, obtenemos

$$(\!|\ fst\ t\ |\!)_{\pi,\,\theta},\ \eta) = (\texttt{Push Fst} \triangleright (\!|\ t\ |\!)_{\pi,\,\theta\,\times\,\theta'},\ \eta) \blacktriangleright^{\theta} d_0 = [\![fst\ t\ ]\!]^{r'+1}_{\pi,\,\theta}\,\rho\ .$$

- Caso (Snd). Similar al anterior.

- Caso (Cond). Debemos ver $((\!|\ ifz\ t\ .\ t'\ |\!)_{\pi,\,\theta},\ \eta) \blacktriangleright^{\theta} IFZ_\theta\ d\ (d_0,\ d_1)$, donde $d = [\![\ t\ ]\!]^{r'}_{\pi,\,\mathbf{int}}$ y $(d_0,\ d_1) = [\![\ t'\ ]\!]^{r'}_{\pi,\,\theta\,\times\,\theta}\rho$. Por hipótesis inductiva, tenemos $(\!|\ t\ |\!)_{\pi,\,\mathbf{int}},\ \eta) \blacktriangleright^{\mathbf{int}} d$ y $((\!|\ t'\ |\!)_{\pi,\,\theta\,\times\,\theta},\ \eta) \blacktriangleright^{\theta\,\times\,\theta} (d_0,\ d_1)$. Luego, por Lema 48, obtenemos

$$(\!|\ ifz\ t\ .\ t'\ |\!)_{\pi,\,\theta},\ \eta) = (\texttt{Push}\ (\!|\ t\ |\!)_{\pi,\,\mathbf{int}} \triangleright (\!|\ t'\ |\!)_{\pi,\,\theta\,\times\,\theta},\ \eta) \blacktriangleright^{\theta} IFZ_\theta\ d\ (d_0,\ d_1)\,.$$

□

A partir del hecho de que el Teorema 9 es válido para cualquier conjunto de observaciones $\perp\!\!\!\perp$ (que cumpla las propiedades requeridas) se desprende el

teorema de corrección del compilador para términos cerrados de tipo **int** cuya semántica denotacional es una constante entera.

Veremos que en la prueba de ese teorema se elige una instancia concreta de $\perp\!\!\!\perp$ para lograr demostrar la propiedad deseada. La demostración también depende del hecho de que todas las cadenas del dominio $[\![\,\mathbf{int}\,]\!]$ contienen a su propio límite, como es el caso en todo dominio llano.

Teorema 10. *Supongamos* $[] \vdash t : \mathbf{int}$, *y además* $[\![\, t \,]\!]_{[],\,\mathbf{int}}\, \varnothing = \iota_{\uparrow}\, k$, *entonces existe* $\eta \in \mathit{MEnv}$ *tal que* $(\!|\, t \,|\!)_{[],\,\mathbf{int}},\ [],\ []) \longmapsto^{*} (\mathtt{Const}\ k,\ \eta,\ [])$.

Prueba. Supongamos $[\![\, t \,]\!]_{[],\,\mathbf{int}}\, \varnothing = \iota_{\uparrow}\, k$. Definimos

$$\perp\!\!\!\perp = \{\, w \mid w \longmapsto^{*} (\mathtt{Const}\ k,\ \eta,\ [])\ \text{ con } \eta \in \mathit{MEnv} \,\}\ .$$

Es claro que $\perp\!\!\!\perp$ es cerrado por anti-ejecución y tiene clausura excluida. Por Teorema 9 se tiene

$$(\!|\, t \,|\!)_{[],\,\mathbf{int}},\ [])\ \blacktriangleright^{\mathbf{int}}\ [\![\, t \,]\!]^{r}_{[],\,\mathbf{int}}\, \varnothing\ ,$$

para todo $r \in \mathbb{N}$.

Por ser $[\![\,\mathbf{int}\,]\!]$ un dominio llano, existe $r' \in \mathbb{N}$ tal que

$$[\![\, t \,]\!]^{r'}_{[],\,\mathbf{int}}\, \varnothing = \bigsqcup_{r \in \mathbb{N}}^{\infty} ([\![\, t \,]\!]^{r}_{[],\,\mathbf{int}}\, \varnothing) = [\![\, t \,]\!]_{[],\,\mathbf{int}}\, \varnothing = \iota_{\uparrow}\, k\ .$$

Por lo tanto, tenemos $(\!|\, t \,|\!)_{[],\,\mathbf{int}},\ [])\ \blacktriangleright^{\mathbf{int}}\ \iota_{\uparrow}\, k$.

Por otro lado, es fácil ver que $[] \in \mathcal{R}^{\mathbf{int}}_{P}(\iota_{\uparrow}\, k)^{\perp} = \mathcal{T}^{\mathbf{int}}(\iota_{\uparrow}\, k)$. Por lo tanto $(\!|\, t \,|\!)_{[],\,\mathbf{int}},\ [],\ []) \in \perp\!\!\!\perp$ como queríamos demostrar. □

Para demostrar completamente la corrección del compilador, debemos ver que cuando $[\![\, t \,]\!]_{[],\,\mathbf{int}}\, \varnothing = \bot$, entonces $(\!|\, t \,|\!)_{[],\,\mathbf{int}},\ [],\ [])$ diverge. Para ello utilizaremos otra relación lógica de aproximación.

Aproximación operacional

Para definir la relación de aproximación operacional aplicaremos un esquema de realizabilidad que utiliza relaciones indexadas decrecientes, similar al que presentamos en la Sección 5.5.

Repetimos aquí las propiedades requeridas sobre el conjunto de observaciones, que son las mismas que en la Definición 78.

Definición 105 (Observaciones). *$\perp\!\!\!\perp_r$ es una relación indexada decreciente sobre Conf tal que:*

1. *$\perp\!\!\!\perp_0 = Conf$ (conjunto total de configuraciones)*
2. *$w' \in \perp\!\!\!\perp_r$ y $w \longmapsto w'$ implica $w \in \perp\!\!\!\perp_{r+1}$.*

De la misma forma, repetimos la definición de los operadores ortogonales para el esquema de realizabilidad usando indexación. Recordemos que esa definición es válida si el operador ortogonal se aplica solamente a conjuntos completos (ver Lema 27), por ello habrá que demostrar que el conjunto de realizadores lo es.

Definición 106 (Operadores ortogonales).

$$X^{\perp} = \{\, (r,\, s) \mid \textit{para todo } (u,\, \alpha) \in X,\, u \leq r \textit{ implica } (\alpha,\, s) \in \perp\!\!\!\perp_u \,\}$$
$$Y^{\top} = \{\, (u,\, \alpha) \mid \textit{para todo } (r,\, s) \in Y,\, r \leq u \textit{ implica } (\alpha,\, s) \in \perp\!\!\!\perp_r \,\}$$

El conjunto de realizadores $\mathcal{R}^{\theta}(d)$ contiene pares de la forma $(u,\, \alpha)$ donde $u \in \mathbb{N}$ y $\alpha \in MClos$. Como antes, el conjunto $\mathcal{R}^{\theta}(d)$ se define en términos de los realizadores primitivos $\mathcal{R}_P^{\theta}(d)$ utilizando el operador de clausura. El conjunto de tests $\mathcal{T}^{\theta}(d) \subseteq \mathbb{N} \times Stack$ se define como $\mathcal{R}^{\theta}(d)^{\perp} = \mathcal{R}_P^{\theta}(d)^{\perp}$.

Definición 107 (Realizadores).

$$\begin{aligned}
\mathcal{R}_P^{\mathbf{int}}(\perp) &= \emptyset \\
\mathcal{R}_P^{\mathbf{int}}(\iota_{\uparrow} k) &= \{\, (u,\, (\mathtt{Const}\ k,\, \eta)) \mid u \in \mathbb{N},\, \eta \in MEnv \,\} \\
\mathcal{R}_P^{\theta \to \theta'}(f) &= \{\, (u,\, (\mathtt{Grab} \triangleright i,\, \eta)) \mid \textit{para todo } u' \leq u, \\
&\quad (u',\, \alpha) \in \mathcal{R}^{\theta}(d), \textit{se tiene } (u',\, (i,\, \alpha :: \eta)) \in \mathcal{R}^{\theta'}(f d) \,\} \\
\mathcal{R}_P^{\theta \times \theta'}((d_0,\, d_1)) &= \{\, (u,\, (\mathtt{Pair}\ (i_0, i_1),\, \eta)) \mid (u,\, (i_0,\, \eta)) \in \mathcal{R}^{\theta}(d_0), \\
&\quad y\ (u,\, (i_1,\, \eta)) \in \mathcal{R}^{\theta'}(d_1) \,\} \\
\mathcal{R}^{\theta}(d) &= \mathcal{R}^{\theta}(d)^{\perp\top}
\end{aligned}$$

Nuevamente se define la relación de aproximación en términos del conjunto de realizadores.

Definición 108 (Aproximación operacional).

$$\alpha \blacktriangleleft_u^{\theta} d \qquad \textit{si y sólo si} \qquad (u,\, \alpha) \in \mathcal{R}^{\theta}(d)\ .$$

Aquí $\alpha \blacktriangleleft_u^\theta d$ se interpreta como "α aproxima a d con índice u y tipo θ". Como $\mathcal{R}_P^\theta(d) \subseteq \mathcal{R}^\theta(d)$ tenemos que $(u, \alpha) \in \mathcal{R}_P^\theta(d)$ implica $\alpha \blacktriangleleft_u^\theta d$. Intuitivamente, los realizadores primitivos son aproximaciones *directas* a los objetos denotacionales, y tomando la extensión vía el operador de clausura obtenemos un conjunto más amplio de aproximaciones que depende de las observaciones elegidas. En particular, requerimos $\mathcal{R}_P^{\mathbf{int}}(\bot) = \emptyset$, esto es, no existen realizadores primitivos de $\bot \in \mathbb{Z}_\bot$, aunque dependiendo de la elección de $\Perp_r$ puede suceder $\mathcal{R}^{\mathbf{int}}(\bot) \neq \emptyset$ (ver Lema 65).

Por definición de $\mathcal{R}_P^{\mathbf{int}}(\iota_\uparrow k)$ se tiene $(\texttt{Const}\ k,\ \eta) \blacktriangleleft_u^{\mathbf{int}} \iota_\uparrow k$ para todo $u \in \mathbb{N}$. Para tipos funcionales tenemos $(\texttt{Grab} \triangleright i,\ \eta) \blacktriangleleft_u^{\theta \to \theta'} f$ si para todo $u' \leq u$, y $d \in [\![\, \theta \,]\!]$, si $\alpha \blacktriangleleft_{u'}^{\theta} d$ entonces $(i,\ \alpha :: \eta) \blacktriangleleft_{u'}^{\theta'} f\, d$. Por último, para los pares, tenemos $(\texttt{Pair}\ (i_0, i_1),\ \eta) \blacktriangleleft_u^{\theta \times \theta'} (d_0,\ d_1)$ si $(i_0,\ \eta) \blacktriangleleft_u^{\theta} d_0$ y $(i_1,\ \eta) \blacktriangleleft_u^{\theta'} d_1$.

A partir del hecho de que $\mathcal{R}_P^{\mathbf{int}}(\bot) = \emptyset$ se desprende el siguiente resultado sobre $\mathcal{T}^{\mathbf{int}}(\bot)$.

Lema 49. $\mathcal{T}^{\mathbf{int}}(\bot) = \mathbb{N} \times \mathit{Stack}$.

Prueba. Es trivial pues $\mathcal{T}^{\mathbf{int}}(\bot) = \mathcal{R}_P^{\mathbf{int}}(\bot)^\bot = \emptyset^\bot = \mathbb{N} \times \mathit{Stack}$. □

Como mencionamos antes, queremos demostrar que $\blacktriangleleft_u^\theta$ es indexada decreciente ya que eso implica que el índice u se puede pensar como el nivel de precisión de la aproximación.

Lema 50. *La relación $\blacktriangleleft_u^\theta$ es indexada decreciente.*

Prueba. Debemos ver que $\blacktriangleleft_u^\theta \subseteq \blacktriangleleft_{u'}^\theta$ cuando $u' \leq u$, o equivalentemente que $\alpha \blacktriangleleft_u^\theta d$ implica $\alpha \blacktriangleleft_{u'}^\theta d$. Por definición, esa implicación es a su vez equivalente a $(u, \alpha) \in \mathcal{R}^\theta(d)$ implica $(u', \alpha) \in \mathcal{R}^\theta(d)$, lo cual es lo mismo que demostrar que $\mathcal{R}^\theta(d)$ es completo. El Lema 26 garantiza que $\mathcal{R}^\theta(d)$ es completo pues $\Perp_r$ es indexada decreciente. □

Como consecuencia de que $\mathcal{R}^\theta(d)$ es completo (Definición 77), se tiene que el conjunto de realizadores primitivos también lo es.

Lema 51. *El conjunto $\mathcal{R}_P^\theta(d)$ es completo.*

Prueba. Se procede por casos en el tipo θ.

- Caso **int**. Si $d = \bot$ se tiene $\mathcal{R}_P^{\mathbf{int}}(\bot) = \emptyset$ por definición, y el conjunto vacío es completo. Si $d = \iota_\uparrow k$, se tiene que $(u, (\texttt{Const}\ k,\ \eta))$ pertenece a $\mathcal{R}_P^{\mathbf{int}}(\iota_\uparrow k)$ para *todo* $u \in \mathbb{N}$ y $\eta \in \mathit{MEnv}$, por lo que $\mathcal{R}_P^{\mathbf{int}}(\iota_\uparrow k)$ es trivialmente completo.

- Caso $\theta \to \theta'$. Supongamos $(u, (\texttt{Grab} \triangleright i, \eta)) \in \mathcal{R}_P^{\theta \to \theta'}(f)$ y $u' \leq u$, debemos ver $(u', (\texttt{Grab} \triangleright i, \eta)) \in \mathcal{R}_P^{\theta \to \theta'}(f)$. Para ello, aplicamos la definición de $\mathcal{R}_P^{\theta \to \theta'}(f)$. Sea $(u'', \alpha) \in \mathcal{R}^{\theta}(d)$ con $u'' \leq u'$. Por transitividad de $\leq$ se tiene $u'' \leq u$, y por lo tanto obtenemos $(u'', (i, \alpha :: \eta)) \in \mathcal{R}^{\theta'}(f\,d)$.

- Caso $\theta \times \theta'$. Supongamos $(u, (\texttt{Pair}\ (i_0, i_1), \eta)) \in \mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$ y tomemos $u' \leq u$. Tenemos $(u, (i_0, \eta)) \in \mathcal{R}^{\theta}(d_0)$ y $(u, (i_1, \eta)) \in \mathcal{R}^{\theta'}(d_1)$. Por Lema 26 se tiene que tanto $\mathcal{R}^{\theta}(d_0)$ y $\mathcal{R}^{\theta'}(d_1)$ son completos. Por lo tanto, $(u', (i_0, \eta)) \in \mathcal{R}^{\theta}(d_0)$ y $(u', (i_1, \eta)) \in \mathcal{R}^{\theta'}(d_1)$. Concluimos entonces $(u', (\texttt{Pair}\ (i_0, i_1), \eta)) \in \mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$.

□

Como la relación $\blacktriangleleft_u^{\theta}$ es decreciente, se tiene que $\blacktriangleleft_0^{\theta}$ debe incluir a todas las demás relaciones con $u > 0$. De hecho, se puede ver que $\blacktriangleleft_0^{\theta} = \mathit{MClos} \times [\![\, \theta \,]\!]$.

Lema 52. *Para todo $\alpha \in \mathit{MClos}$ y $d \in [\![\, \theta \,]\!]$, se cumple $\alpha \blacktriangleleft_0^{\theta} d$.*

Prueba. Para demostrar $(0, \alpha) \in \mathcal{R}^{\theta}(d) = \mathcal{T}^{\theta}(d)^{\top}$ aplicaremos la definición del operador ortogonal. Sea $(u, s) \in \mathcal{T}^{\theta}(d)$. Como $\mathcal{R}^{\theta}(d)$ es completo, basta con analizar únicamente el caso $u = 0$. Por lo tanto vale $(\alpha, s) \in \perp\!\!\!\perp_0 = \mathit{Conf}$. □

Respecto al orden $\sqsubseteq$ del dominio $[\![\, \theta \,]\!]$, se cumple la siguiente propiedad de los realizadores primitivos. Como corolario se tiene que si $\alpha \blacktriangleleft_u^{\theta} d$ y $d \sqsubseteq d'$ entonces $\alpha \blacktriangleleft_u^{\theta} d'$.

Lema 53. *Si $d \sqsubseteq d'$ entonces $\mathcal{R}_P^{\theta}(d) \subseteq \mathcal{R}_P^{\theta}(d')$.*

Prueba. Procedemos por inducción en θ. Usaremos que, por la monotonía del operador de clausura respecto de la inclusión, se tiene $\mathcal{R}^{\theta}(d) \subseteq \mathcal{R}^{\theta}(d')$ cuando $\mathcal{R}_P^{\theta}(d) \subseteq \mathcal{R}_P^{\theta}(d')$.

- Caso **int**. Si $d = \bot$, se tiene $\mathcal{R}_P^{\mathbf{int}}(d) = \emptyset$ y la inclusión es trivial. En otro caso, cuando $d = \iota_{\uparrow} k$, se tiene que $d = d'$ (por definición de $\sqsubseteq$ en el dominio $[\![\, \mathbf{int} \,]\!]$). Nuevamente la inclusión es trivial en ese caso.

- Caso $\theta \to \theta'$. Sea $(u, (\texttt{Grab} \triangleright i, \eta)) \in \mathcal{R}_P^{\theta \to \theta'}(f)$ y $f \sqsubseteq g$. Veamos que $(u, (\texttt{Grab} \triangleright i, \eta)) \in \mathcal{R}_P^{\theta \to \theta'}(g)$ aplicando la definición de $\mathcal{R}_P^{\theta \to \theta'}(g)$. Sea $(u', \alpha) \in \mathcal{R}^{\theta}(d)$ con $u' \leq u$. Obtenemos $(u', (i, \alpha :: \eta)) \in \mathcal{R}^{\theta'}(f\,d)$ por definición de $\mathcal{R}_P^{\theta \to \theta'}(f)$. Como $f\,d \sqsubseteq g\,d$ se tiene por hipótesis inductiva que $\mathcal{R}_P^{\theta'}(f\,d) \subseteq \mathcal{R}_P^{\theta'}(g\,d)$, y por lo tanto $\mathcal{R}^{\theta'}(f\,d) \subseteq \mathcal{R}^{\theta'}(g\,d)$. Podemos concluir entonces que $(u', (i, \alpha :: \eta)) \in \mathcal{R}^{\theta'}(g\,d)$.

- Caso $\theta \times \theta'$. Supongamos $(u, (\texttt{Pair}\ (i_0, i_1), \eta)) \in \mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$ y tomemos $(d_0, d_1) \sqsubseteq (d_0', d_1')$. Por definición de $\mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$ tenemos $(u, (i_0, \eta)) \in \mathcal{R}^{\theta}(d_0)$ y $(u, (i_1, \eta)) \in \mathcal{R}^{\theta'}(d_1)$. Por hipótesis inductiva (y la monotonía del operador de clausura) tenemos $\mathcal{R}^{\theta}(d_0) \subseteq \mathcal{R}^{\theta}(d_0')$, y por lo tanto $(u, (i_0, \eta)) \in \mathcal{R}^{\theta}(d_0')$. De la misma forma $(u, (i_1, \eta)) \in \mathcal{R}^{\theta}(d_1')$. Concluimos entonces $(u, (\texttt{Pair}\ (i_0, i_1), \eta)) \in \mathcal{R}_P^{\theta \times \theta'}((d_0', d_1'))$.

□

Como hicimos en la relación de aproximación denotacional, demostraremos algunos resultados que permiten construir tests a partir de la combinación de otros tests y de realizadores de diferentes tipos.

Lema 54. *Si* $\alpha \blacktriangleleft_u^{\theta} d$ *y* $(u, s) \in \mathcal{T}^{\theta'}(fd)$ *entonces* $(u, \alpha :: s) \in \mathcal{T}^{\theta \to \theta'}(f)$.

Prueba. Debemos demostrar $(u, \alpha :: s) \in \mathcal{T}^{\theta \to \theta'}(f) = \mathcal{R}_P^{\theta \to \theta'}(f)^{\perp}$. Para ello tomemos $(u', (\texttt{Grab} \triangleright i, \eta)) \in \mathcal{R}_P^{\theta \to \theta'}(f)$ con $u' \leq u$. Ya que hemos supuesto $\alpha \blacktriangleleft_u^{\theta} d$, por definición de $\mathcal{R}_P^{\theta \to \theta'}(f)$ tenemos $(u', (i, \alpha :: \eta)) \in \mathcal{R}^{\theta'}(fd)$. Por lo tanto, como $(u', s) \in \mathcal{T}^{\theta'}(fd)$, obtenemos $(i, \alpha :: \eta, s) \in \mathbb{1}_{u'}$. Concluimos entonces $(\texttt{Grab} \triangleright i, \eta, \alpha :: s) \in \mathbb{1}_{u'}$ pues $\mathbb{1}_{u'}$ es cerrado por anti-ejecución. □

Lema 55. *Si* $(u, s) \in \mathcal{T}^{\theta}(d_0)$ *entonces* $(u, (\texttt{Fst}, \eta) :: s) \in \mathcal{T}^{\theta \times \theta'}((d_0, d_1))$. *Similarmente, si* $(u, s) \in \mathcal{T}^{\theta'}(d_1)$ *entonces* $(u, (\texttt{Snd}, \eta) :: s) \in \mathcal{T}^{\theta \times \theta'}((d_0, d_1))$.

Prueba. Veamos $(u, (\texttt{Fst}, \eta) :: s) \in \mathcal{T}^{\theta \times \theta'}((d_0, d_1))$. Para ello tomemos un realizador $(u', (\texttt{Pair}\ (i_0, i_1), \eta')) \in \mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$ con $u' \leq u$. Por definición de $\mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$ tenemos $(u', (i_0, \eta')) \in \mathcal{R}^{\theta}(d_0)$ y $(u', (i_1, \eta')) \in \mathcal{R}^{\theta'}(d_1)$. Por lo tanto tenemos

$$\begin{array}{lll} (\texttt{Pair}\ (i_0, i_1), & \eta', & (\texttt{Fst}, \eta) :: s) \longmapsto \\ (\texttt{Fst}, & \eta, & \langle (i_0, \eta'), (i_1, \eta') \rangle :: s) \longmapsto \\ (i_0, & \eta', & s) \in \mathbb{1}_{u'} \ . \end{array}$$

Luego, $(\texttt{Pair}\ (i_0, i_1), \eta', (\texttt{Fst}, \eta) :: s) \in \mathbb{1}_{u'}$. De manera similar se obtiene la prueba de $(u, (\texttt{Snd}, \eta) :: s) \in \mathcal{T}^{\theta \times \theta'}((d_0, d_1))$. □

Lema 56. *Si* $\alpha_0 \blacktriangleleft_u^{\theta} d_0$, $\alpha_1 \blacktriangleleft_u^{\theta} d_1$, $(u, s) \in \mathcal{T}^{\theta}(IFZ_{\theta}\ d\ (d_0, d_1))$, *entonces se cumple* $(u, \langle \alpha_0, \alpha_1 \rangle :: s) \in \mathcal{T}^{\mathbf{int}}(d)$.

Prueba. Sea $d' = IFZ_{\theta}\ d\ (d_0, d_1)$. Si $d = \bot$, la prueba es directa por Lema 49. Supongamos $d = \iota_{\uparrow} k$, con $k \in \mathbb{Z}$. Tomamos $(u', (\texttt{Const}\ k, \eta)) \in \mathcal{R}_P^{\mathbf{int}}(d)$ con $u' \leq u$. Como $\mathcal{T}^{\theta}(d')$ es completo, $(u', s) \in \mathcal{T}^{\theta}(d')$. Por lo tanto, obtenemos

- Si $k = 0$ se tiene $d' = d_0$ y $(\texttt{Const}\ k,\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \longmapsto (\alpha_0,\ s) \in \mathbb{1}_{u'}$, ya que $\alpha_0 \blacktriangleleft_u^{\theta} d_0$ por hipótesis,
- Si $k \neq 0$ se tiene $d' = d_1$ y $(\texttt{Const}\ k,\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \longmapsto (\alpha_1,\ s) \in \mathbb{1}_{u'}$, ya que $\alpha_1 \blacktriangleleft_u^{\theta} d_1$ por hipótesis.

En ambos casos se obtiene $(\texttt{Const}\ k,\ \eta,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \in \mathbb{1}_{u'}$. Esta prueba vale para todo $(u',\ (\texttt{Const}\ k,\ \eta)) \in \mathcal{R}_P^{\mathbf{int}}(d)$ con $u' \leq u$, por lo tanto concluimos $(u,\ \langle \alpha_0,\ \alpha_1 \rangle :: s) \in \mathcal{T}^{\mathbf{int}}(d)$. □

Lema 57. *Suponiendo $\alpha \blacktriangleleft_u^{\mathbf{int}} d$ y $(u,\ s) \in \mathcal{T}^{\theta}(IFZ_{\theta}\ d\ (d_0,\ d_1))$, entonces se cumple $(u,\ \alpha :: s) \in \mathcal{T}^{\theta \times \theta}((d_0,\ d_1))$.*

Prueba. Tomemos $(u',\ (\texttt{Pair}\ (i_0, i_1),\ \eta)) \in \mathcal{R}_P^{\theta \times \theta}((d_0,\ d_1))$ con $u' \leq u$. Se tiene $(i_0,\ \eta) \blacktriangleleft_{u'}^{\theta} d_0$ y $(i_1,\ \eta) \blacktriangleleft_{u'}^{\theta} d_1$ por definición de $\mathcal{R}_P^{\theta \times \theta}((d_0,\ d_1))$. Como el conjunto $\mathcal{T}^{\theta}(IFZ_{\theta}\ d\ (d_0,\ d_1))$ es completo, $(u',\ s) \in \mathcal{T}^{\theta}(IFZ_{\theta}\ d\ (d_0,\ d_1))$. Por Lema 56 tenemos $(u',\ \langle (i_0,\ \eta),\ (i_1,\ \eta) \rangle :: s) \in \mathcal{T}^{\mathbf{int}}(d)$.

Luego, como $(\texttt{Pair}\ (i_0, i_1),\ \eta,\ \alpha :: s) \longmapsto (\alpha,\ \langle (i_0,\ \eta),\ (i_1,\ \eta) \rangle :: s) \in \mathbb{1}_{u'}$, se obtiene $(\texttt{Pair}\ (i_0, i_1),\ \eta,\ \alpha :: s) \in \mathbb{1}_{u'}$. Concluimos $(u,\ \alpha :: s) \in \mathcal{T}^{\theta \times \theta}((d_0,\ d_1))$. □

Lema 58. *Supongamos $\ominus^n$ es un operador aritmético de aridad n. Entonces para todo $u \in \mathbb{N}$ y $\bar{k},\ \bar{\alpha},\ \bar{d}$ tales que*

$$(1)\ |\bar{k}| + |\bar{\alpha}| = n - 1, \quad (2)\ |\bar{\alpha}| = |\bar{d}|, \quad (3)\ \alpha_r \blacktriangleleft_u^{\mathbf{int}} d_r \text{ para todo } r < |\bar{\alpha}|,$$

y para todo $d' \in [\![\, \mathbf{int} \,]\!]$, $(u,\ s) \in \mathcal{T}^{\mathbf{int}}((\ominus^n \bar{k})_{\perp}(d',\ \bar{d}))$ se cumple

$$(u,\ \{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s) \in \mathcal{T}^{\mathbf{int}}(d') \ .$$

Prueba. Si $d' = \perp$ entonces la prueba es trivial por Lema 49. Supongamos ahora $d' = \iota_{\uparrow} k_0$, con $k_0 \in \mathbb{Z}$. Debemos demostrar, por inducción en el tamaño de $\bar{\alpha}$, que se cumple $(u,\ \{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s) \in \mathcal{T}^{\mathbf{int}}(\iota_{\uparrow} k_0)$. Para cada caso tomamos $(u',\ (\texttt{Const}\ k_0,\ \eta)) \in \mathcal{R}_P^{\mathbf{int}}(\iota_{\uparrow} k_0)$ con $u' \leq u$ y demostramos que se cumple $(\texttt{Const}\ k_0,\ \eta,\ \{\ominus^n \bar{k} \bullet \bar{\alpha}\} :: s) \in \mathbb{1}_{u'}$. Para abreviar, definimos $\hat{\alpha} = (\texttt{Const}\ k_0,\ \eta)$ y $d'' = (\ominus^n \bar{k})_{\perp}(d',\ \bar{d})$.

En el caso $\bar{\alpha} = []$ obtenemos $\bar{d} = []$ y $|\bar{k}| = n - 1$. Por lo tanto tenemos $d'' = \iota_{\uparrow} k'$ donde $k' = \ominus^n (\bar{k},\ k_0)$. Es fácil verificar que se cumple $(\texttt{Const}\ k',\ \eta) \blacktriangleleft_{u'}^{\mathbf{int}} \iota_{\uparrow} k'$, y dado que $(\hat{\alpha},\ \{\ominus^n \bar{k} \bullet\} :: s) \longmapsto (\texttt{Const}\ k',\ \eta,\ s) \in \mathbb{1}_{u'}$, tenemos $(\hat{\alpha},\ \{\ominus^n \bar{k} \bullet\} :: s) \in \mathbb{1}_{u'}$ como queríamos.

En el caso $\bar{\alpha} = (\alpha_0,\ \bar{\alpha}_1)$ tenemos $\bar{d} = (d_0,\ \bar{d}_1)$ para algún $d_0 \in \mathbb{Z}_{\perp}$ tal que $\alpha_0 \blacktriangleleft_u^{\mathbf{int}} d_0$. Como $d'' = (\ominus^n (\bar{k},\ k_0))_{\perp}(d_0,\ \bar{d}_1)$ podemos aplicar hipótesis inductiva para obtener $(u,\ \{\ominus^n \bar{k},\ k_0 \bullet \bar{\alpha}_1\} :: s) \in \mathcal{T}^{\mathbf{int}}(d_0)$. Como $\mathcal{T}^{\mathbf{int}}(d_0)$ es completo, tenemos $(u',\ \{\ominus^n \bar{k},\ k_0 \bullet \bar{\alpha}_1\} :: s) \in \mathcal{T}^{\mathbf{int}}(d_0)$. Por lo tanto, obtenemos que

$(\hat{\alpha},\ \{\ominus^n\ \bar{k} \bullet \bar{\alpha}\} :: s) \longmapsto (\alpha_0,\ \{\ominus^n\ \bar{k},\ k_0 \bullet \bar{\alpha}_1\} :: s) \in \mathbb{1}_{u'}$, y entonces concluimos $(\hat{\alpha},\ \{\ominus^n\ \bar{k} \bullet \bar{\alpha}\} :: s) \in \mathbb{1}_{u'}$ como queríamos demostrar. □

Definimos a continuación la relación de aproximación para entornos:

Definición 109 (Aproximación operacional para entornos).

$$[] \blacktriangleleft_u^{[]} \varnothing$$

$$\alpha :: \eta \blacktriangleleft_u^{\theta :: \pi} (d,\ \rho) \qquad \textit{si y sólo si} \qquad \alpha \blacktriangleleft_u^{\theta} d \quad y \quad \eta \blacktriangleleft_u^{\pi} \rho\ .$$

A continuación mostramos diversas maneras de componer aproximaciones con el fin de obtener nuevas aproximaciones de diferentes objetos denotacionales.

Lema 59. *Si* $\eta \blacktriangleleft_u^{\pi} \rho$ *y* $n < |\pi|$, *entonces* $(\texttt{Access}\ n,\ \eta) \blacktriangleleft_u^{\pi.n} \rho \ \swarrow\ n$.

Prueba. Sea $(u',\ s) \in \mathcal{R}_p^{\theta}(\rho \ \swarrow\ n)$ donde $u' \leq u$ y $\theta = \pi \cdot n$. Luego, como $\eta \blacktriangleleft_u^{\pi} \rho$ y $n < |\pi|$ obtenemos $(\eta \cdot n,\ s) \blacktriangleleft_u^{\theta} \rho \ \swarrow\ n$, y por lo tanto obtenemos $(\texttt{Access}\ n,\ \eta,\ s) \longmapsto (\eta \cdot n,\ s) \in \mathbb{1}_{u'}$. Luego, $(\texttt{Access}\ n,\ \eta,\ s) \in \mathbb{1}_{u'}$. □

Lema 60. *Si* $(i,\ \eta) \blacktriangleleft_u^{\theta \to \theta'} f$ *y* $(i',\ \eta) \blacktriangleleft_u^{\theta} d$, *entonces* $(\texttt{Push}\ i' \triangleright i,\ \eta) \blacktriangleleft_u^{\theta'} fd$.

Prueba. Sea $(u',\ s) \in \mathcal{T}^{\theta'}(fd)$ con $u' \leq u$. Como $\mathcal{R}^{\theta}(d)$ es completo, tenemos $(i',\ \eta) \blacktriangleleft_{u'}^{\theta} d$, y por Lema 54 obtenemos $(u',\ (i',\ \eta) :: s) \in \mathcal{T}^{\theta \to \theta'}(f)$. Luego, como $(\texttt{Push}\ i' \triangleright i,\ \eta,\ s) \longmapsto (i,\ \eta,\ (i',\ \eta) :: s) \in \mathbb{1}_{u'}$, podemos concluir que vale $(\texttt{Push}\ i' \triangleright i,\ \eta,\ s) \in \mathbb{1}_{u'}$. Por lo tanto, $(\texttt{Push}\ i' \triangleright i,\ \eta) \blacktriangleleft_u^{\theta'} fd$. □

Lema 61. *Si* $(i,\ \eta) \blacktriangleleft_u^{\theta \to \theta} f$ *entonces* $(\texttt{Fix} \triangleright i,\ \eta) \blacktriangleleft_u^{\theta} \mathbf{Y}_{[\![\theta]\!]} f$.

Prueba. Sea $d = \mathbf{Y}_{[\![\theta]\!]} f$ y $\alpha = (\texttt{Fix} \triangleright i,\ \eta)$. Queremos ver $\alpha \blacktriangleleft_u^{\theta} d$, procedemos por inducción en u. El caso $u = 0$ es directo por Lema 52. Supongamos $u = u' + 1$. Sea $(r,\ s) \in \mathcal{T}^{\theta}(d)$ con $r \leq u$, veamos $(\alpha,\ s) \in \mathbb{1}_r$.

Si $r \leq u'$, por hipótesis inductiva $\alpha \blacktriangleleft_{u'}^{\theta} d$ y por lo tanto $(\alpha,\ s) \in \mathbb{1}_r$. Supongamos ahora $r = u = u' + 1$. Como d es un punto fijo de f, se tiene $fd = d$, luego $(u,\ s) \in \mathcal{T}^{\theta}(fd)$. Dado que $\mathcal{T}^{\theta}(fd)$ es completo, se obtiene $(u',\ s) \in \mathcal{T}^{\theta}(fd)$. Por hipótesis inductiva $\alpha \blacktriangleleft_{u'}^{\theta} d$, y por Lema 54 se tiene $(u',\ \alpha :: s) \in \mathcal{T}^{\theta \to \theta}(f)$. En consecuencia, $(\alpha,\ s) \longmapsto (i,\ \eta,\ \alpha :: s) \in \mathbb{1}_{u'}$. Por la restricción 2 de la Definición 105 se tiene $(\alpha,\ s) \in \mathbb{1}_{u'+1=r}$ como queríamos. □

Lema 62. *Suponiendo que* $(i_r,\ \eta) \blacktriangleleft_u^{\mathbf{int}} d_r$ *para todo* $r \in \{1,\ \ldots,\ n\}$, *entonces se cumple* $(\texttt{Push}\ i_n \triangleright \ldots \triangleright \texttt{Push}\ i_1 \triangleright \texttt{Frame}\ \ominus^n,\ \eta) \blacktriangleleft_u^{\mathbf{int}} \ominus_{\perp}^n\ (d_1,\ \ldots,\ d_n)$.

Prueba. Sea $(u', s) \in \mathcal{T}^{\mathbf{int}}(\ominus^n_\perp (d_1, \ldots, d_n))$ con $u' \leq u$, sea $\alpha_r = (i_r, \eta)$ donde $r \in \{1, \ldots, n\}$ y sea $\bar{\alpha} = (\alpha_2, \ldots, \alpha_n)$. Como $\mathcal{R}^{\mathbf{int}}(d_r)$ es completo, se tiene $\alpha_r \blacktriangleleft^{\mathbf{int}}_{u'} d_r$. Por Lema 58 obtenemos $(u', \{\ominus^n \bullet \bar{\alpha}\} :: s) \in \mathcal{T}^{\mathbf{int}}(d_1)$. Por lo tanto, como

$$\begin{array}{lllll} (\mathtt{Push}\ i_n \triangleright \ldots \triangleright \mathtt{Push}\ i_1 \triangleright \mathtt{Frame}\ \ominus^n, & \eta, & s) & & \longmapsto^* \\ (\mathtt{Frame}\ \ominus^n, & \eta, & \alpha_1 :: \alpha_2 :: \ldots :: \alpha_n :: s) & & \longmapsto \\ (i_1, & \eta, & \{\ominus^n \bullet \bar{\alpha}\} :: s) & & \in \mathbb{1}_{u'}\ , \end{array}$$

obtenemos $(\mathtt{Push}\ i_n \triangleright \ldots \triangleright \mathtt{Push}\ i_1 \triangleright \mathtt{Frame}\ \ominus^n, \eta, s) \in \mathbb{1}_{u'}$. □

Lema 63. *Suponiendo que* $(i, \eta) \blacktriangleleft^{\theta \times \theta'}_u (d_0, d_1)$, *entonces* $(\mathtt{Push\ Fst} \triangleright i, \eta) \blacktriangleleft^{\theta}_u d_0$ *y además* $(\mathtt{Push\ Snd} \triangleright i, \eta) \blacktriangleleft^{\theta'}_u d_1$.

Prueba. Sea $(u', s) \in \mathcal{T}^{\theta}(d_0)$ con $u' \leq u$. Como consecuencia del Lema 55 obtenemos $(u', (\mathtt{Fst}, \eta) :: s) \in \mathcal{T}^{\theta \times \theta'}((d_0, d_1))$. Luego, obtenemos

$$(\mathtt{Push\ Fst} \triangleright i, \eta, s) \longmapsto (i, \eta, (\mathtt{Fst}, \eta) :: s) \in \mathbb{1}_{u'}\ .$$

Por lo tanto, $(\mathtt{Push\ Fst} \triangleright i, \eta, s) \in \mathbb{1}_{u'}$. La prueba de $(\mathtt{Push\ Snd} \triangleright i, \eta) \blacktriangleleft^{\theta'}_u d_1$ es análoga. □

Lema 64. *Suponiendo que valen* $(i, \eta) \blacktriangleleft^{\theta \times \theta}_u (d_0, d_1)$ *y* $(i', \eta) \blacktriangleleft^{\mathbf{int}}_u d$, *entonces se cumple* $(\mathtt{Push}\ i' \triangleright i, \eta) \blacktriangleleft^{\theta}_u \mathit{IFZ}_\theta\ d\ (d_0, d_1)$.

Prueba. Sea $d' = \mathit{IFZ}_\theta\ d\ (d_0, d_1)$. Tomamos $(u', s) \in \mathcal{T}^{\theta}(d')$ donde $u' \leq u$. Luego, como $\mathcal{R}^{\mathbf{int}}(d)$ es completo, se tiene $(i', \eta) \blacktriangleleft^{\mathbf{int}}_{u'} d$. Además, por Lema 57 se tiene $(u', (i', \eta) :: s) \in \mathcal{T}^{\theta \times \theta}((d_0, d_1))$.

Por lo tanto, $(\mathtt{Push}\ i' \triangleright i, \eta, s) \longmapsto (i, \eta, (i', \eta) :: s) \in \mathbb{1}_{u'}$. Concluimos entonces $(\mathtt{Push}\ i' \triangleright i, \eta) \blacktriangleleft^{\theta}_u d' = \mathit{IFZ}_\theta\ d\ (d_0, d_1)$. □

La diferencia más importante de estos resultados respecto a los análogos de la aproximación denotacional es el Lema 61, donde se demuestra por inducción en el índice u que $(i, \eta) \blacktriangleleft^{\theta \to \theta}_u f$ implica $(\mathtt{Fix} \triangleright i, \eta) \blacktriangleleft^{\theta}_u \mathrm{Y}_{[\![\theta]\!]} f$. Dado que es posible aproximar puntos fijos de forma directa, no será necesario la cadena semántica para demostrar la corrección del compilador. Aplicando los lemas anteriores se puede demostrar que la compilación de un término aproxima a su semántica.

Teorema 11. *Supongamos* $\pi \vdash t : \theta$, *entonces para todo* $u \in \mathbb{N}$, *si* $\eta \blacktriangleleft^{\pi}_u \rho$ *entonces* $(\!|\, t \,|\!)_{\pi, \theta}, \eta) \blacktriangleleft^{\theta}_u [\![\, t \,]\!]_{\pi, \theta} \rho$.

Prueba. Se procede por inducción en la derivación de $\pi \vdash t : \theta$.

- Caso (Abs). Veamos $(u, (\!| \lambda\, t |\!)_{\pi,\, \theta \to \theta'}, \eta)) \in \mathcal{R}_P^{\theta \to \theta'}(f) \subseteq \mathcal{R}^{\theta \to \theta'}(f)$ donde $f = [\![\lambda\, t]\!]_{\pi,\, \theta \to \theta'} \rho$. Sea $(u', \alpha) \in \mathcal{R}^{\theta}(d)$ con $u' \leq u$. Como $\eta \blacktriangleleft_u^{\pi} \rho$ y $\blacktriangleleft_u^{\pi}$ es indexada decreciente, se tiene $\eta \blacktriangleleft_{u'}^{\pi} \rho$ y por lo tanto $\alpha :: \eta \blacktriangleleft_{u'}^{\theta :: \pi} (d, \rho)$. Luego, por hipótesis inductiva, se obtiene

$$(\!| t |\!)_{\theta :: \pi,\, \theta'}, \alpha :: \eta) \blacktriangleleft_{u'}^{\theta'} [\![t]\!]_{\theta :: \pi,\, \theta'} (d, \rho) = f d \ .$$

  Concluimos, por definición de $\mathcal{R}_P^{\theta \to \theta'}(f)$, que $(\!| \lambda\, t |\!)_{\pi,\, \theta \to \theta'}, \eta) \blacktriangleleft_u^{\theta \to \theta'} f$.

- Caso (Var). Como $\eta \blacktriangleleft_u^{\pi} \rho$, $n < |\pi|$ y $\pi \cdot n = \theta$, aplicamos Lema 59 para obtener

$$(\!| \bar{n} |\!)_{\pi,\, \theta}, \eta) = (\mathtt{Access}\ n, \eta) \blacktriangleleft_u^{\theta} \rho \ \swarrow \ n = [\![\bar{n}]\!]_{\pi,\, \theta} \rho \ .$$

- Caso (App). Sea $f = [\![t]\!]_{\pi,\, \theta \to \theta'} \rho$ y $d = [\![t']\!]_{\pi,\, \theta} \rho$. Por hipótesis inductiva tenemos $(\!| t |\!)_{\pi,\, \theta \to \theta'}, \eta) \blacktriangleleft_u^{\theta \to \theta'} f$ y también $(\!| t' |\!)_{\pi,\, \theta}, \eta) \blacktriangleleft_u^{\theta} d$. Por Lema 60 obtenemos

$$(\!| t\ t' |\!)_{\pi,\, \theta'}, \eta) = (\mathtt{Push}\ (\!| t' |\!)_{\pi,\, \theta} \triangleright (\!| t |\!)_{\pi,\, \theta \to \theta'}, \eta) \blacktriangleleft_u^{\theta'} f d = [\![t\ t']\!]_{\pi,\, \theta'} \rho \ .$$

- Caso (Rec). Sea $f = [\![t]\!]_{\pi,\, \theta \to \theta} \rho$. Sabemos $(\!| t |\!)_{\pi,\, \theta \to \theta}, \eta) \blacktriangleleft_u^{\theta \to \theta} f$ por hipótesis inductiva. Por Lema 61 tenemos

$$(\!| rec\ t |\!)_{\pi,\, \theta}, \eta) = (\mathtt{Fix} \triangleright (\!| t |\!)_{\pi,\, \theta \to \theta}, \eta) \blacktriangleleft_u^{\theta} \mathbf{Y}_{[\![\theta]\!]} f = [\![rec\ t]\!]_{\pi,\, \theta} \rho \ .$$

- Caso (Const). Como $[\![\underline{k}]\!]_{\pi,\, \mathbf{int}} \rho = \iota_{\uparrow} k$ y $(u, (\mathtt{Const}\ k, \eta)) \in \mathcal{R}_P^{\mathbf{int}}(\iota_{\uparrow} k)$ obtenemos

$$(\!| \underline{k} |\!)_{\pi,\, \mathbf{int}}, \eta) = (\mathtt{Const}\ k, \eta) \blacktriangleleft_u^{\mathbf{int}} \iota_{\uparrow} k = [\![\underline{k}]\!]_{\pi,\, \mathbf{int}} \rho \ .$$

- Caso (Op). Sea $d_r = [\![t_r]\!]_{\pi,\, \mathbf{int}} \rho$ y $i_r = (\!| t_r |\!)_{\pi,\, \mathbf{int}}$ con $r \in \{1, \ldots, n\}$, queremos ver $(\!| \ominus^n\ (t_1, \ldots, t_n) |\!)_{\pi,\, \mathbf{int}}, \eta) \blacktriangleleft_u^{\mathbf{int}} \ominus_{\perp}^n (d_1, \ldots, d_n)$. Por hipótesis inductiva tenemos $(i_r, \eta) = (\!| t_r |\!)_{\pi,\, \mathbf{int}}, \eta) \blacktriangleleft_u^{\mathbf{int}} d_r$. Luego, por Lema 62, obtenemos

$$\begin{array}{ll} (\!| \ominus^n\ (t_1, \ldots, t_n) |\!)_{\pi,\, \mathbf{int}}, \eta) & = \\ (\mathtt{Push}\ i_n \triangleright \ldots \triangleright \mathtt{Push}\ i_1 \triangleright \mathtt{Frame}\ \ominus^n, \eta) & \blacktriangleleft_u^{\mathbf{int}} \\ \ominus_{\perp}^n (d_1, \ldots, d_n) & = [\![\ominus^n\ (t_1, \ldots, t_n)]\!]_{\pi,\, \mathbf{int}} \rho \ . \end{array}$$

- Caso (Pair). Sea $d_0 = [\![t]\!]_{\pi,\, \theta} \rho$ y $d_1 = [\![t']\!]_{\pi,\, \theta'} \rho$. Por hipótesis inductiva $(\!| t |\!)_{\pi,\, \theta}, \eta) \blacktriangleleft_u^{\theta} d_0$ y $(\!| t' |\!)_{\pi,\, \theta'}, \eta) \blacktriangleleft_u^{\theta'} d_1$, por definición de $\mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))$ obtenemos

$$\begin{array}{rcl} (\!| (t,\, t') |\!)_{\pi,\, \theta \times \theta'}, \eta) & = & (\mathtt{Pair}\ ((\!| t |\!)_{\pi,\, \theta}, (\!| t' |\!)_{\pi,\, \theta'}), \eta) \\ & \in & \mathcal{R}_P^{\theta \times \theta'}([\![(t,\, t')]\!]_{\pi,\, \theta \times \theta'} \rho) \ . \end{array}$$

- Caso (Fst). Sea $(d_0, d_1) = [\![t]\!]_{\pi,\, \theta \times \theta'}\, \rho$. Por hipótesis inductiva obtenemos $(\!| t |\!)_{\pi,\, \theta \times \theta'}, \eta) \blacktriangleleft_u^{\theta \times \theta'} (d_0, d_1)$. Luego, por Lema 63, tenemos

$$(\!| \mathit{fst}\ t |\!)_{\pi,\, \theta}, \eta) = (\mathsf{Push\ Fst} \triangleright (\!| t |\!)_{\pi,\, \theta \times \theta'}, \eta) \blacktriangleleft_u^{\theta} d_0 = [\![\mathit{fst}\ t]\!]_{\pi,\, \theta}\, \rho \ .$$

- Caso (Snd). Similar al anterior.

- Caso (Cond). Sean $d = [\![t]\!]_{\pi,\, \mathbf{int}}\, \rho$ y $(d_0, d_1) = [\![t']\!]_{\pi,\, \theta \times \theta}\, \rho$. Por hipótesis inductiva tenemos $(\!| t |\!)_{\pi,\, \mathbf{int}}, \eta) \blacktriangleleft_u^{\mathbf{int}} d$ y $(\!| t' |\!)_{\pi,\, \theta \times \theta}, \eta) \blacktriangleleft_u^{\theta \times \theta} (d_0, d_1)$. Por Lema 64 obtenemos

$$\begin{array}{rcl} (\!| \mathit{ifz}\ t\ .\ t' |\!)_{\pi,\, \theta}, \eta) & = & (\mathsf{Push}\ (\!| t' |\!)_{\pi,\, \mathbf{int}} \triangleright (\!| t |\!)_{\pi,\, \theta \times \theta}, \eta) \\ & & \blacktriangleleft_u^{\theta}\ \mathit{IFZ}_\theta\ d\ (d_0, d_1) \ . \end{array}$$

□

Antes de demostrar la corrección para términos divergentes, necesitamos el siguiente resultado sobre las aproximaciones de tipo **int**.

Lema 65. *Definimos*

$$\perp\!\!\!\perp_r = \{\, w \mid \textit{se pueden realizar al menos } r \textit{ transiciones a partir de } w \,\} \ .$$

Entonces, si $\alpha \blacktriangleleft_u^{\mathbf{int}} \bot$ *para todo* $u \in \mathbb{N}$, *se tiene* $(\alpha, s) \longmapsto^{\infty}$ *para toda* $s \in \mathit{Stack}$.

Prueba. Sea $(N, s) \in \mathbb{N} \times \mathit{Stack}$. Por hipótesis tenemos $\alpha \blacktriangleleft_N^{\mathbf{int}} \bot$, y además por Lema 49, $(N, s) \in \mathcal{T}^{\mathbf{int}}(\bot)$. Luego $(\alpha, s) \in \perp\!\!\!\perp_N$, es decir, se pueden realizar al menos N transiciones a partir de (α, s). Como N es arbitrario, concluimos que a partir de (α, s) es posible realizar cualquier número de transiciones. Como el sistema de transiciones de la máquina abstracta es determinista, se puede demostrar que si a partir de w se puede realizar una cantidad arbitraria de transiciones, entonces w diverge en el sentido coinductivo de la Definición 25. □

Teorema 12. *Si* $[] \vdash t : \mathbf{int}$ *y* $[\![t]\!]_{[],\, \mathbf{int}}\, \varnothing = \bot$, *entonces* $(\!| t |\!)_{[],\, \mathbf{int}}, [], s) \longmapsto^{\infty}$ *para toda* $s \in \mathit{Stack}$.

Prueba. Definimos $\perp\!\!\!\perp_r$ como en el Lema 65. Luego, por Teorema 11 tenemos $(\!| t |\!)_{[],\, \mathbf{int}}, []) \blacktriangleleft_u^{\mathbf{int}} \bot$ para todo $u \in \mathbb{N}$. Por lo tanto, por Lema 65, concluimos $(\!| t |\!)_{[],\, \mathbf{int}}, [], s) \longmapsto^{\infty}$ para toda $s \in \mathit{Stack}$. □

De esa manera hemos probado la corrección del compilador para términos convergentes (Teorema 10) y para términos divergentes (Teorema 12) a partir de las relaciones de aproximación denotacional y operacional respectivamente. En la formalización en Coq presentamos también una relación lógica definida

mediante la *clausura admisible* de la relación de aproximación denotacional (es decir, la misma relación pero cerrada por límites de cadenas) que no incluimos aquí ya que no resultó necesaria para obtener el teorema de corrección para tipos básicos. Un aspecto importante a considerar es la diferencia en el comportamiento de la máquina de Krivine y la máquina SECD utilizada por Benton [15]. En la primera, el resultado de una computación se deduce observando la configuración final de manera completa (analizando todos los componentes de la configuración). En la segunda, en cambio, el resultado final de una evaluación queda almacenado en el tope de la pila. La máquina SECD distingue claramente entre valores y clausuras, modelando de manera más natural la evaluación estricta. Por ese motivo, el enfoque de Benton es más adecuado para lenguajes estrictos pues distingue entre tres tipos de relaciones lógicas destinadas específicamente a valores, expresiones y computaciones. Nuestro enfoque, en cambio, se centra en la máquina de Krivine como entorno de ejecución, logrando por lo tanto capturar de manera más clara e intuitiva la noción de corrección para lenguajes de evaluación normal.

Capítulo 7

Realizabilidad y Evaluación Lazy

Hasta aquí hemos considerado distintas variantes de la máquina de Krivine que modelan en su comportamiento el orden de evaluación normal que posee el lenguaje fuente. Una de las situaciones en la que se ve reflejado el orden de evaluación es en la ejecución del código de una aplicación. Para ver esto recordemos la Definición 101 para el caso de la aplicación:

$$(\!| \, t \; t' \, |\!)_{\pi,\, \theta'} = \mathsf{Push}\ ((\!| \, t' \, |\!)_{\pi,\, \theta}) \triangleright (\!| \, t \, |\!)_{\pi,\, \theta \to \theta'} \quad .$$

Observamos que al ejecutar la instrucción Push, el código que resulta de compilar el argumento t' se coloca en el tope de la pila y no es ejecutado a menos que, eventualmente, sea invocado por una instrucción Access, indicando que la evaluación del argumento resulta necesaria para continuar la ejecución. Sin embargo, el mecanismo de la máquina de Krivine no impide que el argumento se evalúe más de una vez cuando no es necesario. Por ejemplo, en un término de la forma $(\lambda \oplus (\bar{0}, \bar{0}))\ t'$, donde $\oplus$ es un operador aritmético binario, el código de t' será ejecutado una vez por cada llamada a la instrucción Access 0. Cuando el lenguaje es puramente funcional, es posible evitar esa repetición de la ejecución. Alternativamente, podemos considerar una máquina abstracta con evaluación *lazy*, es decir, que mantenga el orden de evaluación normal pero que a su vez garantice que los argumentos se evalúen una única vez. En este capítulo aplicaremos realizabilidad sobre una variante de la máquina abstracta de Sestoft [111], donde se utilizarán *heaps* para almacenar el valor de los argumentos y de esa forma evitar la repetición de la evaluación. A partir del esquema de realizabilidad se logra obtener un resultado acerca de la corrección de la compilación respecto de la semántica denotacional del lenguaje. Como veremos, la presencia del heap dificulta considerablemente la aplicación de realizabilidad para lograr la prueba de corrección. Por esa razón, limitamos el trabajo de esta tesis a un lenguaje tipado sin recursión, y dejaremos la incorporación del operador de punto fijo como trabajo futuro.

7.1 Sintaxis

El lenguaje de este capítulo es una variante del cálculo lambda utilizado por Launchbury [75] y Sestoft [111] en sus trabajos sobre evaluación lazy. Además de los términos del cálculo lambda, se incorporan la constante $\diamond$ y el término *let t in t′* (llamado *declaración de variable*).

Definición 110 (Términos).

$$t, t' \in \mathit{Term} ::= \bar{n} \mid \lambda\, t \mid t\, \bar{n} \mid \diamond \mid \mathit{let}\ t\ \mathit{in}\ t'$$

Observemos que la aplicación está restringida pues el argumento sólo puede ser una variable. Como veremos, esta restricción asegura que el código del argumento se mantenga siempre dentro del heap, facilitando así que su ejecución se realice a lo sumo una vez.

A diferencia de los trabajos de Launchbury y Sestoft recién mencionados, el término *let t in t′* que posee este lenguaje no puede usarse para definir términos recursivos.

Otra diferencia es que aquí definiremos un sistema de tipos para el lenguaje. De esa forma, obtenemos un lenguaje *fuertemente normalizable*, es decir, todos los términos bien tipados pueden reducirse a una forma canónica.

El sistema de tipos consta de tipos funcionales y del tipo **unit**.

Definición 111 (Tipos).

$$\theta, \theta' \in \mathit{Type} ::= \theta \to \theta' \mid \mathbf{unit}$$

Las siguientes reglas permiten construir derivaciones de juicios de tipado. Estas reglas son las usuales del cálculo lambda, excepto que la aplicación aparece en su versión restringida, y se incorpora además el término *let t in t′* y la constante $\diamond$ de tipo **unit**.

Definición 112 (Reglas de tipado).

$$\textsc{Abs}\ \frac{\theta :: \pi \vdash t : \theta'}{\pi \vdash \lambda\, t : \theta \to \theta'} \qquad \textsc{App}\ \frac{\pi \vdash t : \theta \to \theta'}{\pi \vdash t\, \bar{n} : \theta'}\ \pi \cdot n = \theta$$

$$\textsc{Var}\ \frac{}{\pi \vdash \bar{n} : \theta}\ \pi \cdot n = \theta \qquad \textsc{Unit}\ \frac{}{\pi \vdash \diamond : \mathbf{unit}}$$

$$\textsc{Let}\ \frac{\pi \vdash t : \theta \qquad \theta :: \pi \vdash t' : \theta'}{\pi \vdash \mathit{let}\ t\ \mathit{in}\ t' : \theta'}$$

7.2 Semántica

Ante la ausencia del operador de punto fijo, no es necesario usar dominios para definir la semántica del lenguaje, y utilizaremos en su lugar teoría de conjuntos. Comenzamos definiendo la semántica de los tipos del lenguaje.

Definición 113 (Semántica de tipos).

$$\begin{aligned} [\![\textbf{\textit{unit}}]\!] &= \{ \varnothing \} \\ [\![\theta \to \theta']\!] &= [\![\theta]\!] \to [\![\theta']\!] \end{aligned}$$

El tipo **unit** está asociado con el conjunto $\{ \varnothing \}$ cuyo único elemento es $\varnothing$, aunque podría elegirse cualquier conjunto de cardinalidad 1 (la unidad del producto cartesiano). El tipo funcional $\theta \to \theta'$ está asociado con el espacio de funciones totales de $[\![\theta]\!]$ en $[\![\theta']\!]$. Por otro lado, la semántica de un contexto π es el conjunto de tuplas de la forma $(d_1, \ldots, d_n)$ donde $n = |\pi|$ y cada d_j pertenece a $[\![\pi \cdot j]\!]$. Como antes, las tuplas $\rho \in [\![\pi]\!]$ se denominan *ambientes*.

Definición 114 (Semántica de contextos).

$$\begin{aligned} [\![[]]\!] &= \{ \varnothing \} \\ [\![\theta :: \pi]\!] &= [\![\theta]\!] \times [\![\pi]\!] \end{aligned}$$

De la misma manera que en el Capítulo 6, la semántica denotacional del lenguaje se define de manera intrínseca, asociando semántica a las derivaciones de tipos en lugar de a los términos primitivos del lenguaje. A cada derivación de tipos con conclusión $\pi \vdash t : \theta$ se le asigna una función total de $[\![\pi]\!]$ en $[\![\theta]\!]$.

Definición 115 (Semántica denotacional).

$$\begin{aligned} [\![\lambda\, t]\!]_{\pi,\, \theta \to \theta'}\, \rho &= \hat{\lambda}\, d\, .\, [\![t]\!]_{\theta :: \pi,\, \theta'}\, (d,\, \rho) \\ [\![\bar{n}]\!]_{\pi,\, \theta}\, \rho &= \rho \swarrow n \\ [\![t\ \bar{n}]\!]_{\pi,\, \theta'}\, \rho &= [\![t]\!]_{\pi,\, \theta \to \theta'}\, \rho\, (\rho \swarrow n) \\ [\![\diamond]\!]_{\pi,\, \textbf{\textit{unit}}}\, \rho &= \varnothing \\ [\![\mathit{let}\ t\ \mathit{in}\ t']\!]_{\pi,\, \theta'}\, \rho &= [\![t']\!]_{\theta :: \pi,\, \theta'}\, ([\![t]\!]_{\pi,\, \theta}\, \rho,\, \rho) \end{aligned}$$

7.3 La máquina abstracta

La máquina abstracta de Sestoft [111] es una adaptación de la máquina de Krivine que incorpora a las configuraciones un nuevo componente llamado *heap*,

cuya utilidad radica principalmente en evitar la ejecución innecesaria de código. Los componentes de la máquina se definen a continuación:

Definición 116 (Componentes).

$$
\begin{array}{lcll}
i,\ i' \in \mathit{Code} & ::= & \mathtt{Access}\ n & \textit{(Código)} \\
 & \mid & \mathtt{Grab} \triangleright i \mid \mathtt{Push}\ n \triangleright i & \\
 & \mid & \mathtt{Unit} \mid \mathtt{Let}\ i \triangleright i' & \\
p,\ q \in \mathit{Pointer} & & & \textit{(Punteros)} \\
\Gamma,\ \Delta \in \mathit{Heap} & = & \mathit{Pointer} \to \mathit{MClos} & \textit{(Heaps)} \\
\alpha \in \mathit{MClos} & ::= & (i,\ \eta) & \textit{(Clausuras de máquina)} \\
\eta \in \mathit{MEnv} & ::= & [] \mid p :: \eta & \textit{(Entornos de máquina)} \\
s \in \mathit{Stack} & ::= & [] \mid p :: s \mid \sharp p :: s & \textit{(Pilas)} \\
w \in \mathit{Conf} & ::= & (\Gamma,\ i,\ \eta,\ s) & \textit{(Configuraciones)}
\end{array}
$$

A diferencia de la máquina de Krivine, tanto los entornos de máquina como las pilas están compuestas por punteros y no por clausuras. El heap es una función de dominio finito que asocia a cada puntero una clausura de máquina. Las pilas pueden contener además *marcadores* de la forma $\sharp p$, que como veremos, se utilizan para indicar que la clausura asociada a p dentro del heap debe ser actualizada.

Frecuentemente necesitaremos referirnos al conjunto de todos los punteros que forman parte del entorno (o también de la pila, la clausura o el heap). A continuación definimos el operador $\mathit{ptr}(\_)$ que puede aplicarse a una componente de la máquina, y devuelve el conjunto de punteros que ocurren en la misma.

Definición 117 (Conjunto de punteros en componentes).

$$
\begin{aligned}
\mathit{ptr}(\eta \in \mathit{MEnv}) &= \{\, p \mid \textit{existe } n \in \mathbb{N},\ \eta \cdot n = p \,\} \\
\mathit{ptr}(s \in \mathit{Stack}) &= \{\, p \mid \textit{existe } n \in \mathbb{N},\ s \cdot n = p \textit{ o bien } s \cdot n = \sharp p \,\} \\
\mathit{ptr}((i,\ \eta) \in \mathit{MClos}) &= \mathit{ptr}(\eta) \\
\mathit{ptr}(\Gamma \in \mathit{Heap}) &= \mathit{dom}(\Gamma) \cup \{\, q \mid \textit{existe } p \in \mathit{dom}(\Gamma),\ q \in \mathit{ptr}(\Gamma\ p) \,\}
\end{aligned}
$$

Notemos que $\mathit{ptr}(\Gamma)$ incluye tanto a los punteros del rango como los del dominio del heap Γ. Para el caso de las pilas, $\mathit{ptr}(s)$ contiene a todos los punteros que ocurren en s, incluyendo también a los marcadores.

Continuamos con la definición de las reglas de transición que determinan el comportamiento de la máquina abstracta.

Definición 118 (Transiciones).

$$
\begin{array}{lll}
(\Gamma,\ \texttt{Grab} \triangleright i,\ \eta,\ p :: s) & \longmapsto & (\Gamma,\ i,\ p :: \eta,\ s) \\
(\Gamma,\ \texttt{Grab} \triangleright i,\ \eta,\ \sharp p :: s) & \longmapsto & (\Gamma[\, p \mapsto (\texttt{Grab} \triangleright i,\ \eta)\,],\ \texttt{Grab} \triangleright i,\ \eta,\ s) \\
(\Gamma,\ \texttt{Unit},\ \eta,\ \sharp p :: s) & \longmapsto & (\Gamma[\, p \mapsto (\texttt{Unit},\ \eta)\,],\ \texttt{Unit},\ \eta,\ s) \\
(\Gamma,\ \texttt{Access}\ n,\ \eta,\ s) & \longmapsto & (\Gamma,\ i',\ \eta',\ \sharp p :: s) \\
 & \multicolumn{2}{l}{\textit{donde } n < |\,\eta\,|,\ p = \eta \cdot n,\ (i',\ \eta') = \Gamma\, p} \\
(\Gamma,\ \texttt{Push}\ n \triangleright i,\ \eta,\ s) & \longmapsto & (\Gamma,\ i,\ \eta,\ p :: s) \\
 & \multicolumn{2}{l}{\textit{donde } p = \eta \cdot n} \\
(\Gamma,\ \texttt{Let}\ i \triangleright i',\ \eta,\ s) & \longmapsto & (\Gamma[\, p \mapsto (i,\ \eta)\,],\ i',\ p :: \eta,\ s) \\
 & \multicolumn{2}{l}{\textit{donde } p \notin \mathit{ptr}(\Gamma) \cup \mathit{ptr}(\eta) \cup \mathit{ptr}(s)}
\end{array}
$$

La instrucción $\texttt{Grab} \triangleright i$ tiene dos reglas de transición que corresponden a los casos en los que, en el tope de la pila, se encuentre un puntero o bien un marcador. Si en el tope hay un puntero, la transición consiste en mover ese puntero al entorno y continuar con la ejecución de i. Por otro lado, cuando en el tope de la pila hay un marcador $\sharp p$, se está indicando que la clausura $(\texttt{Grab} \triangleright i,\ \eta)$ (donde η es el entorno actual) es en realidad el resultado de haber ejecutado el código de la clausura $\Gamma\, p$, y ahora debe actualizarse el heap en ese punto para evitar que $\Gamma\, p$ sea ejecutada nuevamente. Con la notación $\Gamma[p \mapsto \alpha]$ denotamos al heap con dominio $\mathit{dom}(\Gamma) \cup \{\, p \,\}$ tal que $\Gamma[p \mapsto \alpha]\, p = \alpha$ y además $\Gamma[p \mapsto \alpha]\, q = \Gamma\, q$ cuando $q \neq p$. Notemos que únicamente las instrucciones $\texttt{Grab} \triangleright i$ y $\texttt{Unit}$ realizan una actualización del heap, puesto que esas instrucciones representan valores del lenguaje (respectivamente, a las abstracciones y la constante $\diamond$) y lo que se busca es evitar que la máquina compute nuevamente esos valores de manera innecesaria.

La instrucción $\texttt{Access}\ n$ comienza con la ejecución del código de la clausura $\Gamma\, p$ donde p es el puntero ubicado en la n-ésima posición del entorno. Además, esta instrucción inserta un marcador $\sharp p$ en el tope de la pila indicando que $\Gamma\, p$ debe actualizarse cuando la ejecución llegue a una clausura de la forma $(\texttt{Grab} \triangleright i,\ \eta)$ o bien $(\texttt{Unit},\ \eta)$. Podría darse el caso en el que la ejecución de $\Gamma\, p$ alcance una configuración de la forma $(\Delta,\ \texttt{Access}\ n',\ \eta')$ donde $\eta' \cdot n' = p$ y $\Delta\, p = \Gamma\, p$. Esa situación se denomina *black hole* [85], pues produce la divergencia de la ejecución. La transición de la instrucción $\texttt{Access}\ n$ podría detectar el potencial black hole eliminando el puntero p del dominio de Γ, es decir, siendo $(\Gamma,\ \texttt{Access}\ n,\ \eta,\ s) \longmapsto (\Gamma - \{\, p \,\},\ \Gamma\, p,\ \sharp p :: s)$ (donde $p = \eta \cdot n$) la regla alternativa para la instrucción $\texttt{Access}$. En ese caso, un eventual black hole produciría el bloqueo de la máquina y no su divergencia. Sin embargo, los black holes no pueden ocurrir cuando el código a ejecutar ha sido generado por el compilador (definido en la siguiente sección). Por ello, y para simplificar el razonamiento, se decidió no incluir la detección de black holes en las transiciones de la máquina.

La instrucción $\texttt{Push}\ n \triangleright i$ inserta en el tope de la pila el puntero ubicado en la n-ésima posición del entorno. A diferencia de la instrucción homónima de la máquina de Krivine, esta versión no introduce una clausura en el tope de la pila sino una referencia (puntero) a la misma.

La instrucción $\texttt{Let}\ i \triangleright i'$ incrementa el tamaño del heap insertando un puntero nuevo p asociado con la clausura (i, η) donde η es el entorno actual. La ejecución del código i' se realiza en el entorno extendido $p :: \eta$. La condición $p \notin ptr(\Gamma) \cup ptr(\eta) \cup ptr(s)$ asegura que el puntero p no ocurra en ningún lugar de la configuración. Como la elección de p se deja sin especificar, el sistema de reglas de transición es no determinista. Sin embargo, es posible evitar ese no-determinismo eligiendo una representación concreta para los punteros y definiendo un algoritmo determinista de generación de nombres nuevos. Sestoft [111] demostró que la generación de punteros nuevos puede hacerse localmente, es decir, analizando sólo la configuración actual y no la traza completa de la ejecución. Esto vale aún en el caso de incluir detección de black holes, donde los punteros que están en la pila o en el entorno no necesariamente ocurren en el dominio del heap.

7.4 Compilación

Continuamos con la definición de la función de compilación:

Definición 119 (Compilación).

$$
\begin{array}{lcl}
(\!| \lambda\, t |\!)_{\pi,\, \theta \to \theta'} & = & \texttt{Grab} \triangleright (\!| t |\!)_{\theta :: \pi,\, \theta'} \\
(\!| \overline{n} |\!)_{\pi,\, \theta} & = & \texttt{Access}\ n \\
(\!| t\ \overline{n} |\!)_{\pi,\, \theta'} & = & \texttt{Push}\ n \triangleright (\!| t |\!)_{\pi,\, \theta \to \theta'} \\
(\!| \diamond |\!)_{\pi,\, \mathit{unit}} & = & \texttt{Unit} \\
(\!| \mathit{let}\ t\ \mathit{in}\ t' |\!)_{\pi,\, \theta'} & = & \texttt{Let}\ (\!| t |\!)_{\pi,\, \theta} \triangleright (\!| t' |\!)_{\theta :: \pi,\, \theta'}
\end{array}
$$

Al igual que en la semántica denotacional, la función de compilación se define para derivaciones de tipos y no para términos primitivos, aunque el resultado final sólo va depender del término involucrado en la conclusión de las derivación. Esta elección obedece al hecho de que, en la formalización en Coq, no se define un tipo inductivo para los términos primitivos, sino que en su lugar se define directamente el tipo inductivo de los "términos bien tipados", que es un tipo dependiente parametrizado por un contexto y un tipo (ver Sección 8.6).

7.5 Relación de aproximación

A diferencia del Capítulo 6, aquí sólo tendremos una única noción de aproximación, dado que no tenemos términos divergentes. En esta sección definiremos la relación de aproximación que permitirá demostrar la corrección de la compilación. Comenzaremos presentando algunas operaciones sobre heaps necesarias para definir la relación de satisfacibilidad de los operadores ortogonales.

Diremos que un heap es *bien formado* cuando todos los punteros que ocurren en el heap están también en su dominio. Por ejemplo, es fácil ver que el heap $\Gamma = \{\, p \mapsto (\texttt{Unit}, [\, q\,])\,\}$ no es bien formado pues $q \notin \mathit{dom}(\Gamma)$. El predicado $\mathit{wf}(\Gamma)$ establece que Γ es bien formado, y su definición se extiende también a pares de la forma $(\Gamma,\ \alpha)$ y $(\Gamma,\ s)$ como sigue:

Definición 120 (Heap bien formado).

$$\begin{aligned} \mathit{wf}(\Gamma) \quad &\textit{si y sólo si} \quad \mathit{ptr}(\Gamma) = \mathit{dom}(\Gamma) \\ \mathit{wf}(\Gamma,\ \alpha) \quad &\textit{si y sólo si} \quad \mathit{wf}(\Gamma)\ \textit{y}\ \mathit{ptr}(\alpha) \subseteq \mathit{dom}(\Gamma)\,, \\ \mathit{wf}(\Gamma,\ s) \quad &\textit{si y sólo si} \quad \mathit{wf}(\Gamma)\ \textit{y}\ \mathit{ptr}(s) \subseteq \mathit{dom}(\Gamma)\ . \end{aligned}$$

Es claro que si Γ es bien formado y $p \in \mathit{dom}(\Gamma)$, entonces el par $(\Gamma,\ \Gamma\, p)$ también es bien formado.

Lema 66. *Si $\mathit{wf}(\Gamma)$ entonces para todo $p \in \mathit{dom}(\Gamma)$, $\mathit{wf}(\Gamma,\ \Gamma\, p)$.*

Los heaps Γ y Δ son *compatibles* si coinciden en los punteros comunes, o equivalentemente:

Definición 121 (Heaps compatibles). *Los heaps Γ y Δ son compatibles (escribimos, $\Gamma \bowtie \Delta$) si para todo $p \in \mathit{dom}(\Gamma) \cap \mathit{dom}(\Delta)$ se tiene $\Gamma\, p = \Delta\, p$.*

Cuando los heaps Γ y Δ son compatibles, la unión de heaps $\Gamma \cup \Delta$ está bien definida.

Definición 122 (Unión de heaps). *Cuando $\Gamma \bowtie \Delta$, el heap unión $\Gamma \cup \Delta$ con dominio $\mathit{dom}(\Gamma) \cup \mathit{dom}(\Delta)$ se define como:*

$$(\Gamma \cup \Delta)\, p = \begin{cases} \Gamma\, p & \textit{si } p \in \mathit{dom}(\Gamma) \\ \Delta\, p & \textit{si } p \in \mathit{dom}(\Delta)\ . \end{cases}$$

De la definición de la relación $\bowtie \subseteq \mathit{Heap} \times \mathit{Heap}$, se desprenden trivialmente las siguientes propiedades:

Lema 67. *Propiedades de la relación* $\bowtie$*:*

1. $\Gamma \bowtie \Gamma$
2. *Si* $\Gamma \bowtie \Delta$ *entonces* $\Delta \bowtie \Gamma$,
3. *Si* $\Gamma \bowtie \Delta$ *y* $\Gamma \bowtie \Delta'$ *entonces* $\Gamma \bowtie (\Delta \cup \Delta')$,
4. *Si* $\Gamma \bowtie (\Delta \cup \Delta')$ *entonces* $\Gamma \bowtie \Delta$ *y* $\Gamma \bowtie \Delta'$.

Es fácil demostrar que $\cup$ es conmutativa y asociativa. Además, cuando Γ y Δ están bien formados, la unión $\Gamma \cup \Delta$ también lo está. Notemos que la recíproca del lema no siempre es verdadera.

Lema 68. *Si* $wf(\Gamma)$ *y* $wf(\Delta)$ *entonces* $wf(\Gamma \cup \Delta)$.

Antes de definir la relación de satisfacibilidad, primero debemos describir las propiedades del conjunto de observaciones $\perp\!\!\!\perp \subseteq \mathit{Conf}$. Esta vez necesitaremos imponer algunas condiciones adicionales al conjunto $\perp\!\!\!\perp$, además de requerir que sea cerrado por anti-ejecución.

Definición 123 (Observaciones). *Asumimos* $\perp\!\!\!\perp$ *es un conjunto de observaciones cerrado por anti-ejecución y además cerrado por las siguientes reglas inductivas:*

$$O_1 \ \frac{(\Gamma,\ i,\ \eta,\ s) \in \perp\!\!\!\perp}{(\Gamma \cup \Delta,\ i,\ \eta,\ s) \in \perp\!\!\!\perp}\ wf(\Gamma),\ \Gamma \bowtie \Delta,\ ptr(\eta) \cup ptr(s) \subseteq dom(\Gamma)$$

$$O_2 \ \frac{(\Gamma,\ i,\ \eta,\ s) \in \perp\!\!\!\perp}{(\Gamma,\ i,\ \eta,\ \sharp p :: s) \in \perp\!\!\!\perp}\ \Gamma\, p = (i,\ \eta)$$

La regla O_1 indica que el heap puede extenderse con punteros que sean irrelevantes a la ejecución. Es decir, dado que $ptr(\eta) \cup ptr(s) \subseteq dom(\Gamma)$ y $wf(\Gamma)$, todos los punteros que ocurren en la configuración $(\Gamma,\ i,\ \eta,\ s)$ están en el dominio de Γ, y por lo tanto $dom(\Delta) - dom(\Gamma)$ es un conjunto de punteros que, a pesar de estar formar parte del heap $\Gamma \cup \Delta$, no serán nunca referenciados durante la ejecución.

La regla O_2 permite incorporar nuevos marcadores a la pila de la configuración. Si $(\Gamma,\ i,\ \eta,\ s) \in \perp\!\!\!\perp$ y $\Gamma\, p = (i,\ \eta)$ podemos incluir el marcador $\#p$ para habilitar una eventual actualización de $\Gamma\, p$. Intuitivamente, esto implica que el mecanismo para evitar la repetición de la ejecución de código, es una optimización que en realidad no debería afectar al conjunto de observaciones elegido.

Si bien omitimos la demostración, se puede comprobar que, por ejemplo, el conjunto de configuraciones que terminan (aquellas que alcanzan una configuración bloqueante) es una instancia válida de $\perp\!\!\!\perp$. Es claro que incorporar

punteros irrelevantes no afecta a la terminación, aunque sí puede pasar que las trazas de ejecución sean diferentes según cómo se defina el algoritmo de generación de nombres nuevos. Habilitar o no una actualización en el heap también puede producir diferencias en las trazas, pero tampoco afecta en la terminación. Si no exigiésemos $wf(\Gamma)$ en O_1, tendríamos el siguiente contra-ejemplo cuando $\perp\!\!\!\perp$ es el conjunto de configuraciones que terminan:

$$(\{\, p \mapsto (\texttt{Access}\ 0, [\, q\,])\,\}, \texttt{Access}\ 0, [\, p\,], s) \in \perp\!\!\!\perp \text{ pero}$$

$$(\{\, p \mapsto (\texttt{Access}\ 0, [\, q\,])\,\} \cup \{\, q \mapsto (\texttt{Access}\ 0, [\, q\,])\,\}, \texttt{Access}\ 0, [\, p\,], s) \notin \perp\!\!\!\perp.$$

La relación de satisfacibilidad $\models \subseteq (\mathit{Heap} \times \mathit{MClos}) \times (\mathit{Heap} \times \mathit{Stack})$, queda definida como sigue:

Definición 124 (Satisfacibilidad).

$$(\Gamma,\ \alpha) \models (\Delta,\ s) \quad \textit{si y sólo si}$$
$$wf(\Gamma,\ \alpha),\ wf(\Delta,\ s),\ \Gamma \bowtie \Delta \quad \textit{implica} \quad (\Gamma \cup \Delta,\ \alpha,\ s) \in \perp\!\!\!\perp\ .$$

A diferencia de lo que sucede en la máquina de Krivine, donde una clausura en sí misma constituye un realizador, aquí el heap también debe formar parte del mismo, dado que se necesita darle sentido a los punteros que ocurren en el entorno de la clausura. Lo mismo sucede con el conjunto de tests, donde las pilas deben estar acompañadas por un heap. Cuando el realizador y el test se unen para formar una configuración, los heaps correspondientes a cada uno de ellos deben combinarse con la operación $\cup$, como lo indica la definición de la relación $\models$. Por lo tanto, se exige que ambos heaps sean compatibles, y también bien formados. Con esa relación de satisfacibilidad, los operadores ortogonales se definen como sigue:

Definición 125 (Operadores ortogonales).

$$X^{\perp} = \{\, (\Delta,\ s) \mid \textit{para todo}\ (\Gamma,\ \alpha) \in X\ \textit{se tiene}\ (\Gamma,\ \alpha) \models (\Delta,\ s)\,\}$$
$$Y^{\top} = \{\, (\Gamma,\ \alpha) \mid \textit{para todo}\ (\Delta,\ s) \in Y\ \textit{se tiene}\ (\Gamma,\ \alpha) \models (\Delta,\ s)\,\}\ .$$

Los realizadores primitivos representan aproximaciones directas a objetos denotacionales, y mediante el operador de clausura se lograr extender ese conjunto de aproximaciones teniendo en cuenta la relación de satisfacibilidad y el conjunto de observaciones elegido.

Definición 126 (Realizadores).

$$\mathcal{R}_P^{\boldsymbol{unit}}(\varnothing) = \{\, (\Gamma, \texttt{Unit}, \eta) \mid ptr(\eta) \subseteq dom(\Gamma) \,\}$$
$$\mathcal{R}_P^{\theta \to \theta'}(f) = \{\, (\Gamma, \texttt{Grab} \triangleright i, \eta) \mid ptr(\eta) \subseteq dom(\Gamma),$$
$$\text{para todo } d \in [\![\, \theta \,]\!],\ (\Gamma', \alpha) \in \mathcal{R}^{\theta}(d) \text{ tal que } wf(\Gamma', \alpha)\ y\ \Gamma \bowtie \Gamma',$$
$$y \text{ para todo } p \in Pointer \text{ tal que } (\Gamma \cup \Gamma') \bowtie \{p \mapsto \alpha\},$$
$$\text{se cumple } (\Gamma \cup \Gamma' \cup \{p \mapsto \alpha\}, i, p :: \eta) \in \mathcal{R}^{\theta'}(f\,d) \,\}$$
$$\mathcal{R}^{\theta}(d) = \mathcal{R}_P^{\theta}(d)^{\perp\,\top} \ .$$

El conjunto de tests se define como $\mathcal{T}^{\theta}(d) = \mathcal{R}^{\theta}(d)^{\perp} = \mathcal{R}_P^{\theta}(d)^{\perp}$. La relación de aproximación se define directamente en términos del conjunto de realizadores.

Definición 127 (Relación de aproximación).

$$(\Gamma, \alpha)\ \blacksquare^{\theta}\ d \quad \text{si y sólo si} \quad (\Gamma, \alpha) \in \mathcal{R}^{\theta}(d) \ .$$

Por definición de $\mathcal{R}_P^{\mathbf{unit}}(\varnothing)$ tenemos $(\Gamma, \texttt{Unit}, \eta)\ \blacksquare^{\mathbf{unit}}\ \varnothing$ para todo heap Γ y $\eta \in MEnv$ tal que $ptr(\eta) \subseteq dom(\Gamma)$. Para ver $(\Gamma, \texttt{Grab} \triangleright i, \eta)\ \blacksquare^{\theta \to \theta'}\ f$, debemos mostrar que al extender el entorno con una aproximación de un elemento $d \in [\![\, \theta \,]\!]$ obtenemos una aproximación de $f\,d \in [\![\, \theta' \,]\!]$. Si suponemos que vale $(\Gamma', \alpha)\ \blacksquare^{\theta}\ d$, como el entorno sólo puede extenderse utilizando un puntero, debemos tomar $p \in Pointer$ y asociarlo en el heap con la clausura α. Para ello necesitamos que $(\Gamma \cup \Gamma') \bowtie \{p \mapsto \alpha\}$, dejando abierta la posibilidad de que p sea un puntero que ya esté en el heap, o que, por el contrario, sea un puntero totalmente nuevo, fuera del dominio de $\Gamma \cup \Gamma'$. Luego se debe probar que $(\Gamma \cup \Gamma' \cup \{p \mapsto \alpha\}, i, p :: \eta)\ \blacksquare^{\theta'}\ f\,d$.

La estrategia para demostrar que $(\Gamma, \alpha) \in \mathcal{R}^{\theta}(d) = \mathcal{T}^{\theta}(d)^{\top}$ es probar que para todo $(\Delta, s) \in \mathcal{T}^{\theta}(d)$ se cumple $(\Gamma, \alpha) \vDash (\Delta, s)$. Para ello se usa la definición de la relación $\vDash$ y se asume $wf(\Gamma, \alpha)$, $wf(\Delta, s)$ y $\Gamma \bowtie \Delta$ y luego se demuestra que $(\Gamma \cup \Delta, \alpha, s) \in \perp\!\!\!\perp$. Como consecuencia de la regla O_1, el heap de un realizador se puede extender mediante la unión con otro heap:

Lema 69. *Suponiendo* $wf(\Gamma, \alpha), \Gamma \bowtie \Delta\ y\ (\Gamma, \alpha)\ \blacksquare^{\theta}\ d$ *entonces* $(\Gamma \cup \Delta, \alpha)\ \blacksquare^{\theta}\ d$.

Prueba. Supongamos $wf(\Gamma \cup \Delta, \alpha)$. Tomemos $(\Delta', s) \in \mathcal{T}^{\theta}(d)$ tal que $wf(\Delta', s)$ y $(\Gamma \cup \Delta) \bowtie \Delta'$. Queremos ver $(\Gamma \cup \Delta \cup \Delta', \alpha, s) \in \perp\!\!\!\perp$.

Como $(\Gamma \cup \Delta) \bowtie \Delta'$, tenemos $\Gamma \bowtie \Delta'$ y $\Delta \bowtie \Delta'$. Luego, dado que se tiene $(\Gamma, \alpha)\ \blacksquare^{\theta}\ d$, podemos concluir $(\Gamma \cup \Delta', \alpha, s) \in \perp\!\!\!\perp$. Por otro lado, como $wf(\Gamma)$ y $wf(\Delta')$, tenemos por Lema 68 que $wf(\Gamma \cup \Delta')$. Además se tiene $(\Gamma \cup \Delta') \bowtie \Delta$

ya que $\Gamma \bowtie \Delta$ y $\Delta \bowtie \Delta'$. Por lo tanto, aplicando la regla O_1, podemos concluir $(\Gamma \cup \Delta' \cup \Delta, \alpha, s) \in \perp\!\!\!\perp$. □

La relación de aproximación $\blacksquare^{\pi}$ se define por inducción estructural en el contexto π, siendo una extensión punto a punto de la relación $\blacksquare^{\theta}$.

Definición 128 (Aproximación de entornos).

$$
\begin{aligned}
&(\Gamma, [])\ \blacksquare^{[]}\ \varnothing\ ,\\
&(\Gamma, p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d, \rho)\ \textit{si y sólo si}\\
&\qquad p \in \mathit{dom}(\Gamma),\ (\Gamma, \Gamma\, p)\ \blacksquare^{\theta}\ d\ \ y\ \ (\Gamma, \eta)\ \blacksquare^{\pi}\ \rho\ .
\end{aligned}
$$

Las siguientes son dos propiedades que se desprenden trivialmente de la definición de $\blacksquare^{\pi}$:

Lema 70. *Si* $(\Gamma, \eta)\ \blacksquare^{\pi}\ \rho$ *entonces* $|\eta| = |\pi|$, *y* $\mathit{ptr}(\eta) \subseteq \mathit{dom}(\Gamma)$.

Lema 71. *Si* $(\Gamma, \eta)\ \blacksquare^{\pi}\ \rho$, *entonces para todo* $n < |\eta|$, $(\Gamma, \Gamma\,(\eta \cdot n))\ \blacksquare^{\pi \cdot n}\ \rho \swarrow n$.

Como consecuencia del Lema 69, la extensión del heap también es válida para los realizadores de ambientes, como se indica a continuación.

Lema 72. *Si* $\mathit{wf}(\Gamma), \Gamma \bowtie \Delta$ *y* $(\Gamma, \eta)\ \blacksquare^{\pi}\ \rho$ *entonces* $(\Gamma \cup \Delta, \eta)\ \blacksquare^{\pi}\ \rho$.

Prueba. Procedemos por inducción en el contexto π. El caso $\pi = []$ es directo por definición ya que $(\Gamma \cup \Delta, [])\ \blacksquare^{[]}\ \varnothing$. Supongamos $(\Gamma, p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d, \rho)$ y veamos $(\Gamma \cup \Delta, p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d, \rho)$. Por definición de $\blacksquare^{\theta :: \pi}$ tenemos $(\Gamma, \eta)\ \blacksquare^{\pi}\ \rho$ y $(\Gamma, \Gamma\, p)\ \blacksquare^{\theta}\ d$. Dado que $\mathit{wf}(\Gamma)$, por Lema 66 tenemos $\mathit{wf}(\Gamma, \Gamma\, p)$, por lo tanto aplicando Lema 69 obtenemos $(\Gamma \cup \Delta, \Gamma\, p)\ \blacksquare^{\theta}\ d$. Como $(\Gamma, \eta)\ \blacksquare^{\pi}\ \rho$, por hipótesis inductiva tenemos $(\Gamma \cup \Delta, \eta)\ \blacksquare^{\pi}\ \rho$. Luego, por definición de $\blacksquare^{\theta :: \pi}$ concluimos $(\Gamma \cup \Delta, p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d, \rho)$. □

El siguiente resultado permite construir una aproximación del ambiente $(d, \rho) \in [\![\, \theta :: \pi \,]\!]$ a partir de una aproximación de $d \in [\![\, \theta \,]\!]$ y de una aproximación de $\rho \in [\![\, \pi \,]\!]$.

Lema 73. *Suponiendo* $\mathit{wf}(\Gamma), \mathit{wf}(\Gamma', \alpha), \Gamma \bowtie \Gamma', (\Gamma', \alpha)\ \blacksquare^{\theta}\ d, (\Gamma, \eta)\ \blacksquare^{\pi}\ \rho$ *y además* $(\Gamma \cup \Gamma') \bowtie \{p \mapsto \alpha\}$, *entonces* $(\Gamma \cup \Gamma' \cup \{p \mapsto \alpha\}, p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d, \rho)$.

Prueba. Sea $\Gamma'' = \Gamma \cup \Gamma' \cup \{p \mapsto \alpha\}$. Por definición de $\blacksquare^{\theta :: \pi}$ debemos ver que $(\Gamma'', \Gamma''\, p)\ \blacksquare^{\theta}\ d$, y por otro lado, que $(\Gamma'', \eta)\ \blacksquare^{\pi}\ \rho$.

Como $\Gamma''\, p = \alpha$, debemos ver $(\Gamma'', \alpha)\ \blacksquare^{\theta}\ d$. Como $\Gamma' \bowtie \Gamma$ y $\Gamma' \bowtie \{p \mapsto \alpha\}$, tenemos $\Gamma' \bowtie (\Gamma \cup \{p \mapsto \alpha\})$. Luego, como $(\Gamma', \alpha)\ \blacksquare^{\theta}\ d$ y además $\mathit{wf}(\Gamma', \alpha)$, por Lema 69 podemos concluir $(\Gamma' \cup \Gamma \cup \{p \mapsto \alpha\}, \alpha) = (\Gamma'', \alpha)\ \blacksquare^{\theta}\ d$. Por otro lado, dado que $(\Gamma, \eta)\ \blacksquare^{\pi}\ \rho$ y $\mathit{wf}(\Gamma)$, por Lema 72 obtenemos $(\Gamma'', \eta)\ \blacksquare^{\pi}\ \rho$. □

De manera similar a como ocurría en el Capítulo 6, se pueden construir tests para una función $f \in [\![\theta \to \theta']\!]$ combinando un realizador de un argumento $d \in [\![\theta]\!]$ y un test para la aplicación $f\,d \in [\![\theta']\!]$.

Lema 74. *Supongamos* $(\Gamma,\ \alpha) \in \mathcal{R}^{\theta}(d)$, $(\Delta,\ s) \in \mathcal{T}^{\theta'}(f\,d)$ *y* $p \in$ *Pointer tales que* $wf(\Gamma,\ \alpha)$, $wf(\Delta,\ s)$, $\Gamma \bowtie \Delta$ *y además* $(\Gamma \cup \Delta) \bowtie \{\, p \mapsto \alpha \,\}$. *Entonces se cumple* $(\Gamma \cup \Delta \cup \{\, p \mapsto \alpha \,\},\ p :: s) \in \mathcal{T}^{\theta \to \theta'}(f)$.

Prueba. Para abreviar definimos $\Gamma' = \Gamma \cup \Delta \cup \{\, p \mapsto \alpha \,\}$; mostremos que se cumple $(\Gamma',\ p :: s) \in \mathcal{R}_P^{\theta \to \theta'}(f)^{\perp}$. Para ello tomemos $(\Gamma'',\ \hat{\alpha}) \in \mathcal{R}_P^{\theta \to \theta'}(f)$ tal que $wf(\Gamma'',\ \hat{\alpha})$ y $\Gamma' \bowtie \Gamma''$, y demostremos $(\Gamma' \cup \Gamma'',\ \hat{\alpha},\ p :: s) \in \perp\!\!\!\perp$.

Por definición de $\mathcal{R}_P^{\theta \to \theta'}(f)$ tenemos $\hat{\alpha} = (\texttt{Grab} \triangleright i,\ \eta)$ para algún $i \in \mathit{Code}$, $\eta \in \mathit{MEnv}$. Además, como $\Gamma'' \bowtie \Gamma$, y $\Gamma'' \cup \Gamma \bowtie \{\, p \mapsto \alpha \,\}$, obtenemos por definición de $\mathcal{R}_P^{\theta \to \theta'}(f)$ que $(\Gamma'' \cup \Gamma \cup \{\, p \mapsto \alpha \,\},\ i,\ p :: \eta) \in \mathcal{R}^{\theta'}(f\,d)$.

Si $\Delta' = \Gamma'' \cup \Gamma \cup \{\, p \mapsto \alpha \,\}$, es fácil ver que $\Delta \bowtie \Delta'$ y $wf(\Delta')$. Luego, como $(\Delta,\ s) \in \mathcal{T}^{\theta'}(f\,d)$, tenemos $(\Delta' \cup \Delta,\ i,\ p :: \eta,\ s) \in \perp\!\!\!\perp$. Se puede comprobar fácilmente que $\Delta' \cup \Delta = \Gamma' \cup \Gamma''$. Luego, tenemos $(\Gamma' \cup \Gamma'',\ i,\ p :: \eta,\ s) \in \perp\!\!\!\perp$. Como $\perp\!\!\!\perp$ es cerrado por anti-ejecución concluimos $(\Gamma' \cup \Gamma'',\ \hat{\alpha},\ p :: s) \in \perp\!\!\!\perp$ como queríamos demostrar. □

A partir de la aproximación de un ambiente, se puede obtener una aproximación para cada una de sus proyecciones, como lo establece el siguiente resultado.

Lema 75. *Si* $wf(\Gamma)$, $(\Gamma,\ \eta)\ \blacksquare^{\pi}\ \rho, n < |\eta|$, *entonces* $(\Gamma,\ \texttt{Access}\ n,\ \eta)\ \blacksquare^{\pi \cdot n}\ \rho \swarrow n$.

Prueba. Sea $\theta = \pi \cdot n$ y $d = \rho \swarrow n$. Tomamos $(\Delta,\ s) \in \mathcal{T}^{\theta}(d)$ tal que $wf(\Delta,\ s)$ y $\Gamma \bowtie \Delta$. Como $n < |\,\eta\,|$, por Lema 71 tenemos que $(\Gamma,\ \Gamma\,q)\ \blacksquare^{\theta}\ d$, donde $q = \eta \cdot n$. Por lo tanto, tenemos $(\Gamma \cup \Delta,\ \Gamma\,q,\ s) \in \perp\!\!\!\perp$. Por definición se tiene $(\Gamma \cup \Delta)\,q = \Gamma\,q$, por lo que podemos aplicar la regla O_2 para obtener $(\Gamma \cup \Delta,\ \Gamma\,q,\ \sharp q :: s) \in \perp\!\!\!\perp$. Luego concluimos $(\Gamma \cup \Delta,\ \texttt{Access}\ n,\ \eta,\ s) \in \perp\!\!\!\perp$ por anti-ejecución. □

Teniendo la aproximación de una función $f \in [\![\theta \to \theta']\!]$, y una aproximación de $\rho \in [\![\pi]\!]$, podemos construir una aproximación para $f(\rho \swarrow n) \in [\![\theta']\!]$, siempre que se cumpla $\pi \cdot n = \theta$.

Lema 76. *Si* $wf(\Gamma)$, $(\Gamma,\ \eta)\ \blacksquare^{\pi}\ \rho, n < |\,\eta\,|$ *y* $(\Gamma,\ i,\ \eta)\ \blacksquare^{(\pi \cdot n) \to \theta'}\ f$, *entonces se cumple* $(\Gamma,\ \texttt{Push}\ n \triangleright i,\ \eta)\ \blacksquare^{\theta'}\ f(\rho \swarrow n)$.

Prueba. Sea $\theta = \pi \cdot n$, $d = \rho \swarrow n$, $q = \eta \cdot n$. Sea $(\Delta,\ s) \in \mathcal{T}^{\theta'}(f\,d)$ tal que $wf(\Delta,\ s)$ y $\Gamma \bowtie \Delta$. Dado que $(\Gamma,\ \eta)\ \blacksquare^{\pi}\ \rho$ y $n < |\,\eta\,|$, por Lema 71 se tiene $(\Gamma,\ \Gamma\,q) \in \mathcal{R}^{\theta}(d)$. Además, es claro que $(\Gamma \cup \Delta) \bowtie \{\, q \mapsto \Gamma\,q \,\}$. Luego por

Lema 74, obtenemos $(\Gamma \cup \Delta,\ q :: s) \in \mathcal{T}^{\theta \to \theta'}(f)$. Como $(\Gamma,\ i,\ \eta)\ \blacksquare^{\theta \to \theta'}\ f$, se tiene que $(\Gamma \cup \Delta,\ i,\ \eta,\ q :: s) \in \perp\!\!\!\perp$. Luego $(\Gamma \cup \Delta, \mathtt{Push}\ n \triangleright i,\ \eta,\ s) \in \perp\!\!\!\perp$ por anti-ejecución. □

Finalmente, a partir de los resultados anteriores, podemos demostrar que la compilación de un término aproxima a su semántica. Esto es, suponiendo que $\pi \vdash t : \theta$ es un juicio válido, $wf(\Gamma)$ y $(\Gamma,\ \eta)\ \blacksquare^{\pi}\ \rho$, se cumple que $(\!|\ t\ |\!)_{\pi,\ \theta},\ \eta)$ aproxima a $[\![\ t\ ]\!]_{\pi,\ \theta}\rho$.

Teorema 13. *Supongamos* $wf(\Gamma)$, $(\Gamma,\ \eta)\ \blacksquare^{\pi}\ \rho$ *y además* $\pi \vdash t : \theta$, *entonces se cumple* $(\Gamma,\ (\!|\ t\ |\!)_{\pi,\ \theta},\ \eta)\ \blacksquare^{\theta}\ [\![\ t\ ]\!]_{\pi,\ \theta}\rho$.

Prueba. Procedemos por inducción en la derivación de $\pi \vdash t : \theta$.

- Caso (Abs). Debemos ver que $(\Gamma, \mathtt{Grab} \triangleright (\!|\ t\ |\!)_{\theta :: \pi,\ \theta'},\ \eta) \in \mathcal{R}_P^{\theta \to \theta'}(f)$ donde $f = [\![\ \lambda\, t\ ]\!]_{\pi,\ \theta \to \theta'}\ \rho$. Recurrimos a la definición de $\mathcal{R}_P^{\theta \to \theta'}(f)$, y tomamos $d \in [\![\ \theta\ ]\!]$, $p \in \mathit{Pointer}$, $(\Gamma',\ \alpha) \in \mathcal{R}^{\theta}(d)$ tal que $wf(\Gamma',\ \alpha)$, $\Gamma \bowtie \Gamma'$ y además $(\Gamma \cup \Gamma') \bowtie \{p \mapsto \alpha\}$.

  Definimos $\Gamma'' = \Gamma \cup \Gamma' \cup \{p \mapsto \alpha\}$; aplicando el Lema 73 obtenemos $(\Gamma'',\ p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d,\ \rho)$. Además, es fácil ver que $wf(\Gamma'')$, por lo que podemos aplicar hipótesis inductiva para obtener

  $$(\Gamma'',\ (\!|\ t\ |\!)_{\theta :: \pi,\ \theta'},\ p :: \eta)\ \blacksquare^{\theta'}\ [\![\ t\ ]\!]_{\theta :: \pi,\ \theta'}\ (d,\ \rho) = f\, d\ .$$

  Esto demuestra que $(\Gamma, \mathtt{Grab} \triangleright (\!|\ t\ |\!)_{\theta :: \pi,\ \theta'},\ \eta) \in \mathcal{R}_P^{\theta \to \theta'}(f)$ como queríamos.

- Caso (Var). Como $(\Gamma,\ \eta)\ \blacksquare^{\pi}\ \rho$ y $n < |\,\pi\,| = |\,\eta\,|$, por Lema 75 tenemos:

  $$(\Gamma,\ (\!|\ \bar{n}\ |\!)_{\pi,\ \theta},\ \eta) = (\Gamma, \mathtt{Access}\ n,\ \eta)\ \blacksquare^{\theta}\ \rho\ \swarrow\ n = [\![\ \bar{n}\ ]\!]_{\pi,\ \theta}\rho\ .$$

- Caso (App). Debemos ver $(\Gamma,\ (\!|\ t\ \bar{n}\ |\!)_{\pi,\ \theta'},\ \eta)\ \blacksquare^{\theta'}\ [\![\ t\ \bar{n}\ ]\!]_{\pi,\ \theta'}\rho$. Por hipótesis inductiva tenemos $(\Gamma,\ (\!|\ t\ |\!)_{\pi,\ \theta \to \theta'},\ \eta)\ \blacksquare^{\theta \to \theta'}\ [\![\ t\ ]\!]_{\pi,\ \theta \to \theta'}\ \rho$, donde $\theta = \pi \cdot n$. Luego, por Lema 76, obtenemos:

  $$(\Gamma,\ (\!|\ t\ \bar{n}\ |\!)_{\pi,\ \theta'},\ \eta) = (\Gamma, \mathtt{Push}\ n \triangleright (\!|\ t\ |\!)_{\pi,\ \theta \to \theta'},\ \eta)\ \blacksquare^{\theta'}\ [\![\ t\ ]\!]_{\pi,\ \theta \to \theta'}\ \rho\ (\rho\ \swarrow\ n).$$

- Caso (Unit). Es trivial, pues $(\Gamma, \mathtt{Unit},\ \eta) \in \mathcal{R}^{\mathbf{unit}}(\varnothing)$.

- Caso (Let). Debemos ver $(\Gamma,\ (\!|\ \mathit{let}\ t\ \mathit{in}\ t'\ |\!)_{\pi,\ \theta'},\ \eta)\ \blacksquare^{\theta'}\ [\![\ \mathit{let}\ t\ \mathit{in}\ t'\ ]\!]_{\pi,\ \theta'}\ \rho$. Sea $d = [\![\ t\ ]\!]_{\pi,\ \theta}\rho$ y $d' = [\![\ t'\ ]\!]_{\theta :: \pi,\ \theta'}\ (d,\ \rho)$. Luego $[\![\ \mathit{let}\ t\ \mathit{in}\ t'\ ]\!]_{\pi,\ \theta'}\ \rho = d'$.

Tomemos $(\Delta,\ s) \in \mathcal{T}^{\theta'}(d')$ tal que $wf(\Delta,\ s)$ y $\Gamma \bowtie \Delta$. Debemos demostrar que $(\Gamma \cup \Delta,\ \texttt{Let}\ (\!|\ t\ |\!)_{\pi,\ \theta} \triangleright (\!|\ t'\ |\!)_{\theta::\pi,\ \theta'},\ \eta,\ s) \in \perp\!\!\!\perp$. Sea $\alpha = ((\!|\ t\ |\!)_{\pi,\ \theta},\ \eta)$, por anti-ejecución alcanza con ver $((\Gamma \cup \Delta)[p \mapsto \alpha],\ (\!|\ t'\ |\!)_{\theta::\pi,\ \theta'},\ \eta,\ s) \in \perp\!\!\!\perp$, donde p es un puntero nuevo: $p \notin ptr(\Gamma \cup \Delta) \cup ptr(\eta) \cup ptr(s)$.

Por hipótesis inductiva, tenemos $(\Gamma,\ \alpha)\ \blacksquare^{\theta}\ d$. Como $\Gamma \bowtie \Gamma$ y además se tiene $\Gamma \bowtie \{\, p \mapsto \alpha \,\}$, obtenemos $(\Gamma \cup \{\, p \mapsto \alpha \,\},\ p :: \eta)\ \blacksquare^{\theta::\pi}\ (d,\ \rho)$ por Lema 73. Luego se cumple $(\Gamma \cup \{\, p \mapsto \alpha \,\},\ (\!|\ t'\ |\!)_{\theta::\pi,\ \theta'},\ p :: \eta)\ \blacksquare^{\theta::\pi}\ d'$ por hipótesis inductiva. Es claro que $\Gamma \cup \{\, p \mapsto \alpha \,\} \bowtie \Delta$ y además vale la igualdad $(\Gamma \cup \Delta)[p \mapsto \alpha] = \Gamma \cup \{\, p \mapsto \alpha \,\} \cup \Delta$. Por lo tanto concluimos $((\Gamma \cup \Delta)[p \mapsto \alpha],\ (\!|\ t'\ |\!)_{\theta::\pi,\ \theta'},\ \eta,\ s) \in \perp\!\!\!\perp$.

□

El Teorema 13 permite obtener la corrección del compilador respecto a la semántica denotacional de los términos cerrados con tipo **unit**.

Teorema 14. *Supongamos que* $wf(\Gamma)$ *y* $[] \vdash t : \textbf{\textit{unit}}$*, entonces* $[\![\ t\ ]\!]_{[],\ \textbf{\textit{unit}}}\varnothing = \varnothing$ *y además se cumple* $(\Gamma,\ (\!|\ t\ |\!)_{[],\ \textbf{\textit{unit}}},\ [],\ [])\longmapsto^{*}(\Delta,\ \texttt{Unit},\ \eta,\ [])$ *para algún heap* $\Delta \in Heap$ *y* $\eta \in MEnv$.

Prueba. Definimos

$$\perp\!\!\!\perp = \{\, w \in Conf \mid\ w \longmapsto^{*} (\Delta,\ \texttt{Unit},\ \eta,\ []),\ \Delta \in Heap,\ \eta \in MEnv \,\}\ .$$

Omitimos la demostración de que $\perp\!\!\!\perp$ es efectivamente una instancia válida para el conjunto de observaciones. Como $wf(\Gamma)$ y $(\Gamma,\ [])\ \blacksquare^{[]}\ \varnothing$, por Teorema 13 tenemos $(\Gamma,\ (\!|\ t\ |\!)_{[],\ \textbf{unit}},\ [])\ \blacksquare^{\textbf{unit}}\ [\![\ t\ ]\!]_{[],\ \textbf{unit}}\varnothing = \varnothing$.

Además es claro que $(\Gamma,\ []) \in \mathcal{T}^{\textbf{unit}}(\varnothing)$. Luego $(\Gamma,\ (\!|\ t\ |\!)_{[],\ \textbf{unit}},\ [],\ []) \in \perp\!\!\!\perp$. □

Si se incorpora la detección de black holes junto con la declaración de variables recursiva tal como aparece en la versión original de la máquina de Sestoft [111], se pierde la compatibilidad con la definición de heap *bien formado* puesto que las transiciones no garantizan que todos los punteros de la configuración ocurren en el dominio del heap. Entendemos que demostrar un teorema similar al anterior para la versión recursiva requerirá de un análisis más detallado de las invariantes de la estructura del heap que permitan razonar sobre las referencias circulares que pueden aparecer durante la ejecución.

Durante el doctorado hemos trabajado con la versión original de la máquina y sobre la semántica big-step del lenguaje. En [104] definimos un sistema de tipos alternativo y demostramos que las semántica big-step preserva los tipos del lenguaje. También formalizamos en Coq un resultado [111, Lema 1] donde Sestoft prueba que la generación de punteros nuevos se puede lograr de manera local (sin analizar el árbol completo de evaluación). El código se puede encontrar junto con la formalizaciones que acompañan a esta tesis.

Capítulo 8

Formalización en Coq

Coq [3] es un asistente de demostración que se ha utilizado para formalizar teoremas matemáticos, algoritmos, lenguajes de programación y distintos sistemas lógicos en general. La confianza que el usuario puede depositar en los teoremas demostrados con Coq se basa en las propiedades del Cálculo de Construcciones Inductivas [41, 42], una teoría de tipos dependientes con un enorme poder expresivo tanto para escribir programas como para demostrar propiedades sobre los mismos.

En el área de correcciones de compiladores, se ha utilizado tanto para demostrar la corrección como para definir el compilador en sí mismo, ya que el mecanismo de extracción que posee Coq permite generar código en otros lenguajes funcionales (como Haskell y OCaml) a partir de la formalización. Las demostraciones de corrección pueden ser de gran tamaño dependiendo de cuál es el entorno de ejecución y el lenguaje utilizado, por lo que el uso de un asistente de demostración permite construir las pruebas de manera incremental y modular, contando a su vez con los mecanismos automáticos de búsqueda y generación de pruebas que provee la herramienta.

Todas las demostraciones de corrección del compilador presentadas en esta tesis han sido formalizadas en Coq. La formalización completa consta de aproximadamente 5100 líneas de código destinadas a especificaciones y 8600 líneas dedicadas a demostraciones, según indica la herramienta `coqwc`. En este capítulo pretendemos brindar un panorama general de las formalizaciones y sólo comentamos algunos fragmentos relevantes del código. Invitamos al lector a consultar la formalización completa en el sitio `https://cs.famaf.unc.edu.ar/~leorodriguez/tesis.html`. A continuación daremos una breve introducción a Coq con el objetivo de facilitar la lectura de los fragmentos de código que se muestran a lo largo del capítulo.

8.1 Una breve introducción a Coq

Coq es una herramienta diseñada para escribir programas y demostrar propiedades sobre ellos de manera interactiva con el usuario. Visto como un lenguaje de programación, Coq es un lenguaje funcional con *tipos dependientes*, que como veremos, permiten escribir especificaciones completas de los programas y sus propiedades. En esta sección introduciremos Coq sin adentrarnos en su formalismo lógico subyacente [41, 42].

Tipos de datos y funciones

Como un primer ejemplo, veamos la definición en Coq del tipo de los números naturales.

```
Inductive nat : Set :=
 | O : nat
 | S : nat -> nat.
```

El tipo `nat` tiene dos constructores: `O` que representa el número 0, y `S` que representa el sucesor de otro número natural. Por ejemplo, el número 3 se escribe como `S (S (S O))`.

En Coq, todas las definiciones tienen un tipo. El término `O` tiene tipo `nat`, y a su vez `nat` tiene tipo `Set`. El tipo de un término se puede verificar usando el comando `Check`.

```
Check O.
(* 0 : nat *)
Check nat.
(* nat : Set *)
```

Aquí mostramos con un comentario en el formato `(* output *)` a la salida que produce la herramienta luego de ejecutar el comando.

Podemos definir el predecesor de un número natural usando *pattern matching*, separando la definición en un caso por cada constructor del tipo `nat`:

```
Definition pred (n : nat) : nat :=
 match n with
  | O => O
  | S n => n
 end.
```

Otro ejemplo es la función factorial, que se puede definir recursivamente usando `Fixpoint` de la siguiente manera:

```
Fixpoint fact (n : nat) : nat :=
 match n with
  | O => 1
  | S n => S n * fact n
 end.
```

Las definiciones de Coq deben ser *totales*, esto es, todo pattern matching debe ser exhaustivo (todos los constructores deben estar cubiertos) y debe existir algún parámetro estructuralmente decreciente en todas las llamadas recursivas. En el caso de `fact`, cuando el argumento es `S n`, la llamada recursiva se realiza con el argumento `n`, luego el parámetro `n` es decreciente. La restricción de totalidad está fuertemente vinculada con la capacidad de Coq para comportarse como asistente de demostración [46].

Jerarquía de tipos

Como habíamos adelantado, en Coq se pueden especificar proposiciones sobre los términos y los tipos de datos. Las proposiciones se definen estipulando las formas de probarla; así, por ejemplo, la proposición constantemente falsa no tiene ningún constructor:

```
Inductive False : Prop := .
```

mientras que hay una única forma canónica de probar la proposición constantemente verdadera:

```
Inductive True : Prop :=  I : True .
```

Los conectivos lógicos también están definidos de esa manera (el lector interesado puede tratar de pensar cuántos constructores tiene cada conectivo) y nos permiten combinar proposiciones.

```
Definition prop1 : Prop :=  True -> False.
Definition prop2 : Prop :=  True /\ prop1.
Definition prop3 : Prop :=  prop1 \/ prop2.
```

Como otro ejemplo, consideremos la función `is_positive` que determina si un número natural es o no positivo (mayor que 0).

```
Definition is_positive (n : nat) : Prop :=
 match n with
  | O => False
  | S _ => True
 end.
Check is_positive.
(* is_positive : nat -> Prop *)
```

A grandes rasgos, el tipo `Set` puede pensarse como el tipo de todos los programas, es decir, de todas las funciones o tipos de datos con algún contenido computacional. El tipo `Prop` contiene a todas las proposiciones, que expresan propiedades sobre los programas, pero no tienen contenido computacional. Tanto `Set` como `Prop` tienen tipo `Type`. El tipo `Type` es en realidad una jerarquía de tipos `Type(n)` donde `n` es un número entero. Esta jerarquía infinita de tipos es necesaria para evitar inconsistencias en la lógica subyacente de Coq [39].

Tipos dependientes

Regresando a la función `pred`, la definición de `pred 0` es difícil de justificar desde un punto de vista intuitivo ya que no hay una definición natural para el predecesor del número 0 dentro del rango de los números naturales. Para evitar ese inconveniente, podemos usar la función `is_positive` para definir la función predecesor sólo para números positivos.

```
Definition pred_dep (n : nat) : is_positive n -> nat :=
 match n with
  | O => fun zero_is_positive =>
              match zero_is_positive with
              end
  | S n => fun _ => n
 end.
Check pred_dep.
(* forall n : nat, is_positive n -> nat *)
```

El segundo argumento de `pred_dep` es una prueba de que el número `n` es positivo. La función `pred_dep` se define usando pattern matching sobre `n`, en ambos constructores el tipo del resultado es `is_positive n -> nat`. En el caso del constructor `0`, el término `zero_is_positive` tiene tipo `is_positive 0` que fue definido como `False`. Como el tipo `False` no tiene constructores, hacer pattern matching sobre `zero_is_positive` concluye la definición, pues no es posible construir una prueba de `is_positive 0`. En el caso del constructor `S`, la prueba de `is_positive (S n)` es ignorada, por lo que se escribe el guión bajo en el término `fun _ => n`.

El tipo de `pred_dep` es *dependiente* ya que justamente depende del valor de `n`, esto es, el tipo del término `pred_dep 0` es distinto al tipo de `pred_dep 1`. Los constructores de un tipo inductivo también pueden tener tipos dependientes, por ejemplo, el tipo `le` que define el orden usual de los números naturales. La proposición `le n m` (cuya notación es `n <= m`) establece que `n` es menor o igual que `m`.

```
Inductive le (n : nat) : nat -> Prop :=
 | le_n : n <= n
 | le_S : forall m : nat, n <= m -> n <= S m.

Definition example_leq : 2 <= 4 := le_S _ _ (le_S _ _ (le_n 2)).
```

Sistema de tácticas

Los tipos dependientes permiten escribir una enorme variedad de propiedades que no son expresables directamente en otros lenguajes funcionales como Haskell [1]. Como un ejemplo, el tipo dependiente `forall n, pred (S n) = n` permite especificar la función `pred` para cualquier número positivo.

```
Lemma pred_prop : forall n, pred (S n) = n.
Proof.
  intro n.
  (*
    n : nat
    ============================
    pred (S n) = n
  *)
  simpl.
  (*
    n : nat
    ============================
    n = n
  *)
  constructor.
Qed.
```

La demostración de ese lema se construye de manera interactiva con el usuario. Luego de la definición del lema, la herramienta muestra el objetivo (o *goal*) de la demostración que en nuestro caso es:

```
 (*
   ============================
   forall n, pred (S n) = n
 *)
```

En la prueba usamos tres *tácticas* `intro`, `simpl` y `constructor` que operan (a grandes rasgos) como se describe a continuación. La táctica `intro` introduce la hipótesis `n : nat` eliminando la cuantificación `forall n`. La táctica `simpl` intenta simplificar el objetivo desplegando definiciones, en este caso `pred (S n)` se puede igualar a `n` por definición. La táctica `constructor` intenta aplicar un

constructor para demostrar el objetivo; en este caso el único constructor de la igualdad es `eq_refl`:

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  eq_refl : x = x.
```

La proposición del lema anterior es lo suficientemente simple como para que la táctica `auto` pueda completar la demostración sin necesidad de mayor interacción:

```
Lemma pred_prop' : forall n, pred (S n) = n.
Proof.
  auto.
Qed.
```

Como otro ejemplo de proposición, podemos probar que ambas definiciones de la función predecesor coinciden para números positivos.

```
Lemma same_pred:
  forall n, forall q:is_positive n,  pred n = pred_dep n q.
Proof.
  intro n.
  case n.
  intro q.
  contradiction.
  intro q.
  simpl.
  auto.
Qed.
```

Por brevedad, omitimos la explicación específica de cada táctica, que pueden consultarse en el manual de usuario [4]. Las aplicación de tácticas construye internamente la definición de un término que puede observarse usando el comando `Print`.

```
Print same_pred.
 (*
  same_pred =
  fun n : nat =>
  match n as n0
   return (forall q : is_positive n0, pred n0 = pred_dep n0 q) with
   | 0 => fun q : is_positive 0 => False_ind (pred 0 = pred_dep 0 q) q
   | S q => fun _ : True => eq_refl
  end
 *)
```

Si bien uno podría construir ese término manualmente, el sistema de tácticas permite hacerlo de manera interactiva. Esta es una característica distintiva de Coq frente a otros asistentes de demostración como Agda [2].

Proposiciones y programas

A pesar de que un término de tipo `Prop` no tiene contenido computacional, las proposiciones tienen su definición análoga en el tipo `Set`. Este hecho está estrechamente relacionado con la correspondencia de Curry-Howard [45, 60], que establece una equivalencia entre los programas y las pruebas. Por ejemplo, podríamos definir una versión en `Set` de la función `is_positive`,

```
Definition is_positiveT (n : nat) : Set :=
 match n with
  | O => Empty_set
  | S _ => unit
 end.
```

El tipo `Empty_set`, al igual que `False`, no tiene constructores. Por otro lado, el tipo `unit`, al igual que `True`, tiene un único constructor.

```
Inductive Empty_set : Set := .
Inductive unit : Set := tt : unit.
```

La diferencia principal entre `Set` y `Prop` es que `Prop` es *impredicativo*, esto es, una cuantificación sobre términos en `Prop` es nuevamente de tipo `Prop`. En cambio, una cuantificación sobre `Set` tiene un tipo más elevado en la jerarquía de tipos `Type`. El hecho de que Coq provea el tipo `Prop` permite formalizar demostraciones basadas en razonamientos impredicativos [8, 9]. Otra diferencia práctica entre los dos tipos está determinada por el mecanismo de extracción, que veremos a continuación.

Extracción

El mecanismo de extracción de Coq permite traducir automáticamente programas en Coq a otros lenguajes de programación. Por ejemplo, la traducción a Haskell de `pred_dep` se obtiene mediante los siguientes comandos:

```
Extraction Language Haskell.
Extraction pred_dep.
(* -- Haskell code:
  pred_dep :: Nat -> Nat
  pred_dep n =
   case n of {
    O -> Prelude.error "absurd case";
    S n0 -> n0
*)
```

Observemos que el parámetro `is_positive n` ha sido eliminado completamente en la versión de `pred_dep` traducida a Haskell. Esto sucede porque `is_positive n` tiene tipo `Prop`. En cambio, consideremos la siguiente función:

```
Definition pred_depT (n : nat) : is_positiveT n -> nat :=
 match n with
  | O => Empty_set_rect _
  | S n => fun _ => n
 end.
```

Aquí usamos la versión en `Set` de `is_positive`. La extracción de esta función resulta en el siguiente programa:

```
Extraction pred_depT.
(*
  -- Haskell code:
  pred_depT :: Nat -> Is_positiveT -> Nat
  pred_depT n =
   case n of {
    O -> unsafeCoerce empty_set_rect;
    S n0 -> (\_ -> n0)}
*)
```

Notemos que el segundo argumento tiene tipo `Is_positiveT`, que resulta de la extracción de la función `is_positiveT`.

Coinducción

En Coq también es posible trabajar con definiciones coinductivas [40]. Por ejemplo, el tipo `natinf` es la interpretación coinductiva de los constructores `O` y `S`.

```
CoInductive natinf : Set :=
 | O : natinf
 | S : natinf -> natinf.
```

Con el tipo `natinf` se representa al conjunto $\mathbb{N} \cup \{ \infty \}$ donde ∞ se puede definir como la aplicación infinita del constructor `S`.

```
CoFixpoint infty : natinf := S infty.
```

Así como las llamadas recursivas de Coq deben tener un argumento estructuralmente decreciente, las llamadas corecursivas deben *extender* el resultado aplicando alguno de los constructores. En el caso de `infty`, la llamada corecursiva aparece como argumento del constructor `S`.

Combinando tipos coinductivos con tipos dependientes, definimos el tipo `le_inf`, una versión coinductiva del tipo `le`.

```
CoInductive le_inf : natinf —> natinf —> Prop :=
 | le_inf_n : forall n, n ≤∞ n
 | le_inf_S : forall n m, n ≤∞ m —> n ≤∞ S m
where "n ≤∞ m" := (le_inf n m).
```

Se puede demostrar que todo `n : natinf` es menor o igual a `infty`.

```
Lemma infty_le:
  forall n, n ≤∞ infty.
Proof.
 cofix.
 (* ... *)
 Guarded.
Qed.
```

La táctica `cofix` comienza una prueba por coinducción. Con `(* ... *)` indicamos que estamos omitiendo un segmento de la prueba. El comando `Guarded` permite verificar que la hipótesis coinductiva siempre se utiliza como argumento de un constructor de un tipo coinductivo. Se pueden encontrar más ejemplos de tipos coinductivos en [20].

En esta sección solo hemos intentado brindar una breve introducción al lector que no conozca Coq para facilitar la lectura de las secciones siguientes. Referimos a Bertot et al. [22], Chlipala [35] o Pierce et al. [92] para mayor información sobre la herramienta.

8.2 El cálculo lambda y su semántica

Con el objetivo de ilustrar cómo hemos traducido las definiciones de la tesis a la sintaxis de Coq, comentaremos sobre la formalización del Lema 1, que es una correspondencia entre la semántica big-step con entornos y la semántica big-step con sustituciones. La representación básica del cálculo lambda puro, con variables nombradas (sin usar índices de Bruijn) puede definirse usando un tipo inductivo con tres constructores: variable, abstracción y aplicación.

Términos (Definición 1)

```
Inductive term : Set :=
 | term_var : var —> term
 | term_abs : var —> term —> term
 | term_app : term —> term —> term.
```

En nuestro caso, el tipo `var` es un sinónimo de `nat`, el tipo de los números naturales. Sin embargo, en esta formalización, el tipo `var` podría definirse de cualquier otra manera siempre y cuando se pueda demostrar que la igualdad es decidible:

```
Definition var_eq_dec : forall x y : var, {x = y} + {x <> y}.
```

La notación {A} + {B} se refiere al tipo `sumbool A B`:

```
Inductive sumbool (A B : Prop) : Set :=
| left : A -> {A} + {B}
| right : B -> {A} + {B}.
```

Observemos que demostrar `var_eq_dec` sería trivial usando lógica clásica mediante el axioma del *tercero excluido*. A pesar de que ese axioma podría postularse en la formalización, preferimos mantener las demostraciones puramente *constructivas* restringiendo el razonamiento a la lógica interna de Coq.

En `var_eq_dec` usamos la igualdad `Logic.eq` (igualdad de Leibniz) por compatibilidad con la librería `Ensembles`, que utilizamos para representar conjuntos de variables.

Definimos la función `substi` que representa la sustitución de una variable por un término cerrado. El término `substi t x t0` es el resultado de reemplazar todas las ocurrencias libres de `x` en el término `t` por el término cerrado `t0`. Esta función se define de forma recursiva usando pattern matching sobre el término `t`. Notemos que `var_eq_dec` se utiliza en el caso del constructor `term_abs` para verificar si la variable a reemplazar es igual a la variable ligada por la propia abstracción. Se utiliza también en el caso `term_var y` para verificar si la variable a reemplazar `x` es igual a la variable libre `y`.

```
Fixpoint substi (t : term) (x : var) (t0 : term) : term :=
  match t with
   | term_abs y t' =>
      match var_eq_dec x y with
       | left _  => term_abs y t'
       | right _ => term_abs y (t' [x \ t0])
      end
   | term_app t1 t2 =>
      term_app (t1 [x \ t0]) (t2 [x \ t0])
   | term_var y =>
      match var_eq_dec x y with
       | left _  => t0
       | right _ => term_var y
      end
  end
where "t [ x \ t' ]" := (substi t x t').
```

El tipo `Ensemble` de la librería estándar se define como sigue:

```
Definition Ensemble (U : Type) := U -> Prop.
```

Un término `X : Ensemble U` representa al conjunto de elementos `e : U` para los cuales puede demostrarse `X e`. Es decir, `X` es un predicado sobre `U` que caracteriza a los elementos del conjunto. Como un ejemplo, el conjunto de las variables libres de un término se define como sigue:

Variables libres (Definición 2)

```
Fixpoint fv (t : term) : Ensemble var :=
  fun y =>
  match t with
      | term_var x => x = y
      | term_abs x t => x <> y /\  fv t y
      | term_app t1 t2 => fv t1 y \/  fv t2 y
    end.
```

El tipo `dvalue` representa a los valores que pueden ser el resultado de una evaluación, que en este caso son únicamente abstracciones. El primer parámetro del constructor es la variable ligada, y el segundo parámetro es el cuerpo de la abstracción.

Valores (Definición 5)

```
Inductive dvalue : Type :=
 | dvalue_abs : var -> term -> dvalue.
```

El prefijo "d" en el nombre del tipo `dvalue` hace referencia a que la definición corresponde a la evaluación *directa* usando sustituciones (versus la evaluación usando entornos).

El conjunto de evaluaciones se representa mediante una familia de tipos indexada por el término inicial y su valor. Para cada regla de la Definición 6 habrá un constructor de esta familia.

Reglas de evaluación (Definición 6)

```
Inductive deval : term -> dvalue -> Prop :=
 | deval_abs :
    forall x t,  term_abs x t ⟹ dvalue_abs x t
 | deval_app :
    forall t1 t2 y t' v,
     t1 ⟹ dvalue_abs y t' ->
     t' [y \ t2] ⟹ v ->
     term_app t1 t2 ⟹ v
where "t ⟹ v" := (deval t v).
```

De manera similar, definimos las reglas de evaluación big-step con entornos. Esta vez, los valores son abstracciones equipadas con un entorno.

Valores (Definición 8)

```
Inductive value :=
 | value_abs : var -> term -> env -> value.
```

Reglas de evaluación con entornos (Definición 9)

```
Inductive eval : env -> term -> value -> Prop :=
 | eval_abst :
      forall e x t,
        e ⊢ term_abs x t ⇨ value_abs x t e
 | eval_app :
      forall e y t1 t2 t' e' v,
        e ⊢ t1 ⇨ value_abs y t' e' ->
        (y ↦ (t2, e) :: e') ⊢ t' ⇨ v ->
        e ⊢ term_app t1 t2 ⇨ v
 | eval_var :
      forall e x v t' e',
      forall (q : dom_env e x),
        env_lookup e x q = {t', e'} ->
        e' ⊢ t' ⇨ v ->
        e ⊢  term_var x ⇨ v
where "e ⊢ t ⇨ v" := (eval e t v).
```

En el constructor `eval_var`, el término `q : dom_env e x` es una prueba de que la variable `x` está en el dominio del entorno `e`, y el término `env_lookup e x q` es la clausura asociada con `x` en el entorno `e`.

El predicado `fv_compat t e` establece que todas las variables libres de `t` están en el dominio del entorno `e`. Usamos `Ensemble.Include` para representar la inclusión de conjuntos.

```
Definition fv_compat (t :term) (e : env) : Prop
:= Included _ (fv t) (dom_env e).
```

Por otro lado, el predicado recursivo `well_formed e` establece que para toda clausura `(t, e0)` en el rango de `e` se cumple `fv_compat t e0` y `well_formed e0`. Se define por inducción estructural en el entorno:

```
Fixpoint well_formed (e : env) :=
  match e with
    {*}  => True
  | (y ↦ (t, e0)) :: e1 => well_formed e0 /\ well_formed e1 /\ fv_compat t e0
  end.
```

La función `flat` realiza una sustitución sucesiva de las variables libres de un término, reemplazándolas por el contenido de su entorno. Cuando se cumple `fv_compat t e` y `well_formed e`, el resultado de `flat t e` es un término cerrado.

Aplanamiento (Definición 10)

```
Fixpoint flat (t : term) (e : env) : term :=
  match e with
    | {*}  => t
    | (y ↦ (t', e0)) :: e1 =>
       t [ y \ t' # e0 ] # e1
  end
where "t # e" := (flat t e).

Lemma flat_fv_compat:
 forall e, well_formed e -> forall t, fv_compat t e -> closed (t # e).
```

Es posible usar la función `flat` para convertir un valor de tipo `value` a un valor de tipo `dvalue`:

```
Definition flat_value (v : value) : dvalue :=
  match v with
      value_abs x t e => dvalue_abs x (t # (env_delete e x))
  end.

Notation "#[ v ]" := (flat_value v) (at level 5, no associativity).
```

La función `env_delete` elimina una variable del dominio del entorno. En la definición de `flat_value` se usa para evitar la sustitución de la variable ligada `x` en el cuerpo de la abstracción. El comando `Notation` permite definir notaciones con distinto nivel de precedencia y asociatividad. También es posible definir notaciones en simultáneo con la definición de un tipo; para ello se utiliza `where` al final de dicha definición.

Ahora podemos enunciar el lema que relaciona la semántica big-step con entornos y la semántica big-step directa.

Lema 1

```
Lemma eval_deval:
forall e t v,
    e ⊢ t ⇨ v ->
    well_formed e ->
    fv_compat t e ->
    t # e ⟹ #[v].
Proof.
    induction 1.
    (* ... *)
Qed.
```

Aquí la táctica `induction 1` indica que la prueba se realiza por inducción estructural en las reglas de evaluación. Los puntos suspensivos indican que se omite el resto de la demostración.

8.3 Semántica small-step

En esta sección comentaremos la formalización de los Teoremas 1 y 2, la corrección de un compilador respecto a la semántica small-step del cálculo de clausuras, utilizando las instrucciones de la máquina de Krivine como lenguaje intermedio. Los términos del cálculo de clausuras se definen mediante los siguientes tipos de datos inductivos:

Cálculo de clausuras (Definición 20)

```
Inductive term :=
 | term_var : var -> term
 | term_abs : term -> term
 | term_app : term -> term -> term.

Inductive closure :=
 | closure_term : term -> list closure -> closure
 | closure_app : closure -> closure -> closure.

Notation "t [ e ]" := (closure_term t e) (at level 40, no associativity).
Definition env := list closure.
```

Las variables se representan mediante índices de *De Bruijn*. Notar que el constructor `term_var` toma como argumento al índice de la variable (un número natural).

El constructor `closure_term` toma dos argumentos: el término y el entorno de la clausura. El constructor `closure_app` representa la aplicación de dos clausuras.

Las reglas de contracción se definen con un tipo inductivo, donde cada constructor representa una de las reglas.

Reglas de contracción (Definición 21)

```
Inductive step : closure -> closure -> Prop :=
 | step_beta : forall t e c,
    closure_app ((term_abs t) [e]) c → t [c :: e]
 | step_nu  : forall c0 c1 c2,
    c0 → c1 ->
    closure_app c0 c2 → closure_app c1 c2
 | step_app : forall t t' e,
```

```
    (term_app t t') [e] → closure_app (t [e]) (t' [e])
 | step_var : forall n t' e e' ,
    lookup e n = Some (t' [e']) —>
    (term_var n) [e] → t' [e']
where "c → c'" := (step c c').
```

En `step_var` usamos el constructor `Some` del tipo `option`:

```
Inductive option (A : Type) :=  Some : A —> option A | None : option A.
```

El tipo `option` se utiliza usualmente para definir funciones parciales dentro de la lógica de funciones totales de Coq (similar al tipo `Maybe` de Haskell). La proposición `lookup e n = Some (t' [e'])` establece que la clausura en la n-ésima posición de e está definida y es igual a `t' [e']`. Esto es diferente a lo que usamos en el constructor `eval_var` que vimos en la sección anterior, donde el hecho de que la clausura en la posición n está definida dentro del entorno viene demostrado como en un argumento adicional de `env_lookup`.

A continuación definimos las componentes de la máquina de Krivine (código, clausuras de máquina, entornos de máquina, pilas y configuraciones).

Componentes de la máquina de Krivine (Definición 26)

```
Inductive code :=
 | igrab : code —> code
 | ipush : code —> code —> code
 | iaccess : nat —> code.
Inductive mclosure :=
| mclosure_code : code —> list mclosure —> mclosure.

Definition menv := list mclosure.
Definition stack := list mclosure.
Definition conf := mclosure * stack.
```

Las transiciones de la máquina abstracta (una para cada instrucción) se definen como sigue:

Reglas de transición (Definición 27)

```
Inductive trans : conf —> conf —> Prop :=
 | trans_grab :
    forall i e α (s : stack),
      let αo := mclosure_code (igrab i) e in
      let α1 := mclosure_code i (α :: e) in
      (αo , α :: s) ↦ (α1 , s)
 | trans_push :
    forall i i' e (s : stack),
      let αo := mclosure_code (ipush i' i) e in
```

```
      let α1 := mclosure_code i e in
      let α  := mclosure_code i' e in
      (αo, s) ↦ (α1 , α :: s)
 | trans_access :
    forall e n α (s : stack),
      lookup e n = Some α —>
      let αo := mclosure_code (iaccess n) e in
      (αo, s) ↦ (α, s)
where "w ↦ w'" := (trans w w').
```

Sabemos que cada regla de transición representa un único paso de ejecución. Sin embargo, necesitamos también alguna manera de representar secuencias con múltiples transiciones. El tipo inductivo `star` y el tipo coinductivo `infseq` definen respectivamente secuencias finitas e infinitas donde los elementos consecutivos de una secuencia deben estar relacionados mediante `R`. En nuestro caso, el argumento `R` será el tipo `step` o el tipo `trans` para representar respectivamente secuencias de contracciones o de transiciones. El tipo `irr` corresponde con la noción de elemento bloqueante (Definición 22) y el tipo `converges` a la noción de convergencia (Definición 23).

```
Section Sequences.
  Variable A: Type.
  Variable R: A —> A —> Prop.
  Inductive star: A —> A —> Prop :=
   | star_refl: forall a, star a a
   | star_step: forall a b c,
    R a b —> star b c —> star a c.

  CoInductive infseq: A —> Prop :=
   | infseq_step: forall a b,
    R a b —> infseq b —> infseq a.
  Definition moves a : Prop := exists b, R a b.
  Definition irr a : Prop := ~ moves a.
  Definition converges a b := star a b /\ irr b.
End Sequences.
```

El mecanismo de secciones `Section` de Coq se utiliza usualmente para evitar la repetición de los parámetros comunes de un conjunto de definiciones. Nuestra definición en Coq del tipo de las secuencias finitas está basada en una librería escrita por Leroy [77]. En la formalización completa se demuestran algunas propiedades de estas secuencias. La compilación de los términos del cálculo de clausuras a las instrucciones de la máquina de Krivine se define mediante una función recursiva:

Compilación de términos (Definición 28)

```
Fixpoint compile (t : term) : code :=
  match t with
    | term_abs t => igrab (compile t)
    | term_app t1 t2 => ipush (compile t2) (compile t1)
    | term_var n => iaccess n
  end.
```

Por brevedad, omitimos la definición en Coq de las funciones de decompilación (de términos, de clausuras y de configuraciones). Los siguientes son los enunciados de los teoremas de corrección, tanto para términos convergentes como para divergentes.

Teoremas 1 y 2

```
Theorem correctness_for_closed_convergent :
  forall t c,
  converges step (t [nil]) c —>
  exists w',
  converges trans (init t) w' /\ decompile_conf w' = c.

Theorem correctness_for_closed_divergent :
  forall t,
  infseq step (t [nil]) —>
  infseq trans (init t).
```

El término `t` se asume cerrado y por ello la clausura `t [nil]` tiene un entorno vacío. El primer teorema establece que si `t [nil]` converge a una clausura `c` entonces la ejecución de `init t := (mclosure (compile t) nil, nil)` (tanto el entorno como la pila están inicialmente vacías) converge a una configuración que decompila a `c`. Por otro lado, cuando `t [nil]` realiza infinitas contracciones, la configuración `init t` realiza infinitas transiciones.

8.4 Semántica big-step

En el Capítulo 4 demostramos la corrección del compilador de un lenguaje imperativo con respecto a la semántica big-step. Ilustraremos la formalización correspondiente con algunos fragmentos del código.

El tipo `eval` posee un constructor por cada regla de evaluación, aquí mostramos únicamente el constructor correspondiente a la asignación.

Semántica big-step inductiva (Definición 47)

```
Inductive eval (e : env) (σ : state) : term —> value —> Prop :=
 | eval_assign : forall t1 l t2 k,
```

```
    e ~ σ  ⊢ t1 ⇨ value_loc l −>
    e ~ σ  ⊢ t2 ⇨ value_nat k −>
    e ~ σ  ⊢ term_assign t1 t2 ⇨ value_state (replace σ l k)
(* ... *)
where "e ~ σ  ⊢ t ⇨ v" := (eval e σ t v).
```

En la definición de `eval_assign`, los constructores `value_loc`, `value_nat` y `value_state` representan respectivamente a las localizaciones de memoria, las constantes numéricas y los estados, vistos como valores resultantes de la evaluación. El término `t1` evalúa a una localización `l`, y el término `t2` evalúa a una constante `k`. La asignación `term_assign t1 t2` produce como resultado el estado que se obtiene al modificar en σ el valor de la posición `l` por la constante `k`.

A continuación mostramos los constructores correspondientes a las reglas coinductivas de la asignación:

Semántica big-step coinductiva (Definición 50)

```
CoInductive diverges (e : env) (σ : state) : term −> Prop :=
 | diverges_assign_fst :
    forall t1 t2,
      e ~ σ ⊢ t1 ⇨∞ −>
      e ~ σ ⊢ term_assign t1 t2 ⇨∞
 | diverges_assign_snd :
    forall t1 l t2,
      e ~ σ ⊢ t1 ⇨ value_loc l −>
      e ~ σ ⊢ t2 ⇨∞ −>
      e ~ σ ⊢ term_assign t1 t2 ⇨∞
(* ... *)
where "e ~ σ ⊢ t ⇨∞" := (diverges e σ t).
```

El constructor `diverges_assign_fst` representa el caso de una asignación que diverge debido a que el término del lado izquierdo diverge. Por otro lado, en el constructor `diverges_assign_snd`, el término izquierdo converge a una localización, y el término derecho diverge.

Los términos del lenguaje fuente son traducidos por el compilador a las instrucciones de una máquina abstracta. En la Sección 4.2 presentamos una extensión de la máquina de Krivine que soporta construcciones imperativas. A modo de ejemplo, presentamos la definición en Coq de la regla de transición para el operador de asignación.

Reglas de transición (Definición 53)

```
Inductive trans : conf −> conf −> Prop :=
 | trans_assign :
     forall e σ (s : stack) l k,
      let αo := closure_code (iop operator_assign) e in
```

```
      let vs := value_loc l :: value_nat k :: nil in
      let f  := frame_op operator_assign vs nil in
      let α1 := closure_code icont e in
      (α0, σ, stack_val_frame f :: s ) ↦ (α1, replace σ l k, s )
where "w ↦ w'" := (trans w w').
```

Recordemos que para lograr la implementación de operadores estrictos utilizabamos *frames*. La función `frame_op` construye un frame para un operador a partir de una lista de argumentos ya computados y una lista de clausuras que deben ejecutarse para obtener el valor de los argumentos restantes. Dado que el constructor `trans_assign` asume que ya se conocen los argumentos de la asignación, el último argumento de `frame_op` es una lista vacía.

Los teoremas de corrección (para términos convergentes y divergentes) se definieron de la siguiente manera:

Teoremas 6 y 7

```
Theorem correctness_for_convergent :
  forall e σ t v,
    e ~ σ   ⊢ t ⇨ v ->
    forall s,
      let α := closure_code (compile t) (compile_env e) in
      (α, σ, s ) ↣ v.

Theorem correctness_for_divergent :
  forall e σ t,
    e ~ σ  ⊢ t ⇨∞ ->
    forall s,
      let α := closure_code (compile t) (compile_env e) in
      infseq trans (α, σ, s ).
```

En `correctness_for_convergent` hemos usado la relación $\rightarrowtail$ de *alcanzabilidad* (Definición 57) cuya definición en Coq hemos omitido. La definición de la función de compilación se deja también para consultar en la formalización completa.

8.5 Semántica denotacional

En el Capítulo 6 demostramos la corrección de un compilador usando relaciones lógicas de aproximación. La formalización correspondiente ese capítulo incluye la definición de la semántica denotacional del lenguaje, la definición de las relaciones lógicas, la demostración de varias propiedades sobre esas relaciones, y finalmente la prueba de corrección del compilador.

Hemos utilizado como base de la formalización una librería de teoría de dominios desarrollada por Benton et al. [18, 19], a partir de la cual hemos formalizado la semántica denotacional del lenguaje fuente. En esa librería se definen distintas estructuras matemáticas siguiendo el estilo *mixin* [53] que permite combinar de manera jerárquica los diferentes tipos y axiomas involucrados en dichas estructuras. La librería hace un uso intensivo de clases [113] y estructuras canónicas [80] en Coq para lograr un mecanismo de coerciones automáticas entre las distintas estructuras algebraicas.

A modo de ejemplo consideremos la formalización de de conjuntos *preordenados* que brinda la librería:

Conjunto preordenado (fragmento de la librería)

```
Module PreOrd.
Definition axiom T (Ole : T -> T -> Prop) :=
forall x , Ole x x /\ forall y z, (Ole x y -> Ole y z -> Ole x z).
Record mixin_of T := Mixin {Ole : T -> T -> Prop; _ : axiom Ole}.
Notation class_of := mixin_of (only parsing).
Lemma setAxiom T (c: mixin_of T)
: Setoid.axiom (fun x y => Ole c x y /\ Ole c y x).
(* ... *)
Qed.
Structure type := Pack {sort :> Type; _ : class_of sort; _ : Type}.
(* ... *)
End PreOrd.
Notation ordType := PreOrd.type.
```

Aquí el tipo `T` es el el tipo de los elementos del conjunto y `Ole` es la relación de orden. Como indica `axiom`, para demostrar que `T` es preordenado se debe probar que `Ole` que es reflexivo y transitivo. El record `mixin_of T` combina la relación de orden `Ole` junto con la demostración de `axiom T Ole`. La estructura `type` es básicamente una versión auto-contenida de `mixin_of` donde el parámetro `T` pasa a ser el miembro `sort` del record.

La igualdad `x == y` en `T` se define como `Ole x y /\ Ole y x`, y en `setAxiom` se demuestra que es una relación de equivalencia. La antisimetría de `Ole` respecto a la igualdad `==` es trivial por definición, con lo que podemos ver a `T` y `Ole` como un poset. A partir de `PreOrd` se define el módulo `CPO` (por *complete partial order*, que nosotros llamamos *predominio* en la Definición 88) que demanda la existencia del supremo para todas las cadenas.

Predominio (fragmento de la librería)

```
Module CPO.
Definition axiom (T:ordType) (lub : (natO =-=> T) -> T) :=
  forall (c:natO =-=> T) x n, (c n <= lub c)
```

```
  /\  (( forall n,  c n <= x)  —> lub c <= x).
Record mixin_of (T:ordType) :  Type :=
  Mixin {lub:  (nat0 =—> T) —> T; _ :  axiom lub }.
Record class_of (T:Type) :  Type :=
  Class {base :> PreOrd.class_of T; ext :> mixin_of (PreOrd.Pack base T) }.

Structure type : Type := Pack {  sort :> Type; _ : class_of sort ; _ : Type}.
  (* ... *)
End CPO.
Notation cpoType := CPO.type.
```

Aquí el tipo `T` es un conjunto preordenado, y `lub` es una función que asigna a cada cadena de `T` su supremo. Las cadenas son representadas como funciones monótonas de `nat0` a `T` donde `nat0` es el conjunto preordenado de los números naturales junto con el orden usual. El record `mixin_of` combina la función `lub` junto con la prueba de que `lub c` es efectivamente la menor cota superior de toda cadena `c`.

De manera similar se puede definir el tipo `cppoType` de los *dominios* (predominios con menor elemento), aunque hemos omitido aquí el fragmento de código correspondiente.

A partir de un predominio $\langle P, \leqslant \rangle$ se puede construir el dominio $\langle P_\perp, \leqslant_\perp \rangle$ donde $P_\perp = \{ \iota_\uparrow p \mid p \in P \} \cup \{ \perp \}$ y

- para todo $p, p' \in P$, $\iota_\uparrow p \leqslant_\perp \iota_\uparrow p'$ sí y sólo sí $p \leqslant p'$,
- para todo $p \in P$, $\perp \leqslant_\perp \iota_\uparrow p$ y
- $\perp \leqslant_\perp \perp$.

Claramente $\perp$ es el menor elemento del $P_\perp$. El dominio $P_\perp$ denomina *lifting* de P. Es claro que toda cadena $q_0 \leqslant_\perp q_1 \leqslant_\perp q_2 \leqslant \ldots$ de $P_\perp$ cumple una de las siguientes condiciones:

1. para todo $i \in \mathbb{N}$, $q_i = \perp$,
2. existe $i \in \mathbb{N}$, tal que para todo $j \geq i$, $q_j = \iota_\uparrow p_j$ y además la secuencia $p_i \leqslant p_{i+1} \leqslant p_{i+2} \leqslant \ldots$ es una cadena de P.

En el primer caso, el supremo de la cadena es $\perp$. En el segundo, el supremo es $\iota_\uparrow p$ donde $p = \bigsqcup_{i \in \mathbb{N}}^{\infty} p_i$. Lo anterior demuestra que $P_\perp$ es efectivamente un predominio, aunque el razonamiento aplicado es clásico: el axioma del tercero excluido se usa para determinar si vale 1 o bien 2. Esa situación es explícita en Coq cuando se intenta implementar a $P_\perp$ como `option` P donde `None` $== \perp$ y `Some` $p == \iota_\uparrow p$, pues resulta necesario postular axiomas clásicos para completar

las demostraciones [23]. Alternativamente, la representación de $P_\perp$ se puede lograr usando un tipo de datos coinductivo:

Streams (fragmento de la librería)

```
Variable D : Type.
CoInductive Stream := Eps : Stream -> Stream  | Val : D -> Stream.
CoFixpoint DL_bot := Eps DL_bot.
```

El conjunto subyacente de $P_\perp$ se puede definir como `Stream` P. El menor elemento $\perp$ es representado con la secuencia infinita `DL_bot`, y los demás elementos $\iota_\uparrow p \in P_\perp$ son secuencias finitas cuyo último elemento es igual a `Val` p. Como se muestra en [19], es posible obtener un método constructivo para computar los supremos de todas las cadenas de `Stream` P. Dockins [50] desarrolló otra librería que logra representar constructivamente el lifting de un predominio sin usar coinducción.

Dada una función $f : P \to Q_\perp$, la *extensión estricta* $f_\perp : P_\perp \to Q_\perp$ es la función tal que $f_\perp(\iota_\uparrow p) = f\,p$ y además $f_\perp(\perp) = \perp$. En Coq, la función $f_\perp$ se puede definir como sigue:

Extensión estricta (fragmento de la librería)

```
Definition kleislit D E (f: D -> Stream E) :=
  cofix kl (l:Stream D) :=
   match l with Eps l => Eps (kl l) | Val d => f d end.
(* ... *)
Lemma kleisli_Valeq: forall (D E:ordType) v (d:D) (f : D =-> lift_ordType E),

   v == Val d ->
   kleislit f v == (f d:lift_ordType E).
(* ... *)
Lemma kleisli_bot: forall D E (f:D =-> E _BOT), kleisli f PBot == PBot.
```

Los lemas `kleisli_Valeq` y `kleisli_bot` prueban que `kleislit f` es efectivamente la extensión estricta de `f`. La estructura `lift_ordType` se utiliza para construir el *lifting* de un dominio, en este caso `lift_ordType E` es el lifting del conjunto preordenado `E`. Observemos que el parámetro `f` tiene tipo `D -> Stream E` en `kleislit` mientras que en `kleisli_Valeq` tiene tipo `D =->lift_ordType E`. Este último tipo es en realidad una estructura que contiene a la función propiamente dicha junto con una prueba de monotonía. La coerción implícita de esta estructura al tipo `D -> Stream E` mantiene este hecho transparente en el código. De manera similar, la función `kleisli` que aparece en el lema `kleisli_bot` es una versión de `kleislit` acompañada con una prueba de continuidad.

Entre las extensiones que hicimos a la librería podemos mencionar la función `GKL` que es una generalización de la función `kleisli` donde la imagen de la

función f puede ser cualquier dominio D, no necesariamente obtenido mediante lifting.

```
Definition GKL (f : P —=> D) : P _BOT —=> D.
```

Esta generalización es necesaria para definir la semántica denotacional de la proyección condicional (Definición 97). Las demostraciones de monotonía y continuidad se pueden ver en el archivo PredomMore.v. También extendimos la librería con "morfismos *n*-arios" que permitieron definir la semántica de los operadores aritméticos de cualquier aridad finita (ver PredomNary.v).

La representación de los términos del lenguaje fuente es diferente a las que hemos usado en el resto de las formalizaciones. En este caso los términos deben estar bien tipados bajo un contexto ctx. Mostramos aquí sólo los constructores correspondientes al cálculo lambda y el operador de punto fijo.

Términos bien tipados (Definición 86)

```
Inductive term (ctx : context) : type —> Type :=
| term_abs: forall θ1 θ2, term (θ1 :: ctx ) θ2 —> term ctx (θ1→θ2)
| term_app: forall θ1 θ2, term ctx (θ1→θ2 ) —> term ctx θ1 —> term ctx θ2
| term_var: forall n θ, lookup ctx n = Some θ —> term ctx θ
| term_fix: forall θ, term ctx (θ→θ) —> term ctx θ
(* ... *)
```

La función ty_denot define la semántica denotacional de los tipos:

Semántica de tipos (Definición 93)

```
Fixpoint ty_denot (θ : type) : cppoType :=
  match θ with
      | ty_int => natFlat
      | θ1→θ2 => ty_denot θ1 —=> ty_denot θ2
      | θ1 × θ2 => ty_denot θ1 * ty_denot θ2
  end.
```

Aquí natFlat es el dominio llano de los números naturales. Por otro lado, el dominio ty_denot θ1 —=> ty_denot θ2 es el espacio de funciones continuas de ty_denot θ1 en ty_denot θ2, y el dominio ty_denot θ1 * ty_denot θ2 es el producto cartesiano ordenado punto a punto. La siguiente función define la semántica denotacional de los contextos.

Semántica de contextos (Definición 94)

```
Fixpoint ctx_denot (ctx : context) : cppoType :=
 match ctx with
  | nil => One
  | θ :: ctx =>  ctx_denot ctx * ty_denot θ
 end.
```

Aquí `One` es el dominio cuyo único elemento se denomina `tt`, que hemos usado para representar al ambiente vacío.

La semántica denotacional se define por recursión en la estructura del término. Mostramos a continuación el caso de la abstracción y el operador de punto fijo.

Semántica denotacional (Definición 95)

```
Fixpoint term_denot ctx θ (t : term ctx θ) :
ctx_denot ctx =—> ty_denot θ :=
 match t in term _ θ1 return (ctx_denot ctx =—> ty_denot θ1) with
  | term_abs _ _ t  => exp_fun (term_denot t)
  | term_fix _ t => FIXP << term_denot t
  (* ... *)
 end.
```

En el caso del constructor `term_abs t`, el término `t` tiene tipo `term (θ1 :: ctx) θ2` y `term_denot t` tiene tipo `ctx_denot ctx * ty_denot θ1 =—> ty_denot θ2`. Mediante la aplicación de `exp_fun` se obtiene la versión *currificada* de `term_denot t` que tiene tipo `ctx_denot ctx =—> (ty_denot θ1 =—> ty_denot θ2)`. En realidad, en la librería se define a `exp_fun` en términos *categóricos*, pues es una función que asigna a cada morfismo `X * Y =—> Z` su traspuesta `X =—> (Y =—> Z)` correspondiente al objeto exponencial `Y =—> Z`.

Por otro lado, en el caso de `term_fix t`, el término `t` tiene tipo `term ctx (θ→θ)` y `term_denot t` tiene tipo `ctx_denot ctx =—> ty_denot (θ→θ)`. Mediante la composición con `FIXP` se obtiene una función de tipo `ctx_denot ctx =—> ty_denot θ` La función `FIXP : (D =—> D) =—> D` asigna a cada función continua en `D =—> D` su menor punto fijo en `D`. La formalización de `FIXP` junto con la demostración de su continuidad pueden consultarse en el archivo `PredomFix.v` de la librería [19].

El teorema de corrección para términos cerrados (convergentes y divergentes) se enuncian como sigue:

Teoremas 10 y 12

```
Definition converges_to (w : conf) m : Prop :=
 exists η, star trans w ([ iconst m, η], nil).
Theorem correctness_int_convergent:
 forall (t : term nil ty_int) m,
    term_denot t tt == Val m —>
    converges_to ([compile t, nil], nil) m.

Theorem correctness_int_divergent:
  forall t : term nil ty_int,
    term_denot t tt == PBot —>
    forall s, diverges ([compile t, nil], s).
```

Dejamos para consultar en la formalización completa la definición de las relaciones de aproximación y la demostración de todas sus propiedades. También se omite la definición de los operadores ortogonales, aunque en la siguiente sección se muestra una definición alternativa que se usó en la formalización del Capítulo 7.

8.6 Evaluación Lazy

Por último comentaremos sobre la formalización del teorema de corrección del compilador que se presentó en el Capítulo 7. El lenguaje fuente es una variante del cálculo lambda tipado, pero con una versión restringida de la aplicación (el operando sólo puede ser una variable) y con el operador *let t in t′* que permite declarar variables locales.

Términos bien tipados (Definición 112)

```
Inductive var : ctx -> type -> Type :=
 | zvar : forall c θ, var (θ :: c) θ
 | svar : forall c θ1 θ2, var c θ1 -> var (θ2 :: c) θ1.

Inductive term (c : ctx) : type -> Type :=
 | term_unit : term c tunit
 | term_var : forall θ, var c θ -> term c θ
 | term_abs : forall θ1 θ2, term (θ1 :: c) θ2 -> term c (θ1→θ2)
 | term_app : forall θ1 θ2, term c (θ1→θ2) -> var c θ1 -> term c θ2
 | term_let : forall θ1 θ2, term c θ1 -> term (θ1 :: c) θ2 -> term c θ2.
```

Aquí usamos una representación *fuertemente tipada* [17] de las variables de un término. Cada variable está inherentemente asociada con un tipo, que a su vez está determinado posicionalmente de acuerdo al contexto donde ocurre. Con `zvar` se construye una una variable cuyo tipo está en la primera posición del contexto, mientras que con `svar` el tipo se encuentra en las posiciones subsiguientes del mismo.

El compilador traduce los términos bien tipados a instrucciones de de la máquina abstracta de Sestoft [111]. A diferencia de la máquina de Krivine, esta máquina incorpora un heap en la configuración con el propósito de evitar la repetición innecesaria de evaluaciones. Mostramos a continuación algunas de las transiciones representadas en Coq:

Transiciones (Definición 118)

```
Inductive trans : conf -> conf -> Type :=
 | tgrab_apply : forall Γ i η p s,
     (Γ, (igrab i, η), ¤ p :: s) ↦ (Γ, (i, p :: η) , s)
```

```
| tgrab_update: forall Γ i η p s,
    let v : MClos := (igrab i, η) in
    (Γ, (igrab i, η), # p :: s) ↦ (Map.add p v Γ, (igrab i, η), s)
| tlet : forall Γ i i' η p s,
    fresh_for p Γ η s  —>
    (Γ, (ilet i i', η), s) ↦ (Map.add p (i, η) Γ, (i', p :: η), s)
(* ... *)
where "w1 ↦ w2" := (trans w1 w2).
```

El heap Γ es un mapeo de punteros a clausuras. Fue implementado en Coq usando los módulos `Coq.FMapWeakList` de la librería estándar. Esos módulos fueron extendidos para soportar la noción general de *compatibilidad* (Definición 121) y de *unión* (Definición 122) de mapeos. Los constructores `tgrab_apply` y `tgrab_update` corresponden a las dos reglas de transición de la instrucción `Grab`. La primera transición consiste en eliminar el marcador `¤ p` del tope de la pila e insertar el puntero `p` en el entorno. La segunda transición corresponde a una actualización del heap: el marcador `# p` indica que el heap debe actualizarse en el puntero `p`; la operación `Map.add p` $\alpha\,\Gamma$ inserta el mapeo `p` $\mapsto \alpha$ al heap Γ, sobrescribiendo el valor anterior de `p` si fuese necesario. Por otro lado, la transición `tlet` parte de una configuración con el código `(ilet i i', η)`, inserta un nuevo mapeo `p` $\mapsto$ `(i, η)` en el heap y continua la ejecución de `i'` bajo el entorno `p :: η`. La proposición `fresh_for` establece que justamente `p` es un puntero nuevo, es decir, que no ocurre en ningún componente de la configuración.

La máquina abstracta fue formalizada usando un módulo parametrizado con el tipo `PointerType`, el tipo de los punteros. Las dos condiciones para el tipo son: la igualdad de Leibniz (`Logic.eq`) debe ser decidible y debe ser infinito. La segunda condición es necesaria para poder elegir un puntero nuevo a partir de cualquier conjunto finito de punteros.

```
Definition exists_fresh_pointer
  := forall Γ  η s, exists p, fresh_for p Γ η s .
```

Continuamos con la definición de `Rp( θ , d )`, el conjunto realizadores primitivos de un valor denotacional `d` del conjunto `ty_denot θ`.

Los conjuntos son representados con el tipo `Ensemble`, en este caso necesitamos un subconjunto de `RType`, que son pares de la forma (Γ, α) donde Γ es un heap y α una clausura.

Realizadores (Definición 126)

```
Fixpoint Primitives (θ : type) : ty_denot θ —> Ensemble RType :=
 match θ with
  | tunit => fun _ =>
```

```
    let on_iunit f :=
      fun r => match r with
               | (Γ, (iunit, η)) => f Γ η
               | _ => False
               end
    in
    on_iunit (fun Γ η => env_ptr η ⊆ key_set Γ)
  | θ1→θ2 => fun f =>
    let on_igrab f :=
      fun r => match r with
               | (Γ, (igrab i, η)) => f Γ i η
               | _ => False
               end
      in
      let condition Γ i η :=
           forall d Γ' α,
             (Γ', α) ϵ R(θ1, d) ->
             r_wf (Γ', α) -> Γ ⋈ Γ' ->
             forall p, (Γ ∪ Γ') ⋈ {p ↦ α} ->
                  (add p α (Γ ∪ Γ'), (i, p :: η)) ϵ R(θ2, f d)
      in
      on_igrab (fun Γ i η => env_ptr η ⊆ key_set Γ /\ condition Γ i η)
 end
where "Rp( θ , d )" := (Primitives θ d)
  and "R( θ , d )" := (Primitives θ d ⊥⊤ ).
```

El tipo `Primitives (θ : type)` se define mediante recursión en la estructura del tipo θ. En el caso del tipo `tunit`, sólo los pares de la forma $(\Gamma, (\texttt{iunit}, \eta))$ pueden ser realizadores primitivos, siempre y cuando los punteros que ocurran en η estén incluidos en el dominio del heap Γ. En el caso del tipo $\theta_1 \to \theta_2$ los realizadores de una función `f` son pares de la forma $(\Gamma, (\texttt{igrab i}, \eta))$ que cumplen con algunas condiciones. Al igual que antes, se requiere que los punteros de η estén incluidos en el dominio del heap. La otra condición es una cuantificación universal sobre todos los realizadores $(\Gamma', \alpha) \in R(\theta_1, d)$ bien formados (Definición 120) tales que Γ' es compatible con Γ y a su vez para todo puntero p tal que la unión $\Gamma \cup \Gamma'$ es compatible son el heap $\{p \mapsto \alpha\}$.

Los operadores ortogonales se definieron de manera genérica usando módulos paramétricos:

```
Module Type TstElt.
  Parameter Elt : Type.
  Parameter Tst : Type.
  Parameter satif: Elt -> Tst -> Prop.
End TstElt.
```

```
Module OrthogonalOperators (Import T : TstElt).
  Notation "e ⊨ t" := (satif e t) (at level 70, no associativity).
  Definition btop (T : Ensemble Tst) : Ensemble Elt :=
    fun e => forall t, In _ T t -> e ⊨ t.
  Definition bbot (E : Ensemble Elt) : Ensemble Tst :=
    fun t => forall e, In _ E e -> e ⊨ t.
  Notation "X ⊤" := (btop X) (at level 40, no associativity).
  Notation "X ⊥" := (bbot X) (at level 40, no associativity).
End OrthogonalOperators.
```

El parámetro `T` es un módulo de tipo `TstElt`: debe contener un tipo `Elt` para los elementos, un tipo `Tst` para los tests y una relación de satisfabilidad. También se definieron de manera modular la conexión de Galois (Definición 62) y el operador de clausura (Definición 63). Las propiedades de estos operadores pueden encontrarse en la formalización completa.

Se omite la definición del conjunto de observaciones con el cual posteriormente se da una definición concreta de la relación de satisfacibilidad. Además de las reglas O_1 y O_2 (Definición 123) se asume que el conjunto de observaciones es cerrado por equivalencia de heaps. Dejamos para trabajo futuro la demostración en Coq de que ese conjunto de observaciones tiene instancias no triviales (e.g configuraciones convergentes). El siguiente teorema establece que la compilación de un término se puede ver como un realizador de la semántica denotacional del mismo.

Teorema 13

```
Theorem compiler_correctness :
 exists_fresh_pointer ->
 forall ctx a,
 forall t : term ctx a,
 forall ρ : ctx_denot ctx,
 forall Γ,
  heap_wf Γ ->  forall η,
    (Γ, η) ◆ ρ ->
    (Γ, (⦇ t ⦈, η)) ▪ ⟦ t ⟧ρ .
```

Los símbolos ◆ y ▪ hacen referencia respectivamente a las aproximaciones de ambientes (Definición 128) y de valores denotacionales (Definición 127). La primera hipótesis del teorema puede eliminarse si se elige una representación concreta para los punteros que cumpla las condiciones mencionadas anteriormente, por ejemplo, los números naturales.

Conclusión y Trabajo Futuro

En la tesis analizamos cómo demostrar la corrección de la compilación con respecto a la semántica del lenguaje fuente usando como lenguaje intermedio las instrucciones de distintas máquinas abstractas. Exploramos diferentes lenguajes con evaluación normal, cuya semántica se definió mediante tres métodos: semántica small-step, semántica big-step y semántica denotacional.

Como modelo de ejecución, usamos distintas variantes de la máquina abstracta de Krivine [71], y para el caso de la evaluación lazy, utilizamos la máquina abstracta de Sestoft [111].

Comenzamos con un análisis de la reducción small-step del *cálculo de clausuras* [44], demostramos la corrección de la función de compilación que traduce los términos del cálculo a las instrucciones de la máquina abstracta. Analizamos tanto términos convergentes como divergentes, aplicando respectivamente inducción y coinducción para completar las demostraciones.

Posteriormente consideramos la semántica de evaluación big-step, primero para un lenguaje puramente funcional, y luego para un lenguaje imperativo. En ambos casos desarrollamos la prueba de corrección para términos convergentes y divergentes. En el caso imperativo, la reglas de evaluación modelan de manera explícita la disciplina de pila característica de los lenguajes Algol-like [99].

Desarrollamos un enfoque basado en la noción de *realizabilidad* de Krivine [72] con el cuál logramos demostrar la corrección del compilador con respecto a la semántica denotacional. Se definieron realizadores primitivos sobre la máquina de Krivine y luego se utilizó *biortogonalidad* [94] para extender el conjunto de realizadores mediante un operador de clausura. Además, aplicamos *step-indexing* [7, 11] para definir un esquema de realizabilidad ante la presencia del operador de punto fijo. Si bien Benton et al. [15, 16] aplica técnicas similares

para demostrar la corrección de un compilador utilizando la máquina SECD [73], al enfocarse en lenguajes estrictos, el método que propone no es aplicable sobre la máquina de Krivine ni sobre lenguajes con evaluación normal.

Se presentó además un nuevo esquema de realizabilidad aplicable sobre una máquina abstracta con evaluación lazy, basada en la máquina de Sestoft [111]. Para implementar esa estrategia de evaluación, se utilizó un heap cuyo propósito es evitar la repetición innecesaria de la ejecución de clausuras. Si bien Jaber [61] también explora el uso de heaps, el propósito de los mismos es implementar un lenguaje con referencias (estados mutables) y no optimizar el orden de evaluación; además el heap no forma parte de los realizadores ni de lo tests como sí lo hace en nuestro enfoque. Como continuación de este trabajo, se planea investigar cómo incluir el operador de punto fijo y otros constructores al lenguaje que permitan extender su expresividad.

Todos las demostraciones de corrección fueron formalizadas en Coq. Una de nuestras contribuciones con respecto a las formalización es la extensión de la librería de teoría de dominios desarrollada por Benton et al. [19].

Como trabajo futuro, consideramos que el área de investigación más promisoria es la certificación de compiladores para lenguajes funcionales con evaluación lazy. Un lenguaje que adopta la estrategia de evaluación es Haskell [1, 90], considerado el lenguaje funcional más utilizado en la actualidad. Sin embargo, no existe un compilador certificado para ese lenguaje. Se pueden encontrar trabajos en esa dirección, como por ejemplo la verificación en Coq de la máquina de reducción de grafos STG [93], utilizada en el compilador GHC, pero que todavía están muy lejos de conseguir el objetivo final.

En el doctorado se estudiaron también sistemas de generación de código donde la semántica de un término del lenguaje fuente se describe directamente en términos de su compilación. En particular, estudiamos la generación de código propuesta por Reynolds [98] para lenguajes Algol-like utilizando semántica funtorial [87, 88]. A grandes rasgos, la semántica del lenguaje se define con una categoría funtorial cuyos objetos son funtores que aplicados a un estado producen código en un lenguaje intermedio. Es esperable que como el código generado por el compilador se obtiene directamente de la semántica del lenguaje se pueda demostrar con facilidad su corrección, sin embargo no se ha explorado aún con profundidad este enfoque en el área de corrección de compiladores.

Dejamos también como trabajo futuro explorar el enfoque propuesto por Danvy et al. [47, 48] en el cual, a partir de una serie de transformaciones, se logra derivar una máquina abstracta a partir de la semántica operacional del lenguaje. La estrecha relación entre la semántica y la máquina abstracta sugiere la posibilidad de una prueba directa por construcción de la corrección del compilador. Un punto de partida es la formalización en Coq que desarrolló Sieczkowski et al. [112] de este mismo enfoque.

En la tesis hemos investigado la certificación de compiladores que generan código ejecutable sobre máquinas abstractas, en particular utilizamos la máquina abstracta de Krivine. Las máquinas abstractas son útiles como una representación intermedia de muchos compiladores, porque permiten generar código sin lidiar con los detalles de implementación impuestos al utilizar un lenguaje ensamblador de un microprocesador real. Sin embargo, para lograr un compilador que sea útil en la práctica, es necesario generar código objeto en una representación más cercana al lenguaje máquina.

Un lenguaje intermedio ampliamente utilizado en la actualidad es LLVM IR correspondiente al framework de compilación LLVM [74]. Este framework permite generar código objeto para diferentes arquitecturas de forma sencilla y transparente. Consideramos factible desarrollar un compilador certificado utilizando como lenguaje objeto la representación LLVM IR. Para construir la prueba de corrección se podría utilizar una librería desarrollada por el proyecto VeLLVM [119, 120], que consiste de una formalización en Coq de la semántica del lenguaje LLVM IR junto con un conjunto de resultados útiles para razonar sobre el mismo.

Bibliografía

[1] Haskell, A Purely Functional Programming Language. https://www.haskell.org/.

[2] The Agda Proof Assistant. http://wiki.portal.chalmers.se/agda.

[3] The Coq Proof Assistant. http://coq.inria.fr/.

[4] The Coq Proof Assistant, Reference Manual. https://coq.inria.fr/distrib/current/refman/.

[5] Samson Abramsky and Achim Jung. Domain Theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994.

[6] Peter. Aczel. *An Introduction to Inductive Definitions*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. Elsevier, 1977.

[7] Amal Ahmed. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 69–83, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] Thorsten Altenkirch. A Formalization of the Strong Normalization Proof for System F in LEGO. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, pages 13–28, London, UK, UK, 1993. Springer-Verlag.

[9] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free Normalisation for a Polymorphic System. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, Washington, DC, USA, 1996. IEEE Computer Society.

[10] Andrew W. Appel and Sandrine Blazy. Separation Logic for Small-step Cminor. *CoRR*, abs/0707.4389, 2007.

[11] Andrew W. Appel and David McAllester. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. *ACM Transactions on Programing Languages and Systems*, 23(5):657–683, September 2001.

[12] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised Report on the Algorithm Language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[13] Hendrik Pieter Barendrekt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Pub, North-Holland, 1981.

[14] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally Verified Optimizing Compilation in ACG-based Flight Control Software. In *ERTS2 2012: Embedded Real Time Software and Systems*. AAAF, SEE, 2012.

[15] Nick Benton and Chung-Kil Hur. Biorthogonality, Step-indexing and Compiler Correctness. *SIGPLAN Not.*, 44(9):97–108, 2009.

[16] Nick Benton and Chung-Kil Hur. Realizability and Compositional Compiler Correctness for a Polymorphic Language. Technical report, MSR-TR 2010-62, Microsoft Research, 2010.

[17] Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012.

[18] Nick Benton, Andrew Kennedy, and Carsten Varming. Some Domain Theory and Denotational Semantics in Coq. In *TPHOLs*, pages 115–130, 2009.

[19] Nick Benton, Andrew Kennedy, and Carsten Varming. Formalizing Domains, Ultrametric Spaces and Semantics of Programming Languages. Unpublished, 2010.

[20] Yves Bertot. CoInduction in Coq. *CoRR*, 2006.

[21] Yves Bertot. Theorem Proving Support in Programming Language Semantics. *CoRR*, 2007.

[22] Yves Bertot, Pierre Castéran, Gérard Huet, and Christine Paulin-Mohring. *Interactive Theorem Proving and Program Development : Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, Berlin, New York, 2004.

[23] Yves Bertot and Vladimir Komendantsky. Fixed Point Semantics and Partial Recursion in Coq. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '08, pages 89–96, New York, NY, USA, 2008. ACM.

[24] Malgorzata Biernacka and Olivier Danvy. A Concrete Framework for Environment Machines. *ACM Transactions on Computational Logic*, 9(1), 2007.

[25] Dariusz Biernacki and Piotr Polesiuk. Logical Relations for Coherence of Effect Subtyping. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–122, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[26] Garret Birkhoff. *Lattice Theory*, volume 25. 1940.

[27] Aleš Bizjak and Lars Birkedal. *Step-Indexed Logical Relations for Probability*, pages 279–294. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[28] Thomas Braibant. *Coquet: A Coq Library for Verifying Hardware*, pages 330–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[29] Thomas Braibant and Adam Chlipala. *Formal Verification of Hardware Synthesis*, pages 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[30] N.G De Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[31] Venanzio Capretta and Amy P. Felty. *Combining de Bruijn Indices and Higher-Order Abstract Syntax in Coq*, pages 63–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[32] Adam Chlipala. A Certified Type-preserving Compiler from Lambda Calculus to Assembly Language. *SIGPLAN Not.*, 42(6):54–65, 2007.

[33] Adam Chlipala. Parametric Higher-order Abstract Syntax for Mechanized Semantics. volume 43, pages 143–156, New York, USA, 2008. ACM.

[34] Adam Chlipala. A Verified Compiler for an Impure Functional Language. In *POPL*, pages 93–106, 2010.

[35] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2015. http://adam.chlipala.net/cpdt/.

[36] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 33:346–366, 1932.

[37] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *The American Journal Of Mathematics*, 58:345–363, 1936.

[38] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[39] Thierry Coquand. An Analysis of Girard's Paradox. In *In Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, 1986.

[40] Thierry Coquand. Infinite Objects in Type Theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 62–78. Springer, Berlin, Heidelberg, 1993.

[41] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[42] Thierry Coquand and Christine Paulin. Inductively Defined Types. In *Proceedings of the International Conference on Computer Logic*, COLOG '88, pages 50–66, London, UK, UK, 1990. Springer-Verlag.

[43] Solange Coupet-Grimal and Line Jakubiec. Coq and Hardware Verification: A Case Study. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '96, pages 125–139, London, UK, UK, 1996. Springer-Verlag.

[44] Pierre-Louis Curien. An Abstract Framework for Environment Machines. *Theoretical Computer Science*, 82(2):389–402, 1991.

[45] Haskell Curry. Functionality in Combinatory Logic. In *National Academy of Sciences*, volume 20, pages 584–590, 1934.

[46] D. A. Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

[47] Olivier Danvy. Defunctionalized Interpreters for Programming Languages. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 131–142. ACM, 2008.

[48] Olivier Danvy and Lasse Nielsen. Refocusing in Reduction Semantics. Research report BRICS RS-04-26, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, 2004.

[49] Stephan Diehl and Peter Sestoft. Abstract Machines for Programming Language Implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[50] Robert Dockins. Formalized, Effective Domain Theory in Coq. `http://rwd.rdockins.name/domains/`.

[51] Derek Dreyer, Georg Neis, and Lars Birkedal. The Impact of Higher-order State and Control Effects on Local Relational Reasoning. *SIGPLAN Not.*, 45(9):143–156, 2010.

[52] Eric Eide and John Regehr. Volatiles are Miscompiled, and What to Do About It. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 255–264, New York, USA, 2008. ACM.

[53] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer.

[54] Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael Mislove, and Dana S. Scott. *Continuous Lattices and Domains*, volume 93 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2003.

[55] Timothy G. Griffin. A Formulae-as-type Notion of Control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM.

[56] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional Runtime Systems within the Lambda-sigma Calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.

[57] Haskell and Robert Feys. *Combinatory Logic*, volume 1. North-Holland Pub, 1958.

[58] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the AMS*, 146:29–60, 1969.

[59] Ralf Hinze. Church Numerals, Twice! *Journal Functional Programming*, 15(1):1–13, 2005.

[60] William A. Howard. The Formulas-as-types Notion of Construction. pages 479–490. Academic Press, 1980. Reprint of 1969 article.

[61] Guilhem Jaber and Nicolas Tabareau. Krivine Realizability for Compiler Correctness. In *Workshop LOLA 2010, Syntax and Semantics of Low Level Languages*, Edinburgh, United Kingdom, 2010.

[62] Guilhem Jaber and Nicolas Tabareau. The Journey of Biorthogonal Logical Relations to the Realm of Assembly Code. In *Workshop LOLA 2011, Syntax and Semantics of Low Level Languages*, pages 1–15, Toronto, Canada, 2011.

[63] Bart Jacobs and Jan Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 62:62–222, 1997.

[64] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating lr(1) parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 397–416, Berlin, Heidelberg, 2012. Springer-Verlag.

[65] G. Kahn. Natural Semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, UK, 1987. Springer-Verlag.

[66] S. C. Kleene. On the Interpretation of Intuitionistic Number Theory. *The Journal of Symbolic Logic*, 10(4):109–124, 1945.

[67] Adam Koprowski and Henri Binsztok. TRX: A Formally Verified Parser Interpreter. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 345–365, Berlin, Heidelberg, 2010. Springer-Verlag.

[68] C.P.J. Koymans. Models of the Lambda Calculus. *Information and Control*, 52(3):306 – 332, 1982.

[69] Georg Kreisel. Interpretation of Analysis by Means of Constructive Functionals of Finite Types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. Amsterdam, North-Holland Pub. Co., 1959.

[70] Jean-Louis Krivine. Classical Logic, Storage Operators and Second-order Lambda-calculus. *Annals of Pure and Applied Logic*, 68(1):53–78, 1994.

[71] Jean-Louis Krivine. A Call-by-name Lambda-calculus Machine. *Higher Order Symbolic Computation*, 20(3):199–207, September 2007.

[72] Jean-Louis Krivine. Realizability in Classical Logic. *Panoramas et Synthèses*, 27:197–229, 2009.

[73] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.

[74] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, pages 75–88, San Jose, CA, USA, 2004.

[75] John Launchbury. A Natural Semantics for Lazy Evaluation. In *POPL*, pages 144–154, 1993.

[76] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[77] Xavier Leroy. Mechanized Semantics - with Applications to Program Proof and Compiler Verification. In *Logics and Languages for Reliability and Security*, pages 195–224. IOS Press, 2010.

[78] Xavier Leroy and Hervé Grall. Coinductive Big-step Operational Semantics. *Information and Computation*, 207(2):284–304, February 2009.

[79] Theodore A. Linden. A Summary of Progress Toward Proving Program Correctness. In *AFIPS Fall Joint Computing Conference (1)*, pages 201–211, 1972.

[80] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, 2013. Springer.

[81] John McCarthy and James Painter. Correctness of a Compiler for Arithmetic Expressions. In *Mathematical Aspects of Computer Science*, volume 19 of *1*, pages 33–41. American Mathematical Society, 1967.

[82] Albert R. Meyer. What is a Model of the Lambda Calculus? *Information and Control*, 52(1):87 – 122, 1982.

[83] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[84] F. Lockwood Morris. Advice on Structuring Compilers and Proving them Correct. In *POPL*, pages 144–152, 1973.

[85] Keiko Nakata. Denotational Semantics for Lazy Initialization of Letrec: Black Holes as Exceptions Rather than Divergence. In *7th Workshop on Fixed Points in Computer Science*, 2010.

[86] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler (with retrospective). In *Best of PLDI*, pages 612–625, 1998.

[87] Frank J. Oles. *A Category-theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, Syracuse, NY, USA, 1982.

[88] Frank J. Oles. Type Algebras, Functor Categories and Block Structure. In Maurice Nivat and John C Reynolds, editors, *Algebraic methods in semantics*, pages 543–573. Cambridge University Press, New York, NY, USA, 1986.

[89] Oystein Ore. Galois Connexions. *Transactions of the American Mathematical Society*, 55:493–513, 1944.

[90] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, 2003.

[91] F. Pfenning and C. Elliott. Higher-order Abstract Syntax. *SIGPLAN Not.*, 23(7):199–208, 1988.

[92] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. http://www.cis.upenn.edu/~bcpierce/sf.

[93] Maciej Pirog and Dariusz Biernacki. A Systematic Derivation of the STG Machine Verified in Coq. *SIGPLAN Not.*, 45(11):25–36, 2010.

[94] A. M. Pitts and I. D. B. Stark. Operational Reasoning for Functions with Local State. In Andrew D. Gordon and Andrew M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–274. Cambridge University Press, New York, NY, USA, 1998.

[95] Gordon D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logical and Algebraic Methods in Programming*, 60-61:17–139, 2004.

[96] Julian Rathke, Vladimiro Sassone, and Paweł Sobociński. *Semantic Barbs and Biorthogonality*, pages 302–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[97] John C. Reynolds. The Coherence of Languages with Intersection Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, TACS '91, pages 675–700, London, UK, UK, 1991. Springer-Verlag.

[98] John C. Reynolds. Using Functor Categories to Generate Intermediate Code. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 25–36. ACM, New York, NY, USA, 1995.

[99] John C. Reynolds. The Essence of Algol. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like Languages, Volume 1*, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[100] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, New York, NY, USA, 1999.

[101] John C. Reynolds. The Meaning of Types : From Intrinsic to Extrinsic Semantics. Technical Report RS-00-32, BRICS, 2000.

[102] John C. Reynolds. What Do Types Mean?: From Intrinsic to Extrinsic Semantics. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, pages 309–327. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[103] M. Rittri and Institutionen för informationsbehandling (Göteborg). *Proving the Correctness of a Virtual Machine by a Bisimulation*. Department of Computer Sciences, 1988.

[104] Leonardo Rodríguez. An Intrinsic Denotational Semantics for a Lazy Functional Language. In *Brazilian Programming Languages Symposium - SBLP 2015*, volume 9325 of *Lecture Notes in Computer Science*, pages 77–80. Springer, 2015.

[105] Leonardo Rodríguez, Daniel Fridlender, and Miguel Pagano. A Certified Extension of the Krivine Machine for a Call-by-Name Higher-Order Imperative Language. In Ralph Matthes and Aleksy Schubert, editors, *19th*

International Conference on Types for Proofs and Programs, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 230–250, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[106] Leonardo Rodríguez, Daniel Fridlender, and Miguel Pagano. Proving Compiler Correctness Using Step-indexed Logical Relations. In *Logic and Semantic Frameworks with Applications - LSFA 2015*, Electronic Notes in Theoretical Computer Science. Elsevier, 2015.

[107] Leonardo Rodríguez, Daniel Fridlender, and Miguel Pagano. Mechanized Semantics of an Algol-like Language. Short-paper en TYPES 2013, 2013.

[108] Dana Scott. Outline of a Mathematical Theory of Computation. Technical Report PRG02, OUCL, November 1970.

[109] Dana Scott. *Domains for Denotational Semantics*, pages 577–610. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[110] Peter Selinger. From Continuation Passing Style to Krivine's Abstract Machine. Manuscript, 2003. Available in Peter Selinger's web site.

[111] Peter Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programing*, 7(3):231–264, 1997.

[112] Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki. *Automating Derivations of Abstract Machines from Reduction Semantics*, pages 72–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[113] Matthieu Sozeau and Nicolas Oury. *First-Class Type Classes*, pages 278–293. Springer Berlin Heidelberg, 2008.

[114] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, (2):285–309.

[115] Alan Turing. The p-functions in λ-k-conversion. *Journal of Symbolic Logic*, page 164, 1937.

[116] Jaap Van Oosten. Realizability: A Historical Essay. *Mathematical. Structures in Computer Science*, 12(3):239–263, 2002.

[117] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA, 1993.

[118] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, 2011.

[119] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.

[120] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal Verification of SSA-based Optimizations for LLVM. *SIGPLAN Not.*, 48(6):175–186, 2013.