Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Semántica natural como marco de verificación de compiladores en Coq

Tesis

que para optar por el grado de:

Doctor en Ciencia e Ingeniería de la Computación

Presenta:

Angel Francisco Zúñiga Chávez

Tutor:
Dr. Gerardo Eugenio Sierra Martínez
Instituto de Ingeniería, UNAM

Cotutora:
Dra. Gemma Bel Enguix
Instituto de Ingeniería, UNAM

Miembro del Comité Tutor:
Dra. Helena Montserrat Gómez Adorno
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, UNAM

Ciudad Universitaria, Ciudad de México, junio de 2021

Resumen

En la literatura dedicada a la verificación de compiladores en asistentes de demostración, usualmente se emplean formalizaciones ad-hoc, es decir, aquellas en las que se utiliza más de un formalismo. Esta situación clama por una solución en la que se abstraiga y unifique, en un sólo marco, todo lo necesario para realizar esta tarea. En este trabajo, se presenta la semántica natural como un marco de verificación de la corrección total de compiladores en Coq, en el cual se utiliza un único formalismo, semántica natural (coinductiva), para verificar la corrección de un compilador tanto en cómputos finitos como en cómputos infinitos (corrección total).

Publicaciones

El contenido principal de esta tesis está publicado en los siguientes artículos:

[1] Angel Zúñiga y Gemma Bel-Enguix. "A Correct Compiler from Mini-ML to a Big-Step Machine Verified Using Natural Semantics in Coq". En: *Actas de las XVIII Jornadas de Programación y Lenguajes (PROLE 2018)*. Ed. por Yolanda Ortega-Mallén. Universidad de Sevilla. Sevilla, España: Biblioteca digital SISTEDES, sep. de 2018. URL: http://hdl.handle.net/11705/PROLE/2018/002.

[2] Angel Zúñiga y Gemma Bel-Enguix. "Coinductive Natural Semantics for Compiler Verification in Coq". En: *Mathematics* 8.9 (sep. de 2020), pág. 1573. ISSN: 2227-7390. DOI: 10.3390/math8091573.

Adicionalmente, han sido publicados los siguientes artículos que no aparecen en esta tesis:

[3] Angel Zúñiga, Gerardo Sierra, Gemma Bel-Enguix y Sofía N. Galicia-Haro. "Towards a Natural Language Compiler". En: *Advances in Computational Intelligence*. Ed. por Ildar Batyrshin, María de Lourdes Martínez-Villaseñor y Hiram Eredín Ponce Espinosa. Vol. 11289. LNAI. Guadalajara, Mexico: Springer International Publishing, oct. de 2018, págs. 67-80. ISBN: 978-3-030-04497-8. DOI: 10.1007/978-3-030-04497-8_6.

[4] Angel Zúñiga, Gerardo Sierra, Gemma Bel-Enguix y Javier Gomez. "SICIoT: A simple instruction compiler for the Internet of Things". En: *Internet of Things* 12 (dic. de 2020), pág. 100304. ISSN: 2542-6605. DOI: 10.1016/j.iot.2020.100304.

Agradecimientos

A Dios

Índice general

Resumen I

Publicaciones II

Agradecimientos III

1. Introducción **1**
1.1. Contexto 1
1.2. Problema 5
1.3. Contribuciones 6

2. Verificación de compiladores en Coq **9**
2.1. Técnicas de verificación 9
2.2. Derivación de máquinas abstractas 11
2.3. Formalizaciones ad-hoc 12
2.4. CertiCoq 12
2.5. Semántica 13

3. Semántica natural **16**
3.1. Traducción a índices de de Bruijn 16
3.1.1. MiniML 16
3.1.2. $\mathsf{MiniML}^{\mathsf{dB}}$ 23
3.1.3. Traducción 24
3.1.4. Corrección 27
3.2. Generación de código 28
3.2.1. Máquina MSECD 28
3.2.1.1. Compilación 31
3.2.1.2. Corrección 32
3.2.2. Máquina MSECD de paso largo 32
3.2.2.1. Compilación 36
3.2.2.2. Corrección 38

4. Semántica natural coinductiva **40**
4.1. Traducción a índices de de Bruijn 40
4.1.1. MiniML 40
4.1.2. $\mathsf{MiniML}^{\mathsf{dB}}$ 43

4.1.3. Corrección . . . 43
4.2. Generación de código . . . 44
4.2.1. Máquina MSECD . . . 44
4.2.1.1. Semántica de paso corto de cómputos infinitos . . . 44
4.2.1.2. Corrección . . . 46
4.2.2. Máquina MSECD de paso largo . . . 46
4.2.2.1. Semántica de paso largo de cómputos infinitos . . . 47
4.2.2.2. Corrección . . . 51

5. Algoritmo de traducción a Coq **52**
5.1. Generalidades . . . 52
5.2. Lenguajes . . . 52
5.2.1. Sintaxis abstracta . . . 53
5.2.2. Semántica natural . . . 53
5.2.3. Semántica natural coinductiva . . . 53
5.3. Traducciones . . . 53
5.3.1. Semántica natural . . . 54
5.3.2. Corrección . . . 54
5.3.2.1. Casos especiales . . . 54

6. Conclusiones y trabajo futuro **58**
6.1. Semántica natural para todos los cómputos . . . 59
6.2. Concurrencia en semántica natural . . . 60
6.3. Nombres de las variables . . . 61
6.4. Semántica natural en otros asistentes de demostración . . . 62
6.5. Semántica natural en Maude . . . 65

A. Demostraciones **67**

Bibliografía **99**

Capítulo 1

Introducción

Esta sección introduce el problema que se trata en esta tesis (Sección 1.2) y presenta las contribuciones de la misma (Sección 1.3). Previamente (Sección 1.1), se hace una exposición de la forma usual en que se presenta el problema en la literatura y se discute con base en ésta los propósitos y aportaciones de la tesis.

1.1. Contexto

En este trabajo se ataca el problema de la verificación de compiladores en asistentes de demostración. Para introducir este problema, tomemos como punto de partida uno más simple, a saber, el problema de la verificación de compiladores en abstracto (es decir, en "papel y lápiz", sin tomar en cuenta su formalización en un asistente de demostración). Tomemos como referencia una presentación usual de este problema en la literatura, por ejemplo, como se expone en el libro de Nielson y Nielson [78]. Nielson y Nielson establecen la corrección de un compilador de un lenguaje de alto nivel a una máquina abstracta de la siguiente manera: suponiendo que e es una expresión del lenguaje fuente, si e se evalúa a v mediante semántica natural $e \Rightarrow v$, entonces la compilación de e, $[\![e]\!]$, también debe evaluarse a v mediante el cierre reflexivo y transitivo de la semántica de paso corto de la máquina $[\![e]\!] \stackrel{*}{\to} v$, (simulación hacia adelante, *forward simulation*); además, debe cumplirse el sentido contrario, es decir, si el código de máquina correspondiente a la compilación de una expresión e, $[\![e]\!]$, se evalúa a un valor v mediante el cierre reflexivo y transitivo de la semántica de paso corto de la máquina $[\![e]\!] \stackrel{*}{\to} v$, entonces e también debe evaluarse a v mediante semántica natural $e \Rightarrow v$ (simulación hace atrás, *backward simulation*). Si ambas propiedades se cumplen entonces se dice que el compilador es correcto.

Partiendo de esta forma de establecer la corrección a la que llamaremos corrección de referencia, a continuación, iremos haciendo varias observaciones con el fin de describir las aportaciones de esta tesis.

Primero, hemos de decir que aunque uno de los usos más conocidos de la semántica natural es como formalismo para la especificación de la semántica de un lenguaje de alto nivel, Kahn [55] introduce la semántica natural como formalismo para expresar la semántica de lenguajes, de máquinas y de traducciones, es decir, como un

marco semántico unificador. Posteriormente, Despeyroux [38] muestra cómo usando semántica natural en esta forma se puede establecer y demostrar la corrección de una traducción. En nuestro problema, usando semántica natural como marco se tendría: si e se evalúa a v mediante semántica natural $e \Rightarrow v$, si e se traduce a c mediante semántica natural $e \Downarrow c$, entonces c se evalúa a v mediante semántica natural $c \Rightarrow v$; por otra parte, en sentido contrario, si c se evalúa a v mediante semántica natural $c \Rightarrow v$, si e se compila a c mediante semántica natural $e \Downarrow c$, entonces e se evalúa a v mediante semántica natural $e \Rightarrow v$. Esto nos lleva a preguntarnos en general ¿por qué usar la semántica natural como marco? y, en específico, en comparación con nuestra corrección de referencia, a las preguntas: ¿por qué especificar la traducción como $e \Downarrow c$ y no como $[\![e]\!]$? y ¿por qué usar semántica de paso largo para especificar la semántica de la máquina? En cuanto a nuestra pregunta general, como respuesta inmediata, podemos dar las siguientes razones:

- El uso de la semántica natural como marco, al dar uniformidad, facilita y simplifica la tarea de establecer y razonar sobre las diferentes propiedades que pudiera tener la especificación de un compilador.

- En general, puede facilitar al usuario la lectura de la especificación de un compilador, en particular, la de un diseñador o un escritor de compiladores.

- Debido a su simpleza, intuitividad y claridad puede facilitar que nuevos usuarios exploren el área de verificación de compiladores.

- El que no sea necesario trabajar con otros formalismos pudiera llegar a traducirse, por una parte, en menos tiempo para escribir la especificación de un compilador y, por otra, en menos tiempo para mecanizar una especificación. De ser así, menos tiempo, significa menos tiempo en recursos humanos altamente calificados y, por tanto, se esperaría que esto derivara en menos recursos económicos.

- La semántica natural ha demostrado ser útil en la especificación, verificación y mecanización de compiladores [63, 71, 25, 9] y en la obtención de intérpretes eficientes [9].

Una reflexión más profunda revela razones quizás más esenciales y nuestras intenciones últimas. La semántica natural puede ser vista como una lógica basada en el sistema de deducción natural [55]; de aquí que, nuestra intención es usar la conexión de la semántica natural con la lógica como puente para interpretar, encontrar usos, de la lógica en computación, específicamente en la verificación de compiladores. Por ejemplo, en el campo de la lógica existen sistemas de deducción natural para lógicas modales [94], como la semántica natural está basada en deducción natural, es fácil pensar que la semántica natural se puede extender, con base en los resultados de la lógica, para convertirse en una semántica natural modal. Puesto que uno de los usos de las lógicas modales es la capacidad de expresar concurrencia, una semántica natural modal serviría para poder expresar concurrencia en la semántica de un lenguaje de

programación. Además, si se utiliza la semántica natural como marco, no solo serviría para expresar concurrencia en un lenguaje específico del compilador (por ejemplo, el lenguaje fuente o uno intermedio), sino en todos ellos, pero más aún, en principio, también serviría para expresar concurrencia en la máquina. Así, una de las principales razones de utilizar la semántica natural como marco es que si la semántica natural se extiende con una característica, esta extensión sirve, en principio, para todos los lenguajes del compilador incluyendo la máquina objetivo. En comparación, si se utiliza más de un formalismo en la especificación del compilador, se tendría que extender cada uno de los formalismos empleados con la característica deseada, lo que supone un esfuerzo mucho mayor. Por otra parte, la semántica natural no solo sirve como puente con la lógica, sino también como punto de encuentro con otros mecanismos computacionales basados en deducción natural. Por ejemplo, el análisis sintáctico como deducción (*parsing as deduction*) [88, 98] está basado en deducción natural, por lo que no es difícil vislumbrar que ambos formalismos: semántica natural y análisis sintáctico como deducción se puedan unificar en un único formalismo. Siguiendo la línea de pensamiento derivada de este análisis, como instancia de característica nueva a soportar en el marco de semántica natural, en este trabajo se extiende la semántica natural, como originalmente la introdujo Kahn [55], con la capacidad de expresar, en cada uno de los lenguajes de un compilador y en la máquina objetivo, evaluaciones infinitas.

Pasando ahora a nuestras preguntas específicas ¿por qué especificar la traducción como $e \Downarrow c$ y no como $[\![e]\!]$? previo a dar una respuesta, debemos tener claro que $[\![e]\!]$ denota una función, mientras que $\Rightarrow$ denota una relación; Kahn [55] menciona que las traducciones de un lenguaje a otro están fuertemente guiadas por la estructura del lenguaje por la que la semántica natural es muy adecuada para su especificación. En cuanto a ¿por qué usar semántica de paso largo para especificar la semántica de la máquina? Kahn [55] señala que de esta forma la equivalencia de ciertas secuencias de código se pueden demostrar fácilmente y esto es de utilidad en optimizaciones de código; adicionalmente, se sabe que usando semántica natural se pueden obtener intérpretes eficientes [9], por lo que, en principio, así se obtendrían intérpretes eficientes de la máquina. Por supuesto, en ambas respuestas, adicionalmente, uno de los principales motivos es poder usar la semántica natural como marco por todas las razones que hemos discutido anteriormente.

Analicemos ahora otras cuestiones retomando nuestra corrección de referencia. Notemos que en la corrección de referencia se debe garantizar tanto que partiendo del lenguaje fuente se preserve la semántica en el lenguaje objetivo, simulación hacia adelante; como el sentido contrario, es decir, que partiendo del lenguaje objetivo se preserve la semántica en el lenguaje fuente, simulación hacia atrás. No obstante, Leroy [62, 69] señala que cumplir ambas usualmente es demasiado fuerte como para ser utilizada como noción de preservación semántica en la práctica. Por otra parte, menciona que las simulaciones hacia adelante son significativamente más fáciles de realizar, pero el punto más interesante es que afirma que cuando el lenguaje objetivo es determinista la simulación hacia adelante implica el sentido contrario, es decir, la simulación hacia atrás. Por tanto, cuando se hace simulación hacia el frente y el lenguaje objetivo es determinista, no es necesario hacer simulación hacia atrás. Por

esta razón, debido a que en este trabajo el lenguaje objetivo de las traducciones que presentamos es determinista, únicamente presentamos los teoremas correspondientes a la simulación hacia adelante.

Por otro lado, observemos que en la corrección de referencia se debe considerar que se cumplan dos propiedades: la primera es la preservación semántica y la segunda es que el compilador siempre termine. Analicemos la segunda, para toda expresión e la compilación $[\![e]\!]$ siempre debe terminar, se puede observar que si $[\![e]\!]$ es total y está bien definida, mediante recursión estructural, sobre cada uno de los constructores del lenguaje fuente, $[\![e]\!]$ siempre terminará. En nuestro trabajo, esta propiedad se verifica en Coq (como se muestra en la Sección 3.1.3), aunque nosotros expresamos la compilación como $e \Downarrow c$ debido al uso de semántica natural como marco. Estudiemos ahora la primera, si una expresión del lenguaje fuente e se evalúa a un valor final v mediante semántica natural $e \Rightarrow v$, entonces la compilación de e, $[\![e]\!]$, también se evalúa a v mediante el cierre reflexivo y transitivo de la semántica de paso corto de la máquina $[\![e]\!] \overset{*}{\to} v$. Aquí, debemos notar que, únicamente se está garantizando la preservación semántica en el caso en que la expresión del lenguaje fuente e tenga una evaluación finita, es decir, en el caso de terminación. Por ejemplo, si la expresión es $5 + 2$ su evaluación terminará con el valor 7, o sea $5 + 2 \Rightarrow 7$, así la corrección de referencia garantiza que su compilación también se evaluará a 7, es decir $[\![5 + 2]\!] \overset{*}{\to} 7$. En cambio, si la evaluación de e no termina, es decir, es infinita, entonces no se garantiza nada acerca del comportamiento de su compilación, puede ser que no termine pero también podría ser que termine con un valor cualquiera. Por ejemplo, si tenemos $(\lambda x.x\ x)(\lambda x.x\ x)$ su evaluación será infinita pero la corrección de referencia no garantiza nada acerca de cuál será el comportamiento de su compilación, puedes ser que su evaluación sea infinita, es decir, que $[\![(\lambda x.x\ x)(\lambda x.x\ x)]\!] \overset{*}{\to} \ldots$, pero también puede ser que termine con un valor cualquiera, por ejemplo 4, es decir $[\![(\lambda x.x\ x)(\lambda x.x\ x)]\!] \overset{*}{\to} 4$. Lo que se quisiera es que si la evaluación de la expresión del lenguaje fuente es infinita, es decir, en el caso de no terminación, entonces la evaluación de su compilación también sea infinita, dicho de otro modo, lo que se quisiera es que se preservara la semántica también en el caso de no terminación. El problema es que la semántica natural, como la introdujo Kahn [55], únicamente es capaz de expresar evaluaciones finitas, nosotros extenderemos la semántica natural como marco para que sea capaz de expresar evaluaciones infinitas tanto en el lenguaje fuente, como en los lenguajes intermedios y también en la máquina objetivo; con este fin, tomaremos como base el trabajo de Leroy [63]. Leroy introduce la semántica natural coinductiva para expresar evaluaciones infinitas en el lenguaje fuente, de este modo, Leroy establece la preservación semántica en el caso de no terminación de la siguiente manera: si una expresión del lenguaje fuente e tiene una evaluación infinita mediante semántica natural coinductiva $e \overset{\infty}{\Rightarrow}$, entonces su compilación $[\![e]\!]$, también tiene una evaluación infinita mediante semántica de paso corto coinductiva de la máquina $[\![e]\!] \overset{\infty}{\to}$. Partiendo de este esquema, nosotros avanzaremos el trabajo de Leroy hasta llegar a la semántica natural coinductiva como marco para la preservación semántica en el caso de no terminación. Así, tenemos que si la evaluación de una expresión del lenguaje fuente e tiene una evaluación infinita mediante semántica natural coinductiva $e \overset{\infty}{\Rightarrow}$, además si la expresión e se compila al código c mediante semántica natural $e \Downarrow c$, entonces el

código resultante de la compilación tiene una evaluación infinita mediante semántica natural coinductiva de la máquina $c \overset{\infty}{\Rightarrow}$. Nótese que en comparación con el trabajo de Leroy, nosotros utilizamos semántica natural para especificar la compilación $e \Downarrow c$, en lugar de una función $[\![e]\!]$; más interesante es el hecho que introducimos el uso de semántica natural coinductiva para expresar evaluaciones infinitas en una máquina. A la preservación semántica tanto en el caso de terminación como en el caso de no terminación Leroy [63] la denomina corrección total. De este modo, en este trabajo presentaremos la semántica natural (coinductiva) como marco de verificación de la corrección total de compiladores.

Una vez hecho un estudio preliminar del problema que se trata en este trabajo, en la siguiente sección se introduce el problema más propiamente y la solución propuesta en este trabajo.

1.2. Problema

Esta tesis trata el problema de la verificación de compiladores en asistentes de demostración; específicamente, el problema de la verificación de la corrección total de compiladores de lenguajes funcionales en Coq. Aquí nos referimos a la corrección total en el sentido de Leroy [66] y Grégoire y Leroy [46], es decir, como la corrección de los cómputos que terminan y la corrección de los cómputos que no terminan. Tradicionalmente, en la literatura se usan verificaciones ad-hoc, es decir, aquellas en las que se emplea más de un formalismo. Esta situación clama por una solución que abstraiga y unifique todo lo necesario en un solo marco, el cual simplifique (e incluso, quizás, posibilite la automatización de) esta tarea.

La semántica natural, tal como la introdujo Kahn [55], nació siendo un formalismo unificador en el que se puede expresar la semántica del lenguaje fuente, la semántica de los lenguajes intermedios, la semántica de la máquina abstracta y las traducciones. Despeyroux [38] mostró, además, cómo la semántica natural permite demostrar la corrección de las traducciones. Basado en el trabajo de Kahn y Despeyroux, Boutin [20] formalizó un compilador de Mini-ML a la CAM en Coq (aunque no es posible obtener un compilador verificado a partir de su formalización).

Se sabe que la semántica natural es incapaz de expresar cómputos que no terminan (los trabajos de Kahn, Despeyroux y Boutin tratan únicamente los cómputos que terminan). Si bien Leroy [63] y Leroy y Grall [71] introducen la semántica natural coinductiva como formalismo para expresar cómputos que no terminan, no utilizan la semántica natural coinductiva como marco en el sentido original de Kahn, sino que únicamente la usan para especificar la semántica del lenguaje fuente (un lenguaje de alto nivel). Concretamente, hacen una formalización ad-hoc, en la que, además, utilizan semántica de paso corto en la que después sería nombrada la máquina SECD moderna (*Modern SECD machine*, MSECD) [66], además de una función para definir la compilación.

La idea principal de este trabajo consiste en extender la semántica natural a fin de que se convierta en un marco simple, fácil e intuitivo para la verificación de la corrección total de compiladores en Coq (con la propiedad de que permita obtener

un compilador verificado que se pueda usar en la vida real).

La estrategia para llevar a cabo esta extensión es la siguiente: hasta el momento, tomando como referencia el trabajo de Boutin, la semántica natural se puede utilizar para la formalización de la corrección de un compilador en Coq (únicamente para el caso de terminación, y no se puede obtener un compilador verificado). Lo primero que haremos será mostrar cómo, a partir de una especificación en semántica natural, se puede obtener un intérprete (Sección 3.1.1), o bien un compilador (Sección 3.1.3) correcto (*sound*) y completo (*complete*) con respecto a dicha especificación, según corresponda. Luego, para que la semántica natural sea capaz de expresar cómputos que no terminan, utilizaremos la semántica coinductiva de Leroy. No obstante, esta semántica únicamente se ha utilizado en el lenguaje fuente; nosotros, además, mostraremos cómo utilizar la semántica natural coinductiva para especificar los cómputos que no terminan en una máquina abstracta (Sección 4.2.2.1). También mostraremos que un intérprete que se obtuvo previamente a partir de una semántica natural es correcto y completo con respecto a la especificación en semántica natural coinductiva correspondiente (Sección 4.1.1), es decir, bajo el criterio de no terminación (y no únicamente para el caso de terminación). De esta manera, estamos listos para formular y demostrar la corrección (preservación semántica) de un compilador, tanto para el caso de terminación (Sección 3.1.4) como para el caso de no terminación (Sección 4.1.3), completamente en términos de semántica natural (coinductiva).

Para ilustrar el uso de semántica natural (coinductiva) como marco, mecanizamos un compilador de Mini-ML a una versión de paso largo de la máquina SECD moderna. Nos interesaba mostrar que, en particular, la semántica natural coinductiva es capaz de expresar tanto traducciones entre lenguajes de alto nivel como traducciones de un lenguaje de alto nivel a código de máquina (un lenguaje de bajo nivel). Es por esta razón que nuestro compilador se compone de dos etapas (ver Figura 1.1): traducción de Mini-ML con nombres a Mini-ML con índices de de Bruijn (Secciones 3.1 y 4.1), como instancia de la primera, y generación de código de la máquina SECD moderna (versión de paso largo) a partir de Mini-ML con índices de de Bruijn (Secciones 3.2.2 y 4.2.2), como instancia de la segunda. Por simplicidad, escribiremos MiniML para referirnos a Mini-ML con nombres y $\mathsf{MiniML}^{\mathsf{dB}}$ para referirnos a Mini-ML en notación de de Bruijn.

Por otro lado, en la literatura se utiliza tradicionalmente semántica de paso corto en la máquina. Es por ello que, como máquina objetivo alternativa, ofrecemos la máquina SECD moderna original de paso corto (Secciones 3.2.1 y 4.2.1) extendida para dar soporte a todo el lenguaje Mini-ML, en particular con soporte nativo de recursión, con el fin principal de compararla con nuestra solución, en la que se utiliza una máquina de paso largo, es decir, nuestra versión de paso largo de la máquina SECD moderna.

1.3. Contribuciones

Nuestras principales contribuciones generales y específicas son:

- La semántica natural coinductiva como marco para la verificación de la correc-

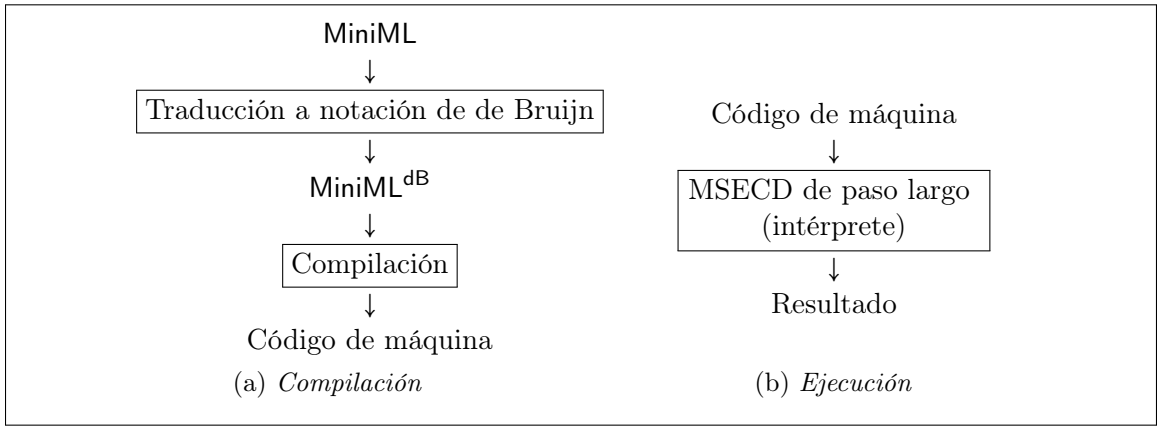


Figura 1.1: Arquitectura del compilador.

ción total de compiladores en Coq, tal que se puede obtener como producto final un compilador verificado utilizable de forma independiente, es decir, un compilador correcto y completo con respecto a una especificación en semántica natural (coinductiva) y correcto con respecto a la traducción especificada

- Un método sistemático para obtener ya sea un intérprete (Secciones 3.1.1 y 4.1.1) o un compilador (Sección 3.1.3) correcto y completo, según corresponda, a partir de un especificación en semántica natural (coinductiva)

- El uso de la semántica natural coinductiva para especificar los cómputos que no terminan en una máquina abstracta (Sección 4.2.2.1)

- Un compilador de Mini-ML a una versión de paso largo de la máquina SECD moderna, incluyendo la verificación de su corrección total en Coq (Secciones 3.1, 4.1, 3.2.2 y 4.2.2)

- Una versión extendida de la máquina SECD moderna original de paso corto que incluye soporte nativo de recursión (Secciones 3.2.1 y 4.2.1)

- Una versión de paso largo de la máquina SECD moderna, incluyendo su formalización en Coq (Secciones 3.2.2 y 4.2.2)

- Una especificación en semántica natural coinductiva de los cómputos que no terminan en Mini-ML (versión extendida de la especificación de Leroy del cálculo lambda puro con constantes, Sección 4.1.1)

- Una especificación en semántica natural coinductiva de los cómputos que no terminan en la máquina SECD moderna de paso largo (Sección 4.2.2.1)

- Un algoritmo para traducir de (una representación abstracta de) la especificación en semántica natural (coinductiva) de un compilador correcto total a su correspondiente formalización en Coq (Capítulo 5)

Comenzaremos dando una exposición del trabajo relacionado (Capítulo 2). La idea general de la presentación es primero atacar el caso de la terminación utilizando semántica natural (Capítulo 3), y luego el caso de la no terminación utilizando semántica natural coinductiva (Capítulo 4). Durante este trabajo, la estrategia que seguiremos será presentar en abstracto cada uno de los componentes del compilador junto con su correspondiente formalización en Coq; de esta manera, se busca que al finalizar nuestro compilador se cuente con la intuición necesaria que está detrás del algoritmo para pasar de un compilador correcto total en abstracto a Coq (Capítulo 5). Finalmente (Capítulo 6), presentamos nuestras conclusiones y una perspectiva del posible trabajo futuro.

A lo largo de este trabajo iremos presentando fragmentos de código correspondientes al desarrollo en Coq, el código completo se puede ver en [116]. Asimismo, ofreceremos bosquejos de las demostraciones de los lemas y teoremas que iremos presentando, la demostración completa, detallada paso por paso, de cada uno de éstos, se puede consultar en el apéndice A.

Capítulo 2

Verificación de compiladores en Coq

En este capítulo daremos un panorama de la verificación de compiladores en Coq. Desde la fundación del proyecto CompCert C [65, 62, 64, 19, 18] liderado por Leroy, ha habido un gran progreso en la literatura dedicada a la verificación de compiladores utilizando asistentes de demostración, en particular en Coq. En este trabajo, se trata específicamente la verificación de compiladores de lenguajes funcionales; en principio, la verificación de compiladores de lenguajes funcionales a máquinas abstractas.

2.1. Técnicas de verificación

Una técnica usual, expuesta por Hardin et al. [47], para llevar a cabo la verificación de un compilador de un lenguaje funcional a una máquina abstracta, es utilizar semántica de paso corto, tanto en el lenguaje fuente como en la máquina abstracta, junto con una función de decompilación y una medida para establecer la corrección. La idea de esta técnica es realizar una simulación de abajo hacia arriba en la que toda transición de la máquina corresponde a cero o una reducción en el lenguaje fuente. Por tanto, se establece una correspondencia inversa donde los estados de la máquina se transforman hacia arriba en expresiones del lenguaje fuente mediante la función de decompilación. Más precisamente, si partiendo de un estado de máquina s se alcanza un estado s' vía una transición de máquina $s \to s'$, y e es la expresión del lenguaje fuente que corresponde al estado s vía decompilación, existe una expresión e' que corresponde a s' vía decompilación, tal que $e = e'$ o e se reduce a e' vía la semántica de paso corto del lenguaje fuente $e \to e'$. Cuando la máquina realiza una transición de un estado a otro y la decompilación de ambos estados corresponde a la misma expresión en el lenguaje fuente, se dice que la máquina realiza una *transición silenciosa* (*silent transition*). Para garantizar que no existe un número infinito de transiciones silenciosas, se utiliza una medida que se define sobre los estados de la máquina, es decir, si $s \to s'$ y la decompilación de s y s' corresponde a la misma expresión e, la medida de s es mayor que la de s'.

Grégoire y Leroy [46] y Grégoire [45] utilizan esta técnica para verificar el compilador de un cálculo lambda con reducción fuerte a una máquina abstracta en Coq; más precisamente, para verificar la corrección de un compilador del cálculo de cons-

trucciones inductivas (CIC) a una variante de la máquina ZAM [70] (adaptada para soportar reducción simbólica débil), de donde se obtiene una implementación verificada basada en un compilador para evaluar términos de Coq. Adicionalmente, Grégoire y Leroy muestran que esta implementación basada en un compilador es más eficiente que el intérprete original de Coq (tal como se esperaba). Más recientemente, Kunze et al. [59] emplean una técnica muy similar para verificar la corrección de un compilador de un cálculo lambda con llamada por valor a una máquina abstracta en Coq.

No obstante, Leroy [63, 66] y Leroy y Grall [71] subrayan la dificultad de una demostración de la corrección utilizando esta técnica, y también señalan que la definición de una decompilación es complicada, y no resulta fácil razonar sobre ella ni extenderla (especialmente en las fases de optimización de un compilador). En consecuencia, proponen una solución basada en semántica de paso largo; en efecto, mencionan que la motivación original de su trabajo es demostrar, utilizando semántica de paso largo, la preservación semántica de compiladores.

La técnica de Leroy [63] y Leroy y Grall [71] consiste en usar semántica de paso largo (coinductiva) en el lenguaje fuente, pero semántica de paso corto en la máquina. De esta forma, para el caso de terminación, si una expresión del lenguaje fuente e se evalúa a v vía semántica de paso largo $e \Rightarrow v$, entonces reducir el código de máquina c vía el cierre transitivo de la semántica de paso corto $\stackrel{+}{\to}$ lleva a la máquina a un estado con v_m en la cima de la pila, donde c corresponde a la compilación de e y v_m es el valor de máquina correspondiente a v. Para el caso de no terminación, si e diverge usando semántica de paso largo coinductiva, entonces c también diverge en la máquina. Leroy y Grall mencionan que su técnica provee una forma más simple de demostrar la preservación de la semántica, en particular para el caso de no terminación.

Actualmente, es bien conocido [63, 71, 25, 9] que el uso de la semántica de paso largo es más fácil y más conveniente para demostrar la corrección de compiladores, así como para obtener intérpretes eficientes [9]. Nuestro propósito es llevar a la semántica de paso largo a sus últimas consecuencias explotándola donde ha demostrado ser de utilidad. Es por esto que proponemos a la semántica natural (coinductiva) como marco para la verificación de compiladores.

La semántica natural (coinductiva) como marco para la verificación de compiladores en Coq tal como se propone en este trabajo es una técnica muy similar en espíritu a aquella de Leroy y Grall, pero va más lejos, ya que la semántica de paso largo no se utiliza únicamente en el lenguaje fuente, sino que también se utiliza en la máquina objetivo (recordemos que Leroy y Grall emplean semántica de paso corto en la máquina). Más aún, para el conocimiento del autor, es la primera vez que la semántica natural coinductiva se propone y utiliza para definir cómputos que no terminan en una máquina abstracta. De este modo, se obtiene una técnica completamente basada en semántica natural (coinductiva) para verificar la corrección de compiladores de lenguajes funcionales a máquinas abstractas en Coq.

Establecer la corrección es incluso más fácil, intuitivo y simple, ya que la semántica natural también se usa en la máquina. Si una expresión del lenguaje fuente e se evalúa a un valor v vía la semántica natural del lenguaje fuente $e \Rightarrow v$, entonces c se evalúa al estado de máquina final que tiene a v_m en la cima de la pila vía la semántica natural de la máquina $s \vdash c \Rightarrow v_m \cdot s$; donde c es la compilación de e vía semántica

natural $e \Downarrow c$, v_m es la compilación de v vía semántica natural $v \Downarrow v_m$, y s es una pila cualquiera. Si e diverge vía semántica natural coinductiva $e \overset{\infty}{\Rightarrow}$, entonces c también diverge vía la semántica natural coinductiva de la máquina $s \vdash c \overset{\infty}{\Rightarrow}$. Aquí notamos que la semántica natural (coinductiva) es suficiente para establecer la corrección: no se requiere de ningún otro formalismo distinto.

Un uso potencial de este marco es tomarlo como base para verificar una implementación convencional, un compilador que genera código de una máquina abstracta, (del núcleo) de un lenguaje funcional real como OCaml. La implementación oficial de OCaml del INRIA consta de dos compiladores [67]: el primero genera código de la máquina ZAM, mientras que el segundo genera código C--. Especulamos que este marco también se puede utilizar como base para verificar el compilador que genera C-- ya que Dargaye [36] ya usa semántica de paso largo (aunque no como marco unificador y únicamente considerando cómputos que terminan) para verificar un compilador de Mini-ML a Cminor (una representación intermedia temprana del compilador CompCert C). La idea de generar código Cminor (o algún otro lenguaje intermedio de CompCert C) en lugar de C-- es inmediata, ya que de esta manera se puede conectar la parte final (*back-end*) del compilador a CompCert C y obtener como resultado final código ensamblador verificado. Este uso toma una mayor relevancia si consideramos que Coq es un programa OCaml (si bien algunos fragmentos de Coq han sido verificados en el mismo Coq [101, 100, 11], el código OCaml verificado resultante de la extracción se ejecutará eventualmente en una implementación de OCaml).

2.2. Derivación de máquinas abstractas

Por su parte, una línea de investigación se dedica a derivar sistemáticamente una máquina abstracta partiendo de un cálculo lambda [5, 17, 14, 92, 99, 16, 15]. La idea general de estos trabajos es partir de un cálculo lambda y realizar una serie de transformaciones hasta obtener la máquina abstracta deseada. Aunque en estos trabajos se emplea una gran variedad de transformaciones, una de las más explotadas es reorientación (*refocusing*) [35]. Algunos, además, incluyen formalización en Coq [14, 92, 99, 16, 15]. Los trabajos más cercanos al nuestro son aquellos que partiendo de la semántica natural de un cálculo lambda derivan una máquina abstracta [5, 92]; específicamente, el trabajo más similar es [92]; en éste, la máquina STG se deriva a partir de la semántica natural de un cálculo lambda perezoso y la derivación se formaliza en Coq. No obstante, se tratan únicamente los cómputos que terminan.

En todos estos trabajos, el énfasis recae en la derivación de la máquina correspondiente. En contraste, en la implementación de un lenguaje funcional la máquina abstracta objetivo se diseña usualmente a mano y sólo después, si acaso, se demuestra que es correcta con respecto a la semántica del cálculo lambda fuente (véase, por ejemplo, [70]). Por consiguiente, la semántica natural (coinductiva) como marco, como se presenta en este trabajo, es más adecuada para verificar implementaciones de lenguajes funcionales (que generan código para una máquina abstracta), ya que en él se considera la máquina objetivo como dada (y no por ser derivada). De igual forma, en caso de haber lenguajes intermedios, éstos se consideran como dados (y no por ser

derivados).

Más aún, si por alguna razón (por ejemplo, la justificación semántica de la máquina objetivo) se considera relevante derivar sistemáticamente la máquina objetivo a partir del cálculo fuente, conjeturamos que la derivación correspondiente también se puede llevar a cabo en el marco de semántica natural. Esto se debe a que cada transformación que lleva paulatinamente a la máquina derivada se puede ver como una traducción intermedia y ser definida en semántica natural. Asimismo, el lenguaje de entrada y de salida de cada transformación se puede ver como un lenguaje intermedio y su semántica correspondiente puede definirse en semántica natural. Desde luego, la máquina abstracta derivada sería una máquina de paso largo.

2.3. Formalizaciones ad-hoc

Los siguientes trabajos están dedicados a la formalización de un pequeño lenguaje funcional en Coq, pero ninguno de ellos utiliza semántica natural como marco unificador para todos los componentes del compilador; en su lugar, utilizan verificaciones ad-hoc: Chlipala [26] presenta un compilador de un pequeño lenguaje funcional impuro a un lenguaje ensamblador idealizado; usa notación de de Bruijn y emplea semántica natural para el lenguaje fuente y el objetivo, pero no para especificar la compilación. Benton y Hur [13] tratan la compilación de un pequeño lenguaje funcional con tipos a la máquina SECD, pero utilizan semántica denotacional para el lenguaje fuente y semántica de paso corto en la máquina, además de que emplean una relación indexada por paso biortogonal (*biorthogonality step-indexed logical relation*) para establecer la corrección. Dargaye [36] entrega un compilador de Mini-ML a Cminor (una representación intermedia temprana del compilador CompCert C [65]), pero no está diseñado para ser una implementación de Mini-ML de propósito general independiente; en su lugar, fue concebida para trabajar únicamente con el código generado por el mecanismo de extracción de Coq. Leroy [63, 71, 66] ofrece un compilador verificado del cálculo lambda puro extendido con constantes a la máquina SECD moderna; comienza con notación de de Bruijn y utiliza semántica natural para el lenguaje fuente, pero recurre a semántica de paso corto para la máquina y una función para definir la compilación.

2.4. CertiCoq

El proyecto CertiCoq [7, 96, 12, 83] busca proveer una secuencia de extracción verificada del lenguaje núcleo de Coq, Gallina, a lenguaje de máquina. Por lo tanto, en CertiCoq únicamente tiene sentido abordar cómputos que terminan. Esta situación se menciona explícitamente en [7]: "... we can restrict our reasoning to terminating programs since Coq is strongly normalizing. This way we avoid backward simulations (forward simulations proofs are much simpler) and avoid proving preservation of divergence". De forma similar, Savary Bélanger [96] señala: "In CertiCoq, we are only concerned with terminating programs: Gallina is strongly normalizing, and our proof of correctness ensures that programs do not acquire non-terminating behaviors along the way".

En lugar de producir directamente código de máquina, CertiCoq genera código de Clight (un lenguaje intermedio de CompCert C). Por tanto, CertiCoq utiliza CompCert C como parte final verificada del compilador (*verified compiler back-end*) para producir código de máquina. De esta manera, el compilador CertiCoq realiza una serie de fases de Gallina a Clight. En CertiCoq, la semántica de los lenguajes (fuente, objetivo e intermedios) así como las demostraciones de corrección (para el caso en que los cómputos terminan) están basadas en semántica de paso largo. Sin embargo, la semántica de paso largo se refina con otras nociones tales como relaciones lógicas indexadas por paso (*step-indexed logical relations*) y semánticas basadas en contextos (*context-based semantics*) [84, 96]; esto con el fin de poder contar con propiedades adicionales, por ejemplo, composicionalidad. Adicionalmente, la posible adopción de esta técnica para lenguajes de propósito general apenas se menciona, de forma breve, en [96]; aunque para este propósito Savary Bélanger [96] sugiere que se emplee semántica de paso corto. Por su parte, Paraskevopoulou y Appel [84] y Paraskevopoulou [83], en la demostración de la corrección de la conversión a cerraduras (*closure conversion*), ya extienden esta técnica para tratar bajo ciertas condiciones los cómputos que no terminan. La conversión a cerraduras es una fase que se realiza en CertiCoq.

Nuestro marco de semántica natural (coinductiva) es más adecuado para verificar implementaciones usuales de lenguajes funcionales a máquinas abstractas, ya que en él se consideran ambos tipos de cómputos: los que terminan y los que no terminan (corrección total). Asimismo, en él se pueden expresar los cómputos que terminan y los que no terminan en una máquina abstracta. En contraste, por diseño [7] CertiCoq únicamente considera, por un lado, evaluaciones que terminan y, por el otro, genera código de Clight, de modo que no se utiliza una máquina abstracta. Esta situación evidencia que nuestro marco de semántica natural (coinductiva) y CertiCoq persiguen diferentes objetivos. Mientras que nuestro marco de semántica natural (coinductiva) es un marco para conducir la verificación total de la corrección de compiladores en Coq, CertiCoq es un compilador verificado (del cálculo núcleo de Coq a Clight), cuyos objetivos principales explícitos no incluyen ofrecer una infraestructura para la verificación de compiladores [7], incluso cuando la infraestructura y técnicas desarrolladas para verificar el compilador CertiCoq pueden adaptarse con este propósito.

2.5. Semántica

Las siguientes son semánticas relacionadas: semántica operacional de paso largo coinductiva (*coinductive big-step operational semantics*) [63, 71], semántica operacional coinductiva basada en trazas (*trace-based coinductive operational semantics*) [76], semántica de paso muy largo (*pretty-big-step semantics*) [25] y semántica de paso largo basada en banderas (*flag-based big-step semantics*) [9].

De estos trabajos, el único que presenta la verificación de la corrección de un compilador es el de Leroy (una verificación ad-hoc), lo cual significa que en ninguno de ellos se utiliza semántica natural (coinductiva) como marco para la verificación de la corrección de compiladores. Específicamente, no se utiliza en la definición de la

semántica de la máquina (ni en la de su intérprete), se prescinde de ella en la definición de las traducciones y no se usa (tanto en el lenguaje fuente como en el lenguaje objetivo) para establecer, ni para demostrar, la corrección de las traducciones. Lo que sí se hace en cada uno de ellos es presentar una semántica natural con coinducción de un lenguaje de alto nivel (que correspondería, usualmente, al lenguaje fuente en un compilador) y es este aspecto el que revisaremos a continuación.

Leroy [63], primero, expresa por separado los cómputos finitos empleando semántica natural "evaluación" (*evaluation*) y los cómputos infinitos "divergencia" (*divergence*) usando semántica natural coinductiva; esta solución es clara y limpia. Después, ofrece una solución alternativa; en ésta se expresan cómputos finitos y cómputos infinitos en una única semántica natural coinductiva "coevaluación" (*coevaluation*). No obstante, esta semántica no se comporta bien, ya que, por una parte, hay cómputos infinitos que no es capaz de expresar, y, por otra, existen cómputos infinitos que se evalúan a un valor cualquiera v. Nakata y Uustalu [76] señalan que este comportamiento parece ser accidental e indeseable.

Nakata y Uustalu [76] definen una semántica natural coinductiva del lenguaje While que expresa cómputos finitos y cómputos infinitos; el diseño, cuidadoso y ad-hoc de esta semántica sigue, en su interior, el de la semántica de paso corto. Adicionalmente, Nakata y Uustalu definen un intérprete utilizando la mónada de traza (*trace monad*) y muestran que es correcto con respecto a dicha semántica. El trabajo de Uustalu y Nakata [76] es el único de los trabajos relacionados expuestos aquí en el que se presenta un intérprete.

Charguéraud [25], introduce la semántica de paso muy largo, una semántica basada en coevaluación de Leroy; desafortunadamente, la semántica de paso muy largo hereda el mal comportamiento de coevaluación. A su vez, Bach Poulsen y Mosses [9] definen la semántica de paso largo basada en banderas con base en la semántica de paso muy largo; infortunadamente, la semántica de paso largo basada en banderas, por medio de la semántica de paso muy largo, también hereda el mal comportamiento de coevaluación.

En este trabajo hemos presentado la semántica natural (coinductiva) como marco para la verificación de la corrección total de compiladores en Coq. Una vez que contamos con una solución simple, fácil, clara e intuitiva para esta tarea, podemos buscar mejorarla. En particular, en los intérpretes hemos utilizado un parámetro natural para acotar la recursión. Leroy [68] ha definido recientemente un intérprete del lenguaje While utilizando la mónada de parcialidad (*partiality monad*); nosotros pensamos adoptar la mónada de parcialidad en nuestro compilador de Mini-ML, y en el marco en general, para evitar el uso de este parámetro.

Por otra parte, se busca llegar a una única semántica coinductiva " $\overset{\omega}{\Rightarrow}$ " capaz de expresar cómputos que terminan y cómputos que no terminan. Charguéraud [25] menciona que, en principio, esta semántica se podría utilizar directamente para demostrar la corrección total de las traducciones, no obstante, señala que la conclusión en el teorema de corrección, usualmente es de la forma $\exists v'.(v \approx v') \wedge (c \overset{\omega}{\Rightarrow} v')$, y que, el soporte actual de coinducción en Coq, únicamente permite el uso de predicados coinductivos en la conclusión, en particular, no permite el uso del cuantificador existencial "$\exists$" ni del conectivo "$\wedge$" cuando se realiza una demostración por coinducción.

Bach Poulsen y Mosses [9] dirigen una crítica similar al soporte actual de coinducción en Coq. Afortunadamente, nuestra semántica natural (coinductiva) es ideal aquí, ya que, al utilizar semántica natural (inductiva) "$\Rightarrow$" para expresar cómputos finitos y semántica natural coinductiva "$\overset{\infty}{\Rightarrow}$" para expresar cómputos infinitos (por separado), la demostración del caso de terminación, donde la conclusión requiere de un $\exists$ y de un $\wedge$, se puede hacer por inducción, mientras que, de esta forma, en el caso de no terminación, en la conclusión no se requiere ni del $\exists$ ni del $\wedge$, únicamente se utiliza el predicado coinductivo $\overset{\infty}{\Rightarrow}$, por lo que se puede demostrar por coinducción (con el soporte actual de Coq).

Aun así, se podría buscar una única semántica $\overset{\omega}{\Rightarrow}$ para tener una definición más concisa; en tal caso, el marco podría automatizar la traducción de ésta a las dos semánticas por separado ($\Rightarrow$ y $\overset{\infty}{\Rightarrow}$), además de establecer y demostrar la equivalencia entre la primera y la unión de las dos últimas. Una vez que se cuenta con estas dos semánticas, se pueden utilizar los resultados actuales del marco.

El problema central es que, hasta donde el autor conoce, a la fecha, no existe en la literatura, ninguna semántica natural coinductiva que exprese cómputos finitos y cómputos infinitos, y que, se comporte bien en Coq. El autor, con base en coevaluación de Leroy, ha logrado definir una semántica natural coinductiva (del cálculo lambda puro extendido con constantes) que expresa cómputos que terminan y cómputos que no terminan y que se comporta bien en Coq. Adicionalmente, ha demostrado la equivalencia de dicha semántica con la unión de las dos semánticas que expresan los cómputos finitos y los cómputos infinitos por separado ($\Rightarrow$ y $\overset{\infty}{\Rightarrow}$). Al parecer, este resultado es correcto [61] y se piensa presentar en trabajos futuros.

Para continuar, se podría atacar el problema de reducir el número de reglas necesarias en una definición en semántica natural coinductiva. Este es el objetivo principal de la semántica de paso muy largo y de la semántica de paso largo basada en banderas. Incluso, de haber en estas semánticas resultados que representan una mejora en ese sentido para nuestro marco, pueden ser adoptados en el mismo.

Capítulo 3

Semántica natural

En este capítulo, atacaremos el caso en que los cómputos terminan utilizando semántica natural. Primero expondremos la traducción a índices de de Bruijn y, después, la generación de código.

3.1. Traducción a índices de de Bruijn

En esta sección, ofrecemos la traducción de MiniML a $\mathsf{MiniML}^{\mathsf{dB}}$, la cual está basada en la ofrecida por el autor en [117].

3.1.1. MiniML

Para comenzar, presentamos el lenguaje fuente, MiniML, dado por el autor en [117], que es esencialmente el cálculo lambda puro extendido con naturales, booleanos, operadores aritméticos y de comparación, definiciones locales, condicionales y recursión nativa por medio de un operador de punto fijo. Su sintaxis abstracta es la siguiente:

e	::=	n	Naturales
	|	b	Booleanos
	|	$e \star e$ con $\star \in \{+, -, *, =\}$	Expresiones aritméticas y de comparación
	|	x	Variables
	|	`if` e `then` e `else` e	Condicionales
	|	`let` x = e `in` e	Definiciones locales
	|	$\lambda x.e$	Abstracción
	|	$\mu f.\lambda x.e$	Punto fijo
	|	$e\ e$	Aplicación

Antes de realizar la codificación de nuestras definiciones en Coq, conviene resaltar algunas características de éste. Coq está basado en el cálculo de construcciones inductivas, el cual es un cálculo lambda con un sistema de tipos sofisticado y expresivo. Al tratarse de un cálculo lambda, se puede utilizar como una lógica, pero también como un lenguaje de programación, es decir, se pueden demostrar proposiciones, pero también se pueden escribir programas.

Esta distinción se hace explícita utilizando los tipos `Prop` y `Set`, respectivamente. A grandes rasgos, cuando un término en Coq tiene tipo `Prop`, se usa como lógica, mientras que si un término tiene tipo `Set` se usa como lenguaje de programación. En realidad, esta distinción sintáctica explícita entre `Prop` y `Set` se introdujo debido al mecanismo de extracción de Coq [85] para distinguir aquellos términos con contenido computacional de aquellos sin él (Paulin-Mohring [85] llama "Spec" a lo que más adelante se le denominó "Set"). De esta manera, si un término en Coq tiene tipo `Set`, el mecanismo de extracción puede generar un programa en un lenguaje de programación general, como OCaml, correspondiente a este término. Por el contrario, si un término tiene tipo `Prop`, no se puede extraer programa alguno.

La sintaxis abstracta de MiniML se puede codificar en Coq como sintaxis abstracta de primer orden utilizando una definición `Inductive` con tipo `Set` de la siguiente manera:

```
Inductive MML_exp: Set :=
| Const : nat → MML_exp
| ...
| Letm: nid → MML_exp → MML_exp → MML_exp
| ...
| Lam: nid → MML_exp → MML_exp
| Mu: nid → nid → MML_exp → MML_exp
| App: MML_exp → MML_exp → MML_exp.
```

De aquí se puede utilizar el mecanismo de extracción de Coq con el siguiente comando:

```
Extraction MML_exp.
```

lo cual da como resultado:

```
type mML_exp =
| Const of nat
| ...
| Letm of nid * mML_exp * mML_exp
| ...
| Lam of nid * mML_exp
| Mu of nid * nid * mML_exp
| App of mML_exp * mML_exp
```

De esta forma, podemos notar cómo una definición `Inductive` con tipo `Set` en Coq corresponde a un ADT en OCaml; en este caso, a la sintaxis abstracta de MiniML.

Para definir la semántica natural de MiniML, primero definimos sus valores por medio de entornos y cerraduras:

$$
\begin{array}{rcll}
v & ::= & n & \text{Números} \\
 & | & b & \text{Booleanos} \\
 & | & (\lambda x.e)[\Gamma] & \text{Cerraduras} \\
 & | & (\mu f.\lambda x.e)[\Gamma] & \text{Cerraduras recursivas}
\end{array}
\qquad
\begin{array}{rcll}
\Gamma & ::= & [] & \text{Entorno vacío} \\
 & | & (x, v) \cdot \Gamma &
\end{array}
$$

Un entorno es una asociación de variables a valores, representado como una secuencia de pares (x, v), donde x es el nombre de una variable y v su valor correspondiente. Al respecto, el predicado $\Gamma \vdash x \longmapsto v$ expresa que, el valor de la variable x en el entorno Γ es v.

$$
\frac{}{(x, v) \cdot \Gamma \vdash x \longmapsto v}
\qquad
\frac{\Gamma \vdash x \longmapsto v \qquad x \neq y}{(y, w) \cdot \Gamma \vdash x \longmapsto v}
$$

La semántica natural de MiniML, tal como lo hace el autor en [117], se define por medio del siguiente predicado:

$$\Gamma \vdash e \Rightarrow v$$

que se lee: dada una expresión e y un entorno Γ, e se evalúa al valor v.

Naturalmente, bajo el supuesto de que Γ contiene el valor de las variables libres de e. La semántica de MiniML es la siguiente:

$$
\frac{}{\Gamma \vdash n \Rightarrow n}
\qquad
\frac{}{\Gamma \vdash b \Rightarrow b}
\qquad
\frac{\Gamma \vdash e_1 \Rightarrow n_1 \qquad \Gamma \vdash e_2 \Rightarrow n_2}{\Gamma \vdash e_1 \star e_2 \Rightarrow n_1 \star n_2}
$$

$$
\frac{\Gamma \vdash x \longmapsto v}{\Gamma \vdash x \Rightarrow v}
\qquad
\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad (x, v_1) \cdot \Gamma \vdash e_2 \Rightarrow v}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow v}
$$

$$
\frac{\Gamma \vdash e_1 \Rightarrow \mathit{true} \qquad \Gamma \vdash e_2 \Rightarrow v}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v}
\qquad
\frac{\Gamma \vdash e_1 \Rightarrow \mathit{false} \qquad \Gamma \vdash e_3 \Rightarrow v}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v}
$$

$$
\frac{}{\Gamma \vdash \lambda x.e \Rightarrow (\lambda x.e)[\Gamma]}
\qquad
\frac{}{\Gamma \vdash \mu f.\lambda x.e \Rightarrow (\mu f.\lambda x.e)[\Gamma]}
$$

$$
\frac{\Gamma \vdash e_1 \Rightarrow (\lambda x.e)[\Gamma_1] \qquad \Gamma \vdash e_2 \Rightarrow v_2 \qquad (x, v_2) \cdot \Gamma_1 \vdash e \Rightarrow v}{\Gamma \vdash e_1\ e_2 \Rightarrow v}
$$

$$
\frac{\Gamma \vdash e_1 \Rightarrow (\mu f.\lambda x.e)[\Gamma_1] \quad \Gamma \vdash e_2 \Rightarrow v_2 \quad (x, v_2) \cdot (f, (\mu f.\lambda x.e)[\Gamma_1]) \cdot \Gamma_1 \vdash e \Rightarrow v}{\Gamma \vdash e_1\ e_2 \Rightarrow v}
$$

En semántica natural, una definición puede verse como una proposición lógica inductiva, por lo que se puede codificar en Coq como una definición `Inductive` con tipo `Prop`. Así, la semántica de MiniML se puede escribir de la siguiente manera:

```
Inductive MML_NS : MML_env → MML_exp → MML_val → Prop :=
| MML_NS_Const: ∀ G:MML_env, ∀ n :nat,
    MML_NS G (Const n) (Num n)

| ...

| MML_NS_Let: ∀ G:MML_env, ∀ x:nid,∀ e1 e2:MML_exp, ∀ v1 v2:MML_val,
    MML_NS G e1 v1 →
    MML_NS ((x, v1):: G) e2 v2 →
    MML_NS G (Letm x e1 e2) v2

| ...

| MML_NS_Lam: ∀ G:MML_env, ∀ x:nid, ∀ e:MML_exp,
    MML_NS G (Lam x e) (Clos x e G)

| MML_NS_Mu: ∀ G:MML_env, ∀ f x:nid, ∀ e:MML_exp,
    MML_NS G (Mu f x e) (Closr f x e G)

| MML_NS_App: ∀ G G1:MML_env, ∀ x:nid, ∀ e1 e2 e:MML_exp, ∀ v v2:MML_val,
    MML_NS G e1 (Clos x e G1) →
    MML_NS G e2 v2 →
    MML_NS ((x, v2):: G1) e v →
    MML_NS G (App e1 e2) v

| MML_NS_Appr: ∀ G G1:MML_env, ∀ f x:nid, ∀ e1 e2 e:MML_exp, ∀ v v2:MML_val,
    MML_NS G e1 (Closr f x e G1) →
    MML_NS G e2 v2 →
    MML_NS ((x, v2)::( f,( Closr f x e G1)):: G1) e v →
    MML_NS G (App e1 e2) v.
```

Como una definición en semántica natural es, en general, una relación, es no determinista en el caso general. Es por esto que, si en un caso particular una definición de este tipo es determinista, debe establecerse formalmente un lema que enuncie esta propiedad, tal como lo hacemos para la semántica de MiniML de la siguiente forma:

```
Lemma MML_NS_deterministic:
∀ G, ∀ e, ∀ v1, MML_NS G e v1 →
          ∀ v2, MML_NS G e v2 →
                  v1 = v2.
```

En efecto, este resultado, en el caso de que se trate de un lenguaje funcional basado en el cálculo lambda, como es el caso de MiniML, también puede leerse como un resultado de confluencia.

Si una relación es determinista, entonces puede escribirse como una función; en consecuencia, se puede codificar como una función en un lenguaje de programación.

En Coq podemos escribir una función recursiva empleando `Fixpoint`; por ejemplo, la función que corresponde a la semántica natural de MiniML se escribe como sigue:

```
Fixpoint MML_NS_interpreter (depthR:nat) (G: MML_env) (e: MML_exp)
: option MML_val :=
match depthR with
| O ⇒ None
| S m ⇒
  match e with
    | Const n ⇒ Some (Num n)
    | ...
    | Letm x e1 e2 ⇒
      match MML_NS_interpreter m G e1 with
        | Some v1 ⇒ MML_NS_interpreter m ((x,v1):: G) e2
        | _ ⇒ None
      end
    | ...
    | Lam x e  ⇒ Some (Clos x e G)
    | Mu f x e  ⇒ Some (Closr f x e G)
    | App e1 e2 ⇒
      match MML_NS_interpreter m G e1 with
        | Some (Clos x e G1) ⇒
          match MML_NS_interpreter m G e2 with
            | Some v2 ⇒ MML_NS_interpreter m ((x,v2):: G1) e
            | _ ⇒ None
          end
        | Some (Closr f x e G1) ⇒
          match MML_NS_interpreter m G e2 with
            | Some v2 ⇒
              MML_NS_interpreter m (( x, v2)::(  f,( Closr f x e G1)):: G1) e
            |_ ⇒ None
          end
        | _ ⇒ None
      end
  end
end.
```

Podemos notar que esta función es en realidad un intérprete. Para garantizar que siempre terminará, hemos añadido el parámetro natural `depthR`, que indica la profundidad de la recursión (`depthR` también es llamado "`fuel`" por algunos autores, ver, por ejemplo [51, 50]). Esto se debe a que el cálculo de construcciones inductivas es fuertemente normalizable [115], lo que significa que, desde la perspectiva de un lenguaje de programación que todos los cálculos deben terminar, por tanto, en Coq todas las funciones deben ser totales y deben terminar.

Desde el punto de vista de la verificación, una proposición lógica sirve como una especificación formal que un programa tiene que cumplir. En este caso, la proposición lógica es la definición `Inductive` en `Prop`, mientras que el programa es el intérprete

en `Set`. Entonces, para verificar que nuestro intérprete es correcto con respecto a la semántica natural de MiniML, debemos demostrar el siguiente lema:

```
Lemma MML_NS_interpreter_soundness:
∀ G e v, MML_NS G e v →
    ∃ n, MML_NS_interpreter n G e = Some v.
```

En sentido contrario, es decir, para verificar que nuestro intérprete es completo con respecto a la semántica natural de MiniML, debemos demostrar este lema:

```
Lemma MML_NS_interpreter_completeness:
∀ n, ∀ G, ∀ e, ∀ v,
MML_NS_interpreter n G e = Some v →
MML_NS G e v.
```

Ahora, escribimos el factorial de 5 en MiniML como programa de ejemplo:

```
Definition example :=
  (App (Mu "fac" "x" (If (Eq (Var "x") (Const 0))
             (Const 1)
             (Times (Var "x")
                    (App (Var "fac")
                         (Minus (Var "x") (Const 1))))))  (Const 5)).
```

luego, podemos evaluar nuestro programa de ejemplo en nuestro intérprete de MiniML, dentro de Coq, como sigue:

```
Compute (MML_NS_interpreter 19 nil example).
```

y se obtiene el resultado esperado:

```
= Some (Num 120) : option MML_val
```

Después, podemos utilizar el mecanismo de extracción como sigue:

```
Extraction MML_NS_interpreter.
```

y así obtener un intérprete verificado, correcto y completo con respecto a la semántica natural de MiniML, escrito en OCaml, listo para ser utilizado en la vida real.

```
let rec mML_NS_interpreter depthR g e =
  match depthR with
  | O → None
  | S m →
    (match e with
     | Const n → Some (Num n)
     | ...
     | Letm (x, e1, e2) →
       (match mML_NS_interpreter m g e1 with
        | Some v1 → mML_NS_interpreter m (Cons ((Pair (x, v1)), g)) e2
        | None → None)
     | ...
     | Lam (x, e0) → Some (Clos (x, e0, g))
     | Mu (f, x, e0) → Some (Closr (f, x, e0, g))
     | App (e1, e2) →
       (match mML_NS_interpreter m g e1 with
        | Some m0 →
          (match m0 with
           | Clos (x, e0, g1) →
             (match mML_NS_interpreter m g e2 with
              | Some v2 → mML_NS_interpreter m (Cons ((Pair (x, v2)), g1)) e0
              | None → None)
           | Closr (f, x, e0, g1) →
             (match mML_NS_interpreter m g e2 with
              | Some v2 →
                mML_NS_interpreter m (Cons ((Pair (x, v2)),
                         (Cons ((Pair (f, (Closr (f, x, e0, g1)))), g1)))) e0
              | None → None)
           | _ → None)
        | None → None))
```

El lector puede preguntarse acerca de la "doble" tarea de mantener ambas definiciones: `Prop` y `Set`. Por una parte, si trabajamos en la parte lógica, en `Prop`, no se puede obtener un compilador verificado que se pueda utilizar en la vida real; mientras que, por otra parte, las definiciones dentro del tipo `Set` nos fuerzan a trabajar con funciones totales y que terminen.

Recordemos que la semántica natural es, en general, inherentemente relacional y no determinista; por tanto, para escribir una definición en semántica natural como una función debemos asegurar que es total y determinista. Aunque para algunos casos particulares esto se cumple, pensamos que si una definición en semántica natural se escribe directamente como una función la esencia de la semántica natural se desvanece.

Adicionalmente, Coq genera automáticamente principios de inducción a partir de definiciones inductivas, no así en el caso de funciones. Esos principios de inducción son de utilidad, ya que se pueden utilizar, por medio de la táctica `induction` al realizar una demostración. En algunos escenarios esto representa una ventaja; específicamente, desde luego, cuando demostración se hace por inducción.

Tomando en cuenta los puntos anteriores, damos definiciones en `Prop` para retener la esencia de la semántica natural y para sacar provecho de los principios de inducción generados por Coq. Además, damos las definiciones correspondientes en `Set`, principalmente para obtener implementaciones verificadas.

3.1.2. MiniML$^{\mathsf{dB}}$

Ahora, presentamos el lenguaje en notación de de Bruijn, MiniML$^{\mathsf{dB}}$, que el autor describe en [117], de forma similar a como lo hicimos con MiniML. La codificación en Coq es análoga, por lo que no la describimos.

La sintaxis de MiniML$^{\mathsf{dB}}$ es:

d	::=	n		Naturales
	|	b		Booleanos
	|	$d \star d$	con $\star \in \{+, -, *, =\}$	Expresiones aritméticas y de comparación
	|	$\underline{i}$		Variables sin nombre (índice de de Bruijn)
	|	`if` d `then` d `else` d		Condicionales
	|	`let` d `in` d		Definiciones locales
	|	$\lambda.d$		Abstracción
	|	$\mu.\lambda.d$		Punto fijo
	|	$d\ d$		Aplicación

Los valores y cerraduras en notación de de Bruijn correspondientes se definen como sigue:

v	::=	n	Números	Ω	::=	$[]$	Entorno vacío
	|	b	Booleanos		|	$v \cdot \Omega$	
	|	$(\lambda.d)[\Omega]$	Cerraduras				
	|	$(\mu.\lambda.d)[\Omega]$	Cerraduras recursivas				

Los entornos sirven como una asociación (implícita) de variables (representadas por índices de de Bruijn) a valores. Tal como lo expresa el predicado $\Omega \vdash \underline{i} \longmapsto v$, el valor de una variable representada por el índice $\underline{i}$ está en la posición i en el entorno (una secuencia de valores).

$$\frac{}{v \cdot \Omega \vdash \underline{0} \longmapsto v} \qquad \frac{\Omega \vdash \underline{i} \longmapsto v}{w \cdot \Omega \vdash \underline{S\ i} \longmapsto v}$$

La semántica natural de MiniML$^{\mathsf{dB}}$, siguiendo la exposición del autor en [117], se define inductivamente mediante el siguiente predicado:

$$\Omega \vdash d \Rightarrow v$$

que se puede leer como sigue: en el entorno Ω, la expresión d se evalúa al valor v.

Ciertamente, en el supuesto que el entorno Ω contiene el valor de las variables libres de la expresión d. La semántica natural de $\mathsf{MiniML}^{\mathsf{dB}}$ se define como sigue:

$$\frac{}{\Omega \vdash n \Rightarrow n} \qquad \frac{}{\Omega \vdash b \Rightarrow b} \qquad \frac{\Omega \vdash d_1 \Rightarrow n_1 \qquad \Omega \vdash d_2 \Rightarrow n_2}{\Omega \vdash d_1 \star d_2 \Rightarrow n_1 \star n_2}$$

$$\frac{\Omega \vdash \underline{i} \longmapsto v}{\Omega \vdash \underline{i} \Rightarrow v} \qquad \frac{\Omega \vdash d_1 \Rightarrow v_1 \qquad v_1 \cdot \Omega \vdash d_2 \Rightarrow v}{\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \Rightarrow v}$$

$$\frac{\Omega \vdash d_1 \Rightarrow true \qquad \Omega \vdash d_2 \Rightarrow v}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Rightarrow v} \qquad \frac{\Omega \vdash d_1 \Rightarrow false \qquad \Omega \vdash d_3 \Rightarrow v}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Rightarrow v}$$

$$\frac{}{\Omega \vdash \lambda.d \Rightarrow (\lambda.d)[\Omega]} \qquad \frac{}{\Omega \vdash \mu.\lambda.d \Rightarrow (\mu.\lambda.d)[\Omega]}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \Rightarrow v_2 \qquad v_2 \cdot \Omega_1 \vdash d \Rightarrow v}{\Omega \vdash d_1\ d_2 \Rightarrow v}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \Rightarrow v_2 \qquad v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \Rightarrow v}{\Omega \vdash d_1\ d_2 \Rightarrow v}$$

3.1.3. Traducción

En esta parte, tratamos la traducción de MiniML a $\mathsf{MiniML}^{\mathsf{dB}}$. Una vez más, la codificación en Coq es muy similar, por lo que sólo mencionaremos los puntos en los que esta difiera.

Primero, debemos definir los contextos empleados en esta traducción. Un contexto Π de la traducción a índices de de Bruijn es una secuencia de nombres de variables, es decir, $\Pi = [x_0, \cdots, x_n]$.

Π	::=	$[]$	Contexto vacío
	|	$x \cdot \Pi$	

En semántica natural, esta traducción, dada por el autor en [117], se define mediante el siguiente predicado:

$$\Pi \vdash e \ \Downarrow \ d$$

el cual expresa que: en el contexto Π, la expresión con nombres e de MiniML se traduce a la expresión en notación de de Bruijn d de $\mathsf{MiniML}^{\mathsf{dB}}$.

En el supuesto de que el contexto Π contiene los nombres de las variables libres de e, la traducción se define como sigue:

$$\frac{}{\Pi \vdash n \;\Downarrow\; n} \qquad \frac{}{\Pi \vdash b \;\Downarrow\; b} \qquad \frac{\Pi \vdash e_1 \;\Downarrow\; d_1 \qquad \Pi \vdash e_2 \;\Downarrow\; d_2}{\Pi \vdash e_1 \star e_2 \;\Downarrow\; d_1 \star d_2}$$

$$\frac{}{x \cdot \Pi \vdash x \;\Downarrow\; \underline{0}} \qquad \frac{\Pi \vdash x \;\Downarrow\; \underline{i} \qquad x \neq y}{y \cdot \Pi \vdash x \;\Downarrow\; \underline{S\ i}}$$

$$\frac{\Pi \vdash e_1 \;\Downarrow\; d_1 \qquad x \cdot \Pi \vdash e_2 \;\Downarrow\; d_2}{\Pi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \;\Downarrow\; \texttt{let}\ d_1\ \texttt{in}\ d_2}$$

$$\frac{\Pi \vdash e_1 \;\Downarrow\; d_1 \qquad \Pi \vdash e_2 \;\Downarrow\; d_2 \qquad \Pi \vdash e_3 \;\Downarrow\; d_3}{\Pi \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \;\Downarrow\; \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3}$$

$$\frac{x \cdot \Pi \vdash e \;\Downarrow\; d}{\Pi \vdash \lambda x.e \;\Downarrow\; \lambda.d} \qquad \frac{x \cdot f \cdot \Pi \vdash e \;\Downarrow\; d}{\Pi \vdash \mu f.\lambda x.e \;\Downarrow\; \mu.\lambda.d}$$

$$\frac{\Pi \vdash e_1 \;\Downarrow\; d_1 \qquad \Pi \vdash e_2 \;\Downarrow\; d_2}{\Pi \vdash e_1\ e_2 \;\Downarrow\; d_1\ d_2}$$

En cuanto a la codificación en Coq de esta traducción, es análoga a la de la semántica de MiniML, lo cual significa que se hace con una definición `Inductive` con tipo `Prop`:

```
Inductive dB_translation_NS: ctx → MML_exp → MML_dB_exp → Prop :=
| dB_translation_NS_Const: ∀ P n,
    dB_translation_NS P (Const n) (Const_dB n)

| ...

| dB_translation_NS_Let: ∀ P e1 e2 d1 d2 x,
    dB_translation_NS P e1 d1 →
    dB_translation_NS (x:: P) e2 d2 →
    dB_translation_NS P (Letm x e1 e2) (Letm_dB d1 d2)

| ...

| dB_translation_NS_Lam: ∀ x e d P,
    dB_translation_NS (x:: P) e d →
    dB_translation_NS P (Lam x e) (Lam_dB d)

| dB_translation_NS_Mu: ∀ x f P e d,
    dB_translation_NS (x:: f:: P) e d →
    dB_translation_NS P (Mu f x e) (Mu_dB d)

| dB_translation_NS_App: ∀ P e1 e2 d1 d2,
    dB_translation_NS P e1 d1 →
    dB_translation_NS P e2 d2 →
    dB_translation_NS P (App e1 e2) (App_dB d1 d2).
```

y la función correspondiente se escribe como sigue:

```
Fixpoint dB_translation_NS_compiler (P:ctx) (e: MML_exp)
: option MML_dB_exp :=
match e with
| Const n ⇒ Some (Const_dB n)
| ...
| Letm x e1 e2 ⇒
  match dB_translation_NS_compiler P e1 with
  | Some d1 ⇒
    match dB_translation_NS_compiler (x:: P) e2 with
    | Some d2 ⇒ Some (Letm_dB d1 d2)
    | _ ⇒ None
    end
  | _ ⇒ None
  end
| ...
| Lam x e ⇒
  match dB_translation_NS_compiler (x:: P) e with
  | Some d ⇒ Some (Lam_dB d)
  | _ ⇒ None
  end
| Mu f x e ⇒
  match dB_translation_NS_compiler (x:: f:: P) e with
  | Some d ⇒ Some (Mu_dB d)
  | _ ⇒ None
  end
| App e1 e2 ⇒
  match dB_translation_NS_compiler P e1 with
  | Some d1 ⇒
    match dB_translation_NS_compiler P e2 with
    | Some d2 ⇒ Some (App_dB d1 d2)
    | _ ⇒ None
    end
  | _ ⇒ None
  end
end.
```

Nótese que aquí la función, en lugar de tratarse de un intérprete, es un compilador, pues traduce una expresión en lugar de evaluarla. Obsérvese, además, que esta vez no es necesario agregar un parámetro natural para acotar la recursión, debido a que la traducción es decidible. Esta propiedad se asegura en Coq utilizando recursión estructural (basada en sintaxis) sobre la expresión `e`.

El siguiente lema expresa que el compilador es correcto con respecto a la definición en semántica natural de la compilación:

```
Lemma dB_translation_NS_compiler_soundness:
∀ P e d,
dB_translation_NS P e d →
dB_translation_NS_compiler P e = Some d.
```

En sentido contrario, el siguiente lema expresa que el compilador es completo con respecto a la definición en semántica natural de la compilación:

```
Lemma dB_translation_NS_compiler_completeness:
∀ e, ∀ P, ∀ d,
dB_translation_NS_compiler P e = Some d →
dB_translation_NS P e d.
```

3.1.4. Corrección

La corrección de la traducción se puede establecer como preservación semántica. Primero, se necesita extender la traducción a valores y entornos, lo que se hace de la siguiente manera:

$$\frac{}{n \Downarrow n} \qquad \frac{}{b \Downarrow b}$$

$$\frac{\Gamma \Downarrow \Omega \qquad x \cdot \prod_1(\Gamma) \vdash e \ \Downarrow \ d}{(\lambda x.e)[\Gamma] \Downarrow (\lambda.d)[\Omega]} \qquad \frac{\Gamma \Downarrow \Omega \qquad x \cdot f \cdot \prod_1(\Gamma) \vdash e \ \Downarrow \ d}{(\mu f.\lambda x.e)[\Gamma] \Downarrow (\mu.\lambda.d)[\Omega]}$$

$$\frac{}{[] \Downarrow []} \qquad \frac{v \Downarrow v_d \qquad \Gamma \Downarrow \Omega}{(x, v) \cdot \Gamma \Downarrow v_d \cdot \Omega}$$

Se puede observar que "$\prod_1(\Gamma)$" denota la proyección canónica sobre el primer elemento de cada par del entorno Γ, es decir, si $\Gamma = [(x_0, v_0), \ldots, (x_n, v_n)]$ entonces $\prod_1(\Gamma) = [x_0, \ldots, x_n]$.

La idea general para establecer la corrección es: si una expresión e se evalúa a un valor v en un entorno Γ, y si d y Ω corresponden a la traducción de e y Γ, respectivamente, entonces existe un valor en notación de de Bruijn v_d que corresponde a la traducción de v y se espera que la expresión d se evalúe a v_d en el entorno Ω. De esta forma, la semántica se preserva bajo la traducción. La corrección se enuncia formalmente, como lo hace el autor en [117], mediante el siguiente teorema:

Teorema 1 (Corrección en el caso de terminación)**.** *Sea Γ un entorno, Ω un entorno sin nombres, e una expresión, d una expresión sin nombres y v un valor. Si*

$$\Gamma \vdash e \Rightarrow v, \quad \Gamma \Downarrow \Omega, \quad \prod_1(\Gamma) \vdash e \ \Downarrow \ d$$

entonces, existe un valor sin nombres v_d tal que $v \Downarrow v_d$ y

$$\Omega \vdash d \Rightarrow v_d$$

Demostración. Procedemos por inducción sobre la derivación de $\Gamma \vdash e \Rightarrow v$. En general, debido a que MiniML$^{\mathsf{dB}}$ tiene una sintaxis abstracta y una semántica análogas a aquellas de MiniML, la demostración es muy simple. La idea clave consiste en mostrar que en cada uno de los casos (cada una de las construcciones) la semántica de MiniML$^{\mathsf{dB}}$ sigue aquella de MiniML.

En los casos base cuando e es un natural, un booleano, una variable, una abstracción o un punto fijo (una abstracción recursiva), la demostración es directamente por definición de la semántica de MiniML$^{\mathsf{dB}}$.

En los casos inductivos cuando e es una expresión aritmética o de comparación, una definición local, un condicional o una aplicación, se debe usar la hipótesis de inducción sobre cada una de las subexpresiones correspondientes y concluir por definición de la semántica natural de MiniML$^{\mathsf{dB}}$. □

Este teorema se escribe en Coq de la siguiente forma:

```
Theorem dB_translation_NS_correctness:
∀ e v G, MML_NS G e v →
    ∀ O, dB_translation_NS_env G O →
    ∀ d, dB_translation_NS (Pi_1 G) e d →
  ∃ vd, dB_translation_NS_val v vd ∧
        MML_dB_NS O d vd.
```

3.2. Generación de código

Esta sección está dedicada a la generación de código. Primero trataremos la de la máquina MSECD original de paso corto y, después, la de nuestra máquina MSECD de paso largo.

3.2.1. Máquina MSECD

Leroy [63, 66] introduce la máquina SECD moderna, una máquina basada en la SECD de Landin [60] con dos diferencias principales: la primera es que no utiliza Dump, sino que emplea marcos en la pila para soportar llamadas a función; la segunda es que utiliza índices de de Bruijn para acceder al entorno.

La SECD moderna original únicamente ofrece soporte para constantes naturales, definiciones locales, abstracción y aplicación. Es por ello que aquí ofrecemos una versión de la MSECD extendida para soportar booleanos, operadores aritméticos y de comparación, condicionales y recursión nativa por medio de cerraduras recursivas. Esta extensión está basada en la ofrecida por el autor en [117]. Cabe mencionar que el soporte de condicionales lo hicimos basados en la presentación de la SECD de Henderson [48].

Las instrucciones de la MSECD extendida son las siguientes:

i	::=	IConst n	Apila el natural n
	|	IConstb b	Apila el Booleano b
	|	IAdd	Realiza una adición
	|	ISub	Realiza una substracción
	|	IMul	Realiza una multiplicación
	|	IEq	Realiza una comparación de igualdad
	|	IAcc $\underline{i}$	Apila el valor de la variable número (índice de de Bruijn) $\underline{i}$
	|	ISel c c	Selecciona una rama de un condicional
	|	IJoin	Retoma el control principal (regresa de un condicional)
	|	ILet	Agrega el valor de una variable de una definición local al entorno
	|	IEndLet	Elimina el valor de una variable de una definición local del entorno
	|	IClos c	Apila una cerradura con código c
	|	IClos$_{rec}$ c	Apila una cerradura recursiva con código c
	|	IApp	Realiza una aplicación de función
	|	IRet	Regresa de una función

Nótese la distinción entre "i" e "$\underline{i}$": lo primero denota una instrucción de máquina, mientras que lo segundo denota un índice de de Bruijn.

El código se define como una secuencia de instrucciones:

c	::=	$[]$	Código vacío
	|	$i \cdot c$	

Los valores de la máquina y su entorno se definen de la siguiente manera:

v_m	::=	n	Naturales	Δ	::=	$[]$	Entorno vacío
	|	b	Booleanos		|	$v_m \cdot \Delta$	
	|	$c[\Delta]$	Cerraduras				
	|	$c[\Delta]_{rec}$	Cerraduras recursivas				

Además de estos valores, los marcos también deben poder almacenarse en la pila. Por ello, los valores de pila y la pila de la máquina se definen de la siguiente manera:

v_s	::=	v_m	Valores de máquina	s	::=	$[]$	Pila vacía
	|	(c, Δ)	Marcos de pila		|	$v_s \cdot \Delta$	

Una *configuración* de la máquina MSECD es una tripleta (c, Δ, s) donde c es un código, Δ un entorno de máquina y s una pila cualquiera.

La semántica de paso corto de la MSECD es una relación de transición que va de una configuración (c_i, Δ_i, s_i) a la siguiente $(c_{i+1}, \Delta_{i+1}, s_{i+1})$ denotada como $(c_i, \Delta_i, s_i) \rightarrow (c_{i+1}, \Delta_{i+1}, s_{i+1})$.

Se muestra a continuación, en la Tabla 3.1, la semántica de paso corto de la MSECD que el autor presenta en [117].

Tabla 3.1: Semántica de paso corto de la MSECD.

Actual			Siguiente		
Código	Entorno	Pila	Código	Entorno	Pila
$(\mathsf{IConst}\ n)\cdot c$	Δ	s	c	Δ	$n\cdot s$
$(\mathsf{IConstb}\ b)\cdot c$	Δ	s	c	Δ	$b\cdot s$
$\mathsf{IAdd}\cdot c$	Δ	$n_2\cdot n_1\cdot s$	c	Δ	$n_1+n_2\cdot s$
$\mathsf{ISub}\cdot c$	Δ	$n_2\cdot n_1\cdot s$	c	Δ	$n_1-n_2\cdot s$
$\mathsf{IMul}\cdot c$	Δ	$n_2\cdot n_1\cdot s$	c	Δ	$n_1*n_2\cdot s$
$\mathsf{IEq}\cdot c$	Δ	$n_2\cdot n_1\cdot s$	c	Δ	$n_1=n_2\cdot s$
$(\mathsf{IAcc}\ \underline{i})\cdot c$	$[v_0,\ldots,v_i,\ldots,v_n]=\Delta$	s	c	Δ	$v_i\cdot s$
$\mathsf{ILet}\cdot c$	Δ	$v\cdot s$	c	$v\cdot\Delta$	s
$\mathsf{IEndLet}\cdot c$	$v\cdot\Delta$	s	c	Δ	s
$(\mathsf{ISel}\ c_1\ c_2)\cdot c$	Δ	$true\cdot s$	c_1	Δ	$(c,[])\cdot s$
$(\mathsf{ISel}\ c_1\ c_2)\cdot c$	Δ	$false\cdot s$	c_2	Δ	$(c,[])\cdot s$
$\mathsf{IJoin}\cdot c$	Δ	$v\cdot(c_b,[])\cdot s$	c_b	Δ	$v\cdot s$
$(\mathsf{IClos}\ c_1)\cdot c$	Δ	s	c	Δ	$c_1[\Delta]\cdot s$
$(\mathsf{IClos}_{rec}\ c_1)\cdot c$	Δ	s	c	Δ	$c_1[\Delta]_{rec}\cdot s$
$\mathsf{IApp}\cdot c$	Δ	$v\cdot c_1[\Delta_1]\cdot s$	c_1	$v\cdot\Delta_1$	$(c,\Delta)\cdot s$
$\mathsf{IApp}\cdot c$	Δ	$v\cdot c_1[\Delta_1]_{rec}\cdot s$	c_1	$v\cdot c_1[\Delta_1]_{rec}\cdot\Delta_1$	$(c,\Delta)\cdot s$
$\mathsf{IRet}\cdot c$	Δ	$v\cdot(c_1,\Delta_1)\cdot s$	c_1	Δ_1	$v\cdot s$

La semántica de paso corto de la MSECD se codifica en Coq de la forma siguiente:

```
Inductive MSECD_SS: conf → conf → Prop :=
| MSECD_SS_IConst: ∀ n: nat, ∀ c: code, ∀ D: env, ∀ s: stack,
    MSECD_SS (IConst n:: c, D, s) (c, D, Val (MInt n):: s)

| ...

| MSECD_SS_ILet: ∀ c: code, ∀ D: env, ∀ v: val, ∀ s: stack,
    MSECD_SS (ILet:: c, D, Val v:: s) (c, v:: D, s)

| MSECD_SS_IEndLet: ∀ c: code, ∀ D: env, ∀ v: val, ∀ s: stack,
    MSECD_SS (IEndLet:: c, v:: D, s) (c, D, s)

| ...

| MSECD_SS_IClos: ∀ cc c: code, ∀ D: env, ∀ s: stack,
    MSECD_SS (IClos cc:: c, D, s) (c, D, Val (MClos cc D):: s)

| MSECD_SS_IClosr: ∀ cc c: code, ∀ D: env, ∀ s: stack,
    MSECD_SS (IClosr cc:: c, D, s) (c, D, Val (MClosr cc D):: s)

| MSECD_SS_IApp: ∀ c c1: code, ∀ D D1: env, ∀ s: stack, ∀ v: val,
    MSECD_SS (IApp:: c, D, Val v:: Val (MClos c1 D1):: s)
             (c1, v:: D1, SFrame c D:: s)
```

```
| MSECD_SS_IAppr:∀ c c1:code, ∀ D D1:env, ∀ s:stack, ∀ v:val,
    MSECD_SS (IApp:: c, D, Val v:: Val (MClosr c1 D1):: s)
             (c1, v::( MClosr c1 D1):: D1, SFrame c D:: s)

| MSECD_SS_IReturn: ∀ c c1:code, ∀ D D1:env, ∀ v:val, ∀ s:stack,
    MSECD_SS (IRet:: c, D, Val v:: SFrame c1 D1 :: s) (c1, D1, Val v:: s).
```

Sean m_1, m_2 y m_3 configuraciones de la MSECD, el cierre transitivo de la semántica de paso pequeño se define inductivamente como sigue:

$$\frac{m_1 \to m_2}{m_1 \stackrel{+}{\to} m_2} \qquad\qquad \frac{m_1 \to m_2 \qquad m_2 \stackrel{+}{\to} m_3}{m_1 \stackrel{+}{\to} m_3}$$

En Coq, este cierre transitivo se escribe como se muestra a continuación:

```
Inductive TC_MSECD_SS: conf → conf → Prop :=
| TC_MSECD_SS_SS:
    ∀ m1 m2, MSECD_SS m1 m2 →
             TC_MSECD_SS m1 m2

| TC_MSECD_SS_Transitivity:
    ∀ m1 m2, MSECD_SS m1 m2 →
      ∀ m3, TC_MSECD_SS m2 m3 →
             TC_MSECD_SS m1 m3.
```

3.2.1.1. Compilación

Leroy [63] define la compilación del cálculo lambda puro, en notación de de Bruijn, extendido con constantes al código de la máquina MSECD como una función. Aquí, extendemos su trabajo a todas las construcciones de MiniML$^{\mathsf{dB}}$:

$$\begin{aligned}
[\![n]\!] &= \mathsf{IConst}\ n \\
[\![b]\!] &= \mathsf{IConstb}\ b \\
[\![d_1 \star d_2]\!] &= [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp} \\
[\![\underline{i}]\!] &= \mathsf{IAcc}\ \underline{i} \\
[\![\texttt{let}\ d_1\ \texttt{in}\ d_2]\!] &= [\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \\
[\![\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3]\!] &= [\![d_1]\!] \cdot \mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \\
[\![\lambda.d]\!] &= \mathsf{IClos}\ ([\![d]\!] \cdot \mathsf{IRet}) \\
[\![\mu.\lambda.d]\!] &= \mathsf{IClos}_{rec}\ ([\![d]\!] \cdot \mathsf{IRet}) \\
[\![d_1\ d_2]\!] &= [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp}
\end{aligned}$$

3.2.1.2. Corrección

La corrección de esta compilación se establece como preservación semántica: para esto, primero es necesario extender la compilación a valores y entornos de la máquina, tal como se muestra a continuación:

$$[\![n]\!] = n \qquad [\![b]\!] = b \qquad [\![(\lambda.d)[\Omega]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega]\!]]$$
$$[\![(\mu.\lambda.d)[\Omega]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega]\!]]_{rec} \qquad [\![v_1 \dots v_n]\!] = [\![v_1]\!] \dots [\![v_n]\!]$$

De esta manera, si una expresión d se evalúa a un valor v en un entorno Ω, se espera que su compilación $[\![d]\!]$ se evalúe a $[\![v]\!]$ en el entorno $[\![\Omega]\!]$. Sin embargo, para poder demostrar este resultado es necesario primero fortalecer la hipótesis (veremos, en la Sección 3.2.2.2, que esto no es necesario cuando se utiliza semántica natural). Aquí, fortalecer la hipótesis significa poder concatenar cualquier código c al final de la compilación $[\![d]\!]$, de tal forma que cuando la evaluación de $[\![d]\!]$ termine se espera que $[\![v]\!]$ esté en la cima de la pila y que el código c quede por evaluarse. Esto se expresa formalmente en el siguiente teorema formulado por Leroy [63]:

Teorema 2. *Si* $\Omega \vdash d \Rightarrow v$*, entonces* $([\![d]\!] \cdot c, [\![\Omega]\!], s) \overset{+}{\to} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ *para todo código* c *y pila* s*.*

Demostración. La demostración se realiza por inducción sobre la derivación de $\Omega \vdash d \Rightarrow v$. Los casos base donde d es un natural, un booleano, una variable sin nombres, una abstracción o un punto fijo (abstracción recursiva) son simples, ya que la compilación $[\![d]\!]$ es una única instrucción de máquina cuya evaluación se realiza en un único paso de la máquina $\to$, este paso sigue la evaluación (la semántica de $\mathsf{MiniML}^{\mathsf{dB}}$) de d.

Para los casos inductivos donde d es una expresión aritmética o de comparación, un condicional, una definición local o una aplicación, la demostración sigue la estructura de la derivación de $\Omega \vdash d \Rightarrow v$; la idea clave es utilizar la transitividad de $\overset{+}{\to}$ junto con la hipótesis de inducción tanto como se requiera, además de evaluar las instrucciones que aparecen en $[\![d]\!]$ realizando el paso de máquina correspondiente $\to$. □

Este teorema se escribe en Coq como sigue:

```
Theorem compile_eval:
∀ O,  ∀ d,  ∀ v,  MML_dB_NS O d v →
     ∀ c,  ∀ s,  TC_MSECD_SS ((compile d)++ c, compile_env O,s)
                             (c, compile_env O,  (Val (compile_value v):: s)).
```

3.2.2. Máquina MSECD de paso largo

En esta sección introducimos nuestra versión de paso largo de máquina SECD moderna, la cual está fuertemente basada en nuestra versión extendida de la MSECD original de paso corto. No obstante, a diferencia de la segunda y como consecuencia del

alto nivel de abstracción de la semántica natural, no es necesario emplear marcos de pila en lo absoluto y, por tanto, las instrucciones de retorno también son innecesarias (IRet que sirve para regresar de una llamada a función y IJoin que sirve para regresar de un condicional). Dada la explicación anterior, podemos afirmar que el uso de semántica natural impacta directamente en el diseño de la máquina, específicamente en sus componentes.

Las instrucciones de la máquina, al igual que su código, se muestran a continuación:

i	::=	IConst n	Naturales	c	::=	[]	Código vacío
	|	IConstb b	Booleanos		|	$i \cdot c$	
	|	IAdd	Adición				
	|	ISub	Substracción				
	|	IMul	Multiplicación				
	|	IEq	Comparación de igualdad				
	|	IAcc $\underline{i}$	Acceso al valor de una variable (índice de de Bruijn)				
	|	ISel c c	Condicionales				
	|	ILet	Definiciones locales				
	|	IEndLet					
	|	IClos c	Abstracción				
	|	IClos$_{rec}$ c	Abstracción recursiva				
	|	IApp	Aplicación				

Los valores y entornos de la máquina se definen como sigue:

v_m	::=	n	Naturales	Δ	::=	[]	Entorno vacío
	|	b	Booleanos		|	$v_m \cdot \Delta$	
	|	$c[\Delta]$	Cerraduras				
	|	$c[\Delta]_{rec}$	Cerraduras recursivas				

Dado que los marcos desaparecen, no es necesario definir valores de pila. Debido a esto, la pila pasa a ser directamente una secuencia de valores de máquina.

s	::=	[]	Pila vacía
	|	$v_m \cdot s$	

El predicado $\Delta \vdash \underline{i} \longmapsto v$ expresa que v es el valor de la variable representada por el índice de de Bruijn $\underline{i}$ en el entorno de máquina Δ.

$$\frac{}{v \cdot \Delta \vdash \underline{0} \longmapsto v} \qquad \frac{\Delta \vdash \underline{i} \longmapsto v}{w \cdot \Delta \vdash \underline{S\ i} \longmapsto v}$$

Un *estado* es un par (Δ, s) donde Δ es un entorno de máquina y s una pila.

La semántica natural de la máquina se define mediante los siguientes dos predicados mutuamente dependientes:

$$\Delta, s \vdash c \Rightarrow (\Delta_f, s_f) \qquad\qquad \Delta_1, s_1 \vdash i \Rightarrow (\Delta_2, s_2)$$

El primero es para código y se puede leer como sigue: si la máquina está en el estado (Δ, s), dado el código c, evaluar c lleva a la máquina al estado (Δ_f, s_f). El segundo, para instrucciones, se puede leer como sigue: si la máquina está en un estado (Δ_1, s_1), dada la instrucción i, evaluar i lleva a la máquina al estado (Δ_2, s_2). No obstante, el punto de entrada para la semántica debe ser el predicado para código. Naturalmente, en el supuesto que, en el primero el entorno Δ contiene el valor de las variables (representadas por instrucciones $\mathsf{IAcc}\ \underline{i}$) en c; mientras que, en el segundo el entorno Δ_1 contiene el valor de la variable en i (si es que i se trata de una instrucción $\mathsf{IAcc}\ \underline{i}$).

La semántica natural de la máquina es la siguiente:

$$\frac{}{\Delta, s \vdash [\,] \Rightarrow (\Delta, s)} \qquad \frac{\Delta, s \vdash i \Rightarrow (\Delta_1, s_1) \qquad \Delta_1, s_1 \vdash c \Rightarrow (\Delta_2, s_2)}{\Delta, s \vdash i \cdot c \Rightarrow (\Delta_2, s_2)}$$

$$\frac{}{\Delta, s \vdash \mathsf{IConst}\ n \Rightarrow (\Delta, n \cdot s)} \qquad \frac{}{\Delta, s \vdash \mathsf{IConstb}\ b \Rightarrow (\Delta, b \cdot s)}$$

$$\frac{}{\Delta, n_2 \cdot n_1 \cdot s \vdash \mathsf{IOp}\ \Rightarrow (\Delta, n_1 \star n_2 \cdot s)}\ \mathsf{IOp} \in \{\mathsf{IAdd}, \mathsf{ISub}, \mathsf{IMul}, \mathsf{IEq}\},\ \star \in \{+, -, *, =\}\ \text{resp.}$$

$$\frac{\Delta \vdash \underline{i} \longmapsto v}{\Delta, s \vdash \mathsf{IAcc}\ \underline{i} \Rightarrow (\Delta, v \cdot s)}$$

$$\frac{\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)}{\Delta, \mathit{true} \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Rightarrow (\Delta_1, s_1)} \qquad \frac{\Delta, s \vdash c_2 \Rightarrow (\Delta_1, s_1)}{\Delta, \mathit{false} \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Rightarrow (\Delta_1, s_1)}$$

$$\frac{}{\Delta, v \cdot s \vdash \mathsf{ILet}\ \Rightarrow (v \cdot \Delta, s)} \qquad \frac{}{v \cdot \Delta, s \vdash \mathsf{IEndLet}\ \Rightarrow (\Delta, s)}$$

$$\frac{}{\Delta, s \vdash \mathsf{IClos}\ c \Rightarrow (\Delta, c[\Delta] \cdot s)} \qquad \frac{}{\Delta, s \vdash \mathsf{IClos}_{rec}\ c \Rightarrow (\Delta, c[\Delta]_{rec} \cdot s)}$$

$$\frac{v \cdot \Delta_1, s \vdash c \Rightarrow (\Delta_2, v_1 \cdot s_1)}{\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp}\ \Rightarrow (\Delta, v_1 \cdot s_1)} \qquad \frac{v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Rightarrow (\Delta_2, v_1 \cdot s_1)}{\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp}\ \Rightarrow (\Delta, v_1 \cdot s_1)}$$

La codificación de la semántica natural de la máquina es similar a la de la semántica natural de MiniML, sólo que aquí es necesario usar una definición mutuamente dependiente en Coq en correspondencia con los predicados mutuamente dependientes de la semántica natural de la máquina, tal como se muestra a continuación:

```
Inductive BSMSECD_NSI: state → instruction → state → Prop :=
| BSMSECD_NSI_IConst: ∀ s: stack, ∀ D: env, ∀ n: nat,
    BSMSECD_NSI (D, s) (IConst n) (D,( MInt n:: s))
```

```
| ...

| BSMSECD_NSI_ILet: ∀ s, ∀ D, ∀ v,
    BSMSECD_NSI (D, v:: s) ILet (v:: D, s)

| BSMSECD_NSI_IEndLet: ∀ s, ∀ D, ∀ v,
    BSMSECD_NSI (v:: D, s) IEndLet (D, s)

| BSMSECD_NSI_IClos: ∀ s, ∀ D, ∀ c,
    BSMSECD_NSI (D, s) (IClos c) (D, MClos c D:: s)

| BSMSECD_NSI_IClosr: ∀ s, ∀ D, ∀ c,
    BSMSECD_NSI (D, s) (IClosr c) (D, MClosr c D:: s)

| BSMSECD_NSI_IApp: ∀ s s1, ∀ D1 D2, ∀ v v1, ∀ c,
    BSMSECD_NSC (v:: D1, s) c (D2, v1:: s1) →
∀ D, BSMSECD_NSI (D, v:: MClos c D1:: s) IApp (D, v1:: s1)

| BSMSECD_NSI_IAppr:∀ s s1, ∀ D1 D2, ∀ v v1, ∀ c,
    BSMSECD_NSC (v::( MClosr c D1):: D1, s) c (D2, v1:: s1) →
∀ D, BSMSECD_NSI (D, v:: MClosr c D1:: s) IApp (D, v1:: s1)


with BSMSECD_NSC: state → code → state → Prop :=
| BSMSECD_NSC_nil: ∀ st: state,
            BSMSECD_NSC st nil st

| BSMSECD_NSC_Seq: ∀ st st1, ∀ i,
            BSMSECD_NSI st i st1 →
∀ st2, ∀ cs, BSMSECD_NSC st1 cs st2 →
            BSMSECD_NSC st (i:: cs) st2.
```

Un razonamiento análogo se sigue en el respectivo lema de solidez del intérprete con respecto a la semántica natural de la máquina, en consecuencia se tiene el siguiente lema:

```
Lemma BSMSECD_NSI_interpreter_soundness:
∀ st rst, ∀ i, BSMSECD_NSI st i rst →
          ∃ n, BSMSECD_NSI_interpreter n st i = Some rst

with BSMSECD_NSC_interpreter_soundness:
∀ st rst, ∀ c, BSMSECD_NSC st c rst →
          ∃ n, BSMSECD_NSC_interpreter n st c = Some rst.
```

De la misma manera, en el respectivo lema de completitud del intérprete con respecto

a la semántica natural de la máquina, por lo que se tiene el siguiente lema:

```
Lemma BSMSECD_NSI_interpreter_completeness:
∀ n, ∀ st rst, ∀ i,
BSMSECD_NSI_interpreter n st i = Some rst →
BSMSECD_NSI st i rst

with BSMSECD_NSC_interpreter_completeness:
∀ n, ∀ st rst, ∀ c,
BSMSECD_NSC_interpreter n st c = Some rst →
BSMSECD_NSC st c rst.
```

La semántica natural de la máquina cuenta con la propiedad que se enuncia en el lema que se presenta a continuación:

Lema 1. *Sean* Δ, Δ_1, Δ_2 *entornos de máquina;* s, s_1, s_2 *pilas;* c_1, c_2 *códigos de máquina. Si*

$$\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1), \qquad \Delta_1, s_1 \vdash c_2 \Rightarrow (\Delta_2, s_2)$$

entonces

$$\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta_2, s_2)$$

Demostración. Por inducción sobre la derivación de $\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)$. En el caso base el código c_1 es el código vacío, es decir, $c_1 = []$, por lo que se concluye simplemente por hipótesis. En el caso inductivo c_1 es no vacío, es decir, $c_1 \neq []$, este caso se demuestra por hipótesis y por definición de $\Rightarrow$. □

La importancia de este lema radica en que es de utilidad para poder demostrar la corrección de la compilación (ver Sección 3.2.2.2).

3.2.2.1. Compilación

Utilizando semántica natural, la compilación de MiniML$^{\mathsf{dB}}$ a código de la máquina SECD moderna de paso largo se define mediante el siguiente predicado:

$$d \Downarrow c$$

que denota que la expresión d de MiniML$^{\mathsf{dB}}$ se compila al código de máquina c.

$$\frac{}{n \Downarrow \mathsf{IConst}\ n} \qquad \frac{}{b \Downarrow \mathsf{IConstb}\ b} \qquad \frac{d_1 \Downarrow c_1 \qquad d_2 \Downarrow c_2}{d_1 \star d_2 \Downarrow c_1 \cdot c_2 \cdot \mathsf{IOp}}\ \star \in \{+, -, *, =\},\ \mathsf{IOp} \in \{\mathsf{IAdd}, \mathsf{ISub}, \mathsf{IMul}, \mathsf{IEq}\}\ \text{resp.}$$

$$\frac{}{\underline{i} \Downarrow \mathsf{IAcc}\ \underline{i}} \qquad \frac{d_1 \Downarrow c_1 \qquad d_2 \Downarrow c_2}{\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet}} \qquad \frac{d_1 \Downarrow c_1 \qquad d_2 \Downarrow c_2 \qquad d_3 \Downarrow c_3}{\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c_1 \cdot \mathsf{ISel}\ c_2\ c_3}$$

$$\frac{d \Downarrow c}{\lambda.d \Downarrow \mathsf{IClos}\ c} \qquad \frac{d \Downarrow c}{\mu.\lambda.d \Downarrow \mathsf{IClos}_{rec}\ c} \qquad \frac{d_1 \Downarrow c_1 \qquad d_2 \Downarrow c_2}{d_1\ d_2 \Downarrow c_1 \cdot c_2 \cdot \mathsf{IApp}}$$

La codificación de la compilación, entendida aquí como la generación de código, se hace en semántica natural de forma análoga a la de la traducción a índices de de Bruijn (Sección 3.1.3), es decir, con una definición `Inductive` con tipo `Prop`.

```
Inductive Compilation_NS: MML_dB_exp → code → Prop :=
| Compilation_NS_Const:
     ∀ n: nat, Compilation_NS (Const_dB n) (IConst n:: nil)

| ...

| Compilation_NS_Let:
  ∀ d1, ∀ c1, Compilation_NS d1 c1 →
  ∀ d2, ∀ c2, Compilation_NS d2 c2 →
              Compilation_NS (Letm_dB d1 d2) (c1 ++ ILet::c2  ++ IEndLet::nil)

| ...

| Compilation_NS_Lam:
    ∀ d, ∀ c, Compilation_NS d c →
              Compilation_NS (Lam_dB d) (IClos c:: nil)

| Compilation_NS_Mu:
    ∀ d, ∀ c, Compilation_NS d c →
              Compilation_NS (Mu_dB d) (IClosr c:: nil)

| Compilation_NS_App:
  ∀ d1, ∀ c1, Compilation_NS d1 c1 →
  ∀ d2, ∀ c2, Compilation_NS d2 c2 →
              Compilation_NS (App_dB d1 d2) (c1 ++ c2 ++ IApp::nil).
```

y la función, más precisamente el compilador, que le corresponde es el siguiente:

```
Fixpoint Compilation_NS_compiler (d:MML_dB_exp) : code :=
match d with
| Const_dB n ⇒ IConst n::nil
| ...
| Letm_dB d1 d2 ⇒ Compilation_NS_compiler d1 ++
                  ILet:: Compilation_NS_compiler d2 ++ IEndLet:: nil
| ...
| Lam_dB d ⇒ IClos (Compilation_NS_compiler d)::nil
| Mu_dB d ⇒ IClosr (Compilation_NS_compiler d)::nil
| App_dB d1 d2 ⇒ Compilation_NS_compiler d1 ++
                 Compilation_NS_compiler d2 ++ IApp::nil
end.
```

El lema de solidez del compilador con respecto a la definición en semántica natural de la generación de código es el que sigue:

```
Lemma Compilation_NS_compiler_soundness:
∀ d: MML_dB_exp, ∀ c: code,
Compilation_NS d c →
Compilation_NS_compiler d = c.
```

mientras que el lema de completitud del compilador con respecto a la definición en semántica natural de la generación de código se muestra a continuación:

```
Lemma Compilation_NS_compiler_completeness:
∀ d: MML_dB_exp, ∀ c: code,
Compilation_NS_compiler d = c →
Compilation_NS d c.
```

3.2.2.2. Corrección

Una vez más, extendemos la compilación a valores y entornos para poder formular la corrección, establecida en este trabajo como la preservación semántica.

$$\frac{}{n \Downarrow n} \qquad \frac{}{b \Downarrow b} \qquad \frac{d \Downarrow c \qquad \Omega \Downarrow \Delta}{(\lambda.d)[\Omega] \Downarrow c[\Delta]} \qquad \frac{d \Downarrow c \qquad \Omega \Downarrow \Delta}{(\mu.\lambda.d)[\Omega] \Downarrow c[\Delta]_{rec}}$$

$$\frac{}{[] \Downarrow []} \qquad \frac{v \Downarrow v_m \qquad \Omega \Downarrow \Delta}{v \cdot \Omega \Downarrow v_m \cdot \Delta}$$

En la formulación de la corrección, se espera que si una expresión d se evalúa a un valor v en un entorno Ω, si c es el código que resulta de compilar d y Δ es lo que resulta de compilar Ω, entonces debe existir un valor de máquina v_m que corresponda a la compilación de v, y cuando c se evalúe comenzando con la máquina en un estado (Δ, s), para cualquier pila s, la evaluación debe llevar a la máquina al estado $(\Delta, v_m \cdot s)$. El teorema de corrección se enuncia formalmente como sigue:

Teorema 3 (Corrección en el caso de terminación)**.** *Sea Ω un entorno sin nombres, Δ un entorno de máquina, d una expresión sin nombres, c un código de máquina, v un valor sin nombres. Si*

$$\Omega \vdash d \Rightarrow v, \quad d \Downarrow c, \quad \Omega \Downarrow \Delta$$

entonces existe un valor de máquina v_m tal que $v \Downarrow v_m$ y para toda pila s,

$$\Delta, s \vdash c \Rightarrow (\Delta, v_m \cdot s)$$

Demostración. Procedemos por inducción sobre la derivación de $\Omega \vdash d \Rightarrow v$. Los casos base donde d es un natural, un booleano, una variable sin nombres, una abstracción, o un punto fijo (abstracción recursiva) son simples. En estos casos, exhibimos un v_m tal que $v \Downarrow v_m$, debido a que c es el resultado de la compilación de d, c es una única

instrucción de máquina por lo que $\Delta, s \vdash c \Rightarrow (\Delta, v_m \cdot s)$ se cumple simplemente por definición.

En los casos inductivos donde d es una expresión aritmética o de comparación, un condicional, una definición local o una aplicación, la idea principal es utilizar la hipótesis de inducción junto con el Lema 1. De esta manera, la evaluación de la máquina sigue la estructura de la derivación $\Omega \vdash d \Rightarrow v$ y la demostración es simple e intuitiva. □

Este teorema se escribe en Coq como sigue:

```
Theorem Compilation_NS_correctness:
∀ O,  ∀ d,  ∀ v,  MML_dB_NS O d v →
           ∀ c,  Compilation_NS d c →
           ∀ D,  Compilation_NS_env O D →
          ∃ mv,  Compilation_NS_val v mv  ∧
           ∀ s,  BSMSECD_NSC (D, s)  c   (D,  mv:: s).
```

Podemos notar inmediatamente que, debido a que la semántica natural se utiliza como formalismo unificador para definir cada uno de los componentes del compilador: lenguaje fuente, compilación y la máquina, el lenguaje fuente se traduce al lenguaje objetivo de manera transparente a través de la compilación. De esta forma, establecer la corrección es más fácil, simple e intuitivo que una solución ad-hoc. Por ejemplo, aquí no fue necesario definir previamente el cierre transitivo de una relación, y tampoco se precisó de fortalecer la hipótesis para poder demostrar el teorema de corrección, en comparación, eso sí fue necesario al utilizar una función para definir la compilación y al emplear semántica de paso pequeño en la máquina.

Capítulo 4

Semántica natural coinductiva

En este capítulo, afrontaremos el caso en que los cómputos no terminan, para lo cual usaremos semántica natural coinductiva. Trataremos primero la traducción a índices de de Bruijn y posteriormente la generación de código.

4.1. Traducción a índices de de Bruijn

Esta sección introduce los cómputos que no terminan en la traducción a índices de de Bruijn.

4.1.1. MiniML

En general, la coinducción nos permite razonar sobre estructuras infinitas. De esta forma, tomando en cuenta el diseño de la semántica natural, podemos emplear una definición coinductiva para expresar las evaluaciones infinitas de un lenguaje, en este caso de MiniML. Siguiendo a Leroy [63], definimos la semántica natural coinductiva para divergencia (evaluaciones infinitas) mediante el siguiente predicado:

$$\Gamma \vdash e \stackrel{\infty}{\Rightarrow}$$

que se puede leer: en el entorno Γ, la evaluación de la expresión e es infinita, diverge o no termina.

Así, las evaluaciones infinitas de MiniML quedan definidas mediante la interpretación coinductiva de las siguientes reglas:

$$\frac{\Gamma \vdash e_1 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash e_1 \star e_2 \stackrel{\infty}{\Rightarrow}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow n_1 \qquad \Gamma \vdash e_2 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash e_1 \star e_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Gamma \vdash e_1 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \stackrel{\infty}{\Rightarrow}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad (x, v_1) \cdot \Gamma \vdash e_2 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Gamma \vdash e_1 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow true \qquad \Gamma \vdash e_2 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \stackrel{\infty}{\Rightarrow}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow false \qquad \Gamma \vdash e_3 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Gamma \vdash e_1 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash e_1\ e_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (\lambda x.e)[\Gamma_1] \qquad \Gamma \vdash e_2 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash e_1\ e_2 \stackrel{\infty}{\Rightarrow}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow (\mu f.\lambda x.e)[\Gamma_1] \qquad \Gamma \vdash e_2 \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash e_1\ e_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (\lambda x.e)[\Gamma_1] \qquad \Gamma \vdash e_2 \Rightarrow v_2 \qquad (x, v_2) \cdot \Gamma_1 \vdash e \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash e_1\ e_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (\mu f.\lambda x.e)[\Gamma_1] \qquad \Gamma \vdash e_2 \Rightarrow v_2 \qquad (x, v_2) \cdot (f, (\mu f.\lambda x.e)[\Gamma_1]) \cdot \Gamma_1 \vdash e \stackrel{\infty}{\Rightarrow}}{\Gamma \vdash e_1\ e_2 \stackrel{\infty}{\Rightarrow}}$$

Adoptamos la convención de Leroy [63], donde líneas dobles horizontales denotan interpretación coinductiva, mientras que una sola línea horizontal denota interpretación inductiva.

El soporte de coinducción en Coq está basado en el trabajo de Giménez [43]; en particular, Coq cuenta con soporte nativo de definiciones coinductivas. De forma similar al modo en que una definición en semántica natural se puede codificar en Coq como una definición `Inductive` con tipo `Prop`, una definición en semántica natural coinductiva se puede codificar en Coq como una definición `CoInductive` con tipo `Prop`. De aquí que la semántica de MiniML para divergencia se escriba en Coq como sigue:

```
CoInductive MML_CNS : MML_env → MML_exp → Prop :=
| ...

| MML_CNS_LetL:
      ∀ G, ∀ e1, MML_CNS G e1 →
      ∀ x, ∀ e2, MML_CNS G (Letm x e1 e2)

| MML_CNS_LetR:
∀ G, ∀ e1, ∀ v1, MML_NS G e1 v1 →
      ∀ x, ∀ e2, MML_CNS ((x, v1):: G) e2 →
                 MML_CNS G (Letm x e1 e2)

| ...

| MML_CNS_AppL:
      ∀ G, ∀ e1, MML_CNS G e1 →
           ∀ e2, MML_CNS G (App e1 e2)

| MML_CNS_AppR:
  ∀ G G1, ∀ e1 e, ∀ x, MML_NS G e1 (Clos x e G1) →
```

```
                ∀ e2, MML_CNS G e2 →
                      MML_CNS G (App e1 e2)

| MML_CNS_AppRr:
∀ G G1, ∀ e1 e, ∀ f x, MML_NS G e1 (Closr f x e G1)   →
                ∀ e2, MML_CNS G e2 →
                      MML_CNS G (App e1 e2)

| MML_CNS_AppF:
  ∀ G G1, ∀ e1 e, ∀ x, MML_NS G e1 (Clos x e G1) →
          ∀ e2, ∀ v2, MML_NS G e2 v2 →
                      MML_CNS ((x, v2):: G1) e →
                      MML_CNS G (App e1 e2)

| MML_CNS_AppFr:
∀ G G1, ∀ e1 e, ∀ f x, MML_NS G e1 (Closr f x e G1) →
          ∀ e2, ∀ v2, MML_NS G e2 v2 →
                      MML_CNS ((x, v2)::( f,( Closr f x e G1)):: G1) e →
                      MML_CNS G (App e1 e2).
```

En el mismo sentido que verificamos (en la Sección 3.1.1) que nuestro intérprete `MML_dB_NS_interpreter` es correcto con respecto a la semántica natural de MiniML, aquí es preciso verificar que también lo es con respecto a la semántica natural coinductiva para divergencia de MiniML, es decir, tenemos que demostrar el siguiente lema:

```
Lemma MML_NS_interpreter_soundness_non_termination:
∀ G, ∀ e, MML_CNS G e →
     ∀ n, MML_NS_interpreter n G e = None.
```

Este lema enuncia que si la evaluación de `d` no termina, sea cual sea el valor `n` del combustible, al intérprete siempre eventualmente se le agotará (`None` significa que el intérprete se quedó sin combustible).

En sentido contrario, para verificar que nuestro intérprete es completo con respecto a la semántica natural coinductiva de MiniML debemos demostrar el lema que sigue:

```
Lemma MML_NS_interpreter_completeness_non_termination:
 ∀ n, ∀ G, ∀ e,
 MML_NS_interpreter n G e = None →
 not (∃ v, MML_NS G e v) ∨
 (∃ m, m > n ∧ ∃ v, MML_NS_interpreter m G e = Some v).
```

Este lema enuncia que si el intérprete se quedó sin combustible entonces no existe una evaluación finita de `d`, o bien sí existe, pero se necesita más combustible para que el intérprete sea capaz de calcularla. La demostración de este lema en Coq requiere razonamiento clásico.

4.1.2. MiniML$^{\mathsf{dB}}$

La semántica natural coinductiva para divergencia de MiniML$^{\mathsf{dB}}$ se define mediante el siguiente predicado:

$$\Omega \vdash d \stackrel{\infty}{\Rightarrow}$$

que se lee: en el entorno Ω la evaluación de la expresión d es infinita, diverge o no termina.

Las evaluaciones infinitas de MiniML$^{\mathsf{dB}}$ se definen como la interpretación coinductiva de las siguientes reglas:

$$\frac{\Omega \vdash d_1 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash d_1 \star d_2 \stackrel{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow n_1 \qquad \Omega \vdash d_2 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash d_1 \star d_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \stackrel{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow v_1 \qquad v_1 \cdot \Omega \vdash d_2 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \Rightarrow true \qquad \Omega \vdash d_2 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \stackrel{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow false \qquad \Omega \vdash d_3 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \stackrel{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \Rightarrow v_2 \qquad v_2 \cdot \Omega_1 \vdash d \stackrel{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \Rightarrow v_2 \qquad v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \stackrel{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}}$$

4.1.3. Corrección

La idea al establecer la corrección, preservación semántica, es que si en el entorno Γ la expresión e diverge, necesariamente en el entorno Ω la expresión d diverge, donde Ω corresponde a la traducción del entorno Γ y d corresponde a la traducción de la expresión e a notación de de Bruijn. Esto se enuncia formalmente en el siguiente teorema:

Teorema 4 (Corrección en el caso de no terminación)**.** *Sea Γ un entorno, Ω un entorno sin nombres, e una expresión, d una expresión sin nombres. Si*

$$\Gamma \vdash e \stackrel{\infty}{\Rightarrow}, \quad \Gamma \Downarrow \Omega, \quad \prod_1(\Gamma) \vdash e \ \Downarrow d$$

entonces

$$\Omega \vdash d \overset{\infty}{\Rightarrow}$$

Demostración. Procedemos por coinducción. Debido a que, por un lado, la semántica natural para cómputos finitos de MiniML$^{\mathsf{dB}}$ es análoga a aquella de MiniML y a que, por el otro, la semántica natural coinductiva para cómputos infinitos de MiniML$^{\mathsf{dB}}$ también es análoga a la de MiniML, la demostración es muy simple e intutiva. La idea clave consiste en que la semántica de MiniML$^{\mathsf{dB}}$ siga a la de MiniML. Así, para la evaluación de las subexpresiones finitas de d se debe utilizar el Teorema 1, mientras que para las subexpresiones infinitas se emplea la hipótesis de coinducción. □

Este teorema se escribe en Coq como sigue:

```
Theorem dB_translation_CNS_correctness:
∀ G,  ∀ e,  MML_CNS G e →
      ∀ O,  dB_translation_NS_env G O →
      ∀ d,  dB_translation_NS (Pi_1 G)  e d →
            MML_dB_CNS O d.
```

Podemos notar que establecer y demostrar la corrección es una tarea muy fácil, simple e intutiva, debido, entre otras razones, al uso de la semántica natural coinductiva tanto en MiniML como en MiniML$^{\mathsf{dB}}$ y a que ambos lenguajes son muy cercanos; particularmente, son lenguajes de alto nivel con la misma estructura. De este modo, la semántica se respeta de forma transparente bajo la traducción a la notación de de Bruijn.

4.2. Generación de código

Esta parte está dedicada a la generación de código. Particularmente, en este apartado introducimos las evaluaciones infinitas en nuestra máquina MSECD de paso largo. Previamente, se describen los cómputos infinitos en la máquina MSECD original de paso corto.

4.2.1. Máquina MSECD

Aquí se tratan los cómputos que no terminan en la verificación de la generación de código de la máquina MSECD original de paso corto.

4.2.1.1. Semántica de paso corto de cómputos infinitos

Comencemos analizando cómo se pueden expresar los cómputos que no terminan en una máquina. Leroy [63] utiliza semántica de paso corto para expresar un número infinito de transiciones en la MSECD. Él define la relación de transición " $\overset{\infty}{\Rightarrow}$ " coinductivamente de la siguiente forma:

$$\frac{m_1 \to m_2 \qquad m_2 \overset{\infty}{\Rightarrow}}{m_1 \overset{\infty}{\Rightarrow}}$$

Esta relación se puede escribir en Coq como sigue:

```
CoInductive transinf: conf → Prop :=
| transinf_intro: ∀ m1 m2,
    MSECD_SS m1 m2 →
    transinf m2 →
    transinf m1.
```

No obstante, utilizando directamente esta definición no es posible demostrar la corrección de la compilación para los casos que no terminan. Esto se debe a que el mecanismo de coinducción de Coq impone la condición de guardia (*guard condition*); ésta consiste en exigir que se utilice al menos una regla (un constructor) de la definición coinductiva, antes de que se utilice la hipótesis de coinducción, durante una demostración por coinducción. La solución que ofrece Leroy [63] es definir una relación auxiliar con la que se puede llevar a cabo esta demostración, y que es equivalente a la definición original anterior. La relación auxiliar se define como sigue:

$$\frac{m \overset{\infty}{\underset{n}{\Rightarrow}}}{m \overset{\infty}{\underset{n+1}{\Rightarrow}}} \ (\overset{\infty}{\underset{n}{\Rightarrow}}\text{-sleep}) \qquad \frac{m_1 \overset{+}{\to} m_2 \qquad m_2 \overset{\infty}{\underset{n'}{\Rightarrow}}}{m_1 \overset{\infty}{\underset{n}{\Rightarrow}}} \ (\overset{\infty}{\underset{n}{\Rightarrow}}\text{-perform})$$

la propiedad más importante de la relación $\overset{\infty}{\underset{n}{\Rightarrow}}$, para nuestro fines, es que permite a la máquina permanecer en una misma configuración a lo más un número finito n de veces (regla $\overset{\infty}{\underset{n}{\Rightarrow}}$-sleep). Esta regla es crucial para cumplir con la condición de guardia al demostrar la corrección. En algún momento antes de que n llegue a 0, o necesariamente cuando n llegue a 0, se debe realizar al menos una transición (regla $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform), a cambio de realizar una transición, el valor de n se restablece a un natural cualquiera n', es decir, se da nuevamente la posibilidad de quedarse en una misma configuración (esta vez, a lo más n' veces). La relación $\overset{\infty}{\Rightarrow}$ y la relación $\overset{\infty}{\underset{n}{\Rightarrow}}$ son equivalentes, tal como lo enuncia el siguiente lema:

Lema 2. *Sea m una configuración, n un natural cualquiera,*

$$m \overset{\infty}{\Rightarrow} \quad \textit{si y sólo si} \quad m \overset{\infty}{\underset{n}{\Rightarrow}}$$

Demostración. La implicación correspondiente a la parte del *si* es por coinducción. Por definición de $\overset{\infty}{\Rightarrow}$ necesariamente $m \to m_1$ y $m_1 \overset{\infty}{\Rightarrow}$, el resultado se obtiene aplicando la hipótesis de coinducción sobre $m_1 \overset{\infty}{\Rightarrow}$ y luego usando la regla $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform. La parte del *sólo si* también es por coinducción. Suponiendo que si $m \overset{\infty}{\underset{n}{\Rightarrow}}$ entonces $m \to m_1$ y $m_1 \overset{\infty}{\underset{n_1}{\Rightarrow}}$, aplicamos la hipótesis de coinducción sobre $m_1 \overset{\infty}{\underset{n_1}{\Rightarrow}}$ y luego el resultado se obtiene por definición de $\overset{\infty}{\Rightarrow}$. □

4.2.1.2. Corrección

Para demostrar la corrección es necesario definir una medida que indique cuántas veces puede permanecer la máquina en una misma configuración con base en las construcciones del lenguaje. La medida que ofrece Leroy (extendida a todo el lenguaje $\mathsf{MiniML}^{\mathsf{dB}}$) es la siguiente:

$$
\begin{aligned}
\|n\| = \|b\| = \|x\| = \|\lambda.d\| = \|\mu.\lambda.d\| &= 0 \\
\|d_1 \star d_2\| &= \|d_1\| + 1 \\
\|\texttt{let } d_1 \texttt{ in } d_2\| &= \|d_1\| + 1 \\
\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\| &= \|d_1\| + 1 \\
\|d_1\ d_2\| &= \|d_1\| + 1
\end{aligned}
$$

Esto se debe a que un paso en la evaluación de una expresión d de $\mathsf{MiniML}^{\mathsf{dB}}$ puede no corresponder, uno a uno en el mismo orden, a una transición al evaluar $[\![d]\!]$ en la máquina, lo cual vuelve necesario que la máquina permanezca en la misma configuración, $\|d\|$ veces, antes de realizar una transición.

De esta manera, estamos listos para enunciar la corrección para el caso de no terminación, utilizando la relación auxiliar $\overset{\infty}{\underset{n}{\Rightarrow}}$ y fortaleciendo la hipótesis, mediante el siguiente lema:

Lema 3. *Si* $\Omega \vdash d \overset{\infty}{\Rightarrow}$, *entonces* $([\![d]\!] \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|d\|}{\Rightarrow}}$ *para todo código* c *y pila* s.

Demostración. Por coinducción. La idea principal de la demostración es utilizar el Teorema 2 para evaluar las partes finitas de d, aplicar la hipótesis de coinducción sobre las partes infinitas de d, y emplear las reglas $\overset{\infty}{\underset{n}{\Rightarrow}}$-sleep y $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform según convenga. □

Esto da la posibilidad de formular el teorema de corrección directamente con la relación $m \overset{\infty}{\Rightarrow}$, como lo hace Leroy [63], de la siguiente manera:

Teorema 5. *Si* $\Omega \vdash d \overset{\infty}{\Rightarrow}$, *entonces* $([\![d]\!] \cdot c, [\![\Omega]\!], s) \overset{\infty}{\Rightarrow}$ *para todo código* c *y pila* s.

Demostración. El resultado se deduce inmediatamente a partir del Lemma 3 seguido por una aplicación del Lemma 2. □

Este teorema se escribe en Coq como sigue:

```
Theorem compile_evalinf:
∀ 0,  ∀ d,  MML_dB_CNS 0 d →
∀ c,  ∀ s,  transinf  ((compile d) ++ c, (compile_env 0), s).
```

4.2.2. Máquina MSECD de paso largo

Esta parte está dedicada a la verificación de los cómputos que no terminan en la generación de código de la máquina MSECD de paso largo.

4.2.2.1. Semántica de paso largo de cómputos infinitos

En esta sección, introduciremos la semántica natural coinductiva para expresar cómputos que no terminan (evaluaciones infinitas) en una máquina. Ilustraremos su uso con nuestra máquina SECD moderna de paso largo.

De manera semejante a la semántica natural (para evaluaciones finitas, Sección 3.2.2), la semántica natural coinductiva para divergencia (evaluaciones infinitas) se define mediante los siguientes dos predicados mutuamente dependientes:

$$\Delta, s \vdash i \Rrightarrow \qquad\qquad \Delta, s \vdash c \Rrightarrow$$

el primero se lee: en el estado (Δ, s) la instrucción i diverge, mientras que el segundo se lee: en el estado (Δ, s) el código c diverge.

Por ende, las evaluaciones infinitas de la máquina se definen como la interpretación coinductiva de las siguientes reglas:

$$1)\ \frac{\Delta, s \vdash i \Rrightarrow}{\Delta, s \vdash i \cdot c \Rrightarrow} \qquad 2)\ \frac{\Delta, s \vdash i \Rightarrow (\Delta_1, s_1) \qquad \Delta_1, s_1 \vdash c \Rrightarrow}{\Delta, s \vdash i \cdot c \Rrightarrow}$$

$$3)\ \frac{\Delta, s \vdash c_1 \Rrightarrow}{\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Rrightarrow} \qquad 4)\ \frac{\Delta, s \vdash c_2 \Rrightarrow}{\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Rrightarrow}$$

$$5)\ \frac{v \cdot \Delta_1, s \vdash c \Rrightarrow}{\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Rrightarrow} \qquad 6)\ \frac{v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Rrightarrow}{\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Rrightarrow}$$

La semántica natural coinductiva de la máquina se codifica en Coq de forma similar a la semántica natural coinductiva de $\mathsf{MiniML}^{\mathsf{dB}}$. La diferencia principal radica en que aquí es necesario utilizar una definición coinductiva mutuamente dependiente en Coq en correspondencia con los predicados mutuamente dependientes de la semántica natural coinductiva de la máquina. Por tanto, se escribe en Coq en la forma siguiente:

```
CoInductive BSMSECD_CNSI: state → instruction → Prop :=
| ...

| BSMSECD_CNSI_IApp:
∀ v, ∀ D1, ∀ s, ∀ c, BSMSECD_CNSC (v:: D1, s) c →
                ∀ D, BSMSECD_CNSI (D, v:: MClos c D1:: s) IApp

| BSMSECD_CNSI_IAppr:
∀ v, ∀ c, ∀ D1, ∀ s, BSMSECD_CNSC (v::( MClosr c D1):: D1, s) c →
                ∀ D, BSMSECD_CNSI (D, v:: MClosr c D1:: s) IApp


with BSMSECD_CNSC: state → code → Prop :=
| BSMSECD_CNSC_IL:
```

```
      ∀ D, ∀ s, ∀ i, BSMSECD_CNSI (D, s) i →
                ∀ c, BSMSECD_CNSC (D, s) (i:: c)

| BSMSECD_CNSC_ICR:
∀ D D1, ∀ s s1, ∀ i, BSMSECD_NSI (D, s) i (D1, s1) →
                ∀ c, BSMSECD_CNSC (D1, s1) c →
                     BSMSECD_CNSC (D, s) (i:: c).
```

Por su parte, el respectivo lema de solidez del intérprete de la máquina con respecto a la semántica natural coinductiva de la máquina consiste en realidad de dos lemas mutuamente dependientes que corresponden, una vez más, con los dos predicados mutuamente dependientes de la semántica de la máquina. Estos lemas son los siguientes:

```
Lemma BSMSECD_NSI_interpreter_soundness_non_termination:
∀ st, ∀ i, BSMSECD_CNSI st i →
      ∀ n, BSMSECD_NSI_interpreter n st i = None

with BSMSECD_NSC_interpreter_soundness_non_termination:
∀ st, ∀ c, BSMSECD_CNSC st c →
      ∀ n, BSMSECD_NSC_interpreter n st c = None.
```

Lo mismo pasa con el respectivo lema de completitud del intérprete. En consecuencia, se tienen los siguientes lemas:

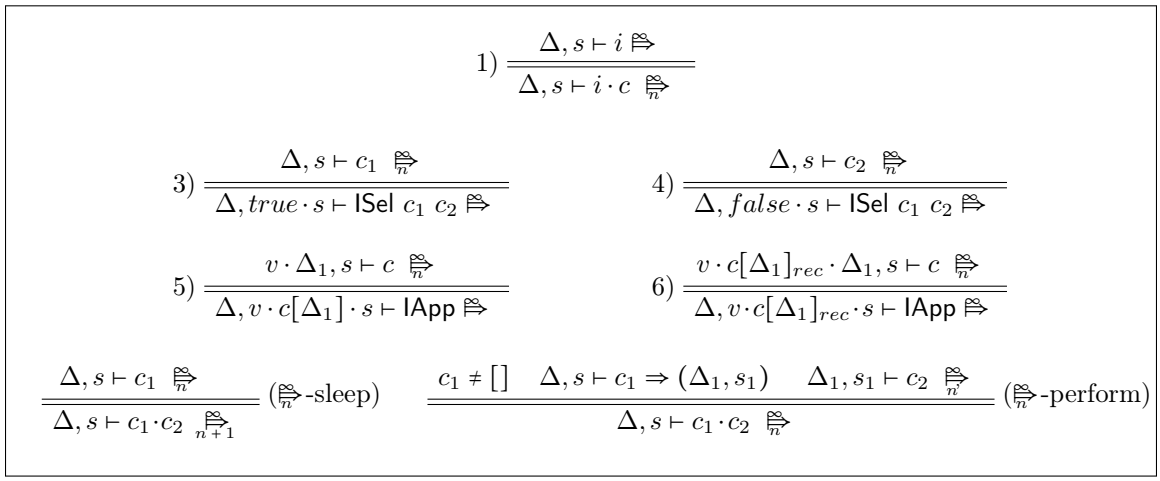
```
Lemma BSMSECD_CNSI_interpreter_completeness:
∀ n, ∀ st, ∀ i,
BSMSECD_NSI_interpreter n st i = None →
not (∃ sti, BSMSECD_NSI st i sti) ∨
(∃ m, m > n ∧ ∃ sti, BSMSECD_NSI_interpreter m st i = Some sti)

with BSMSECD_CNSC_interpreter_completeness:
∀ n, ∀ st, ∀ c,
BSMSECD_NSC_interpreter n st c = None →
not (∃ stf, BSMSECD_NSC st c stf) ∨
(∃ m, m > n ∧ ∃ stf, BSMSECD_NSC_interpreter m st c = Some stf).
```

Dada la definición de la semántica natural coinductiva de la máquina, ofrecemos aquí una breve explicación de sus reglas. Al evaluar un código, se debe comenzar evaluando la primera instrucción i; esta evaluación puede ser finita (regla 2) o infinita (regla 1). En el caso de la regla 1, si la primera instrucción diverge, el código completo diverge. ¿Cómo es que una instrucción puede diverger? Recordemos (Sección 3.2.2) que, debido al alto nivel de abstracción de la MSECD de paso largo, en el caso de terminación las instrucciones ISel y IApp se evalúan por completo en un único paso largo, el cual incluye la evaluación de sus sub-códigos; es por ello que para estas

instrucciones se necesitan reglas para expresar la posibilidad de que sus correspondientes sub-códigos diverjan (reglas 3-6). En el caso de la regla 2, si la evaluación de la primera instrucción termina, pero el código restante diverge, el código completo (incluyendo la primer instrucción) diverge.

Aquí notamos que, en principio, únicamente la regla 2 es necesaria para expresar divergencia en la máquina, ya que, intuitivamente, una instrucción realiza solamente una operación básica, y esta regla es la análoga a la relación de transición correspondiente a la semántica de paso corto $\overset{\infty}{\Rightarrow}$. No obstante, como se mencionó anteriormente, nuestra máquina de paso largo tiene dos instrucciones, concretamente ISel y IApp, que son de alto nivel (y por lo tanto se evalúan de forma distinta a sus contrapartes con semántica de paso corto). Es por tal motivo que estas instrucciones requieren reglas específicas, mientras que las demás instrucciones realizan solamente una operación básica; por ejemplo, IConst n pone n en la cima de la pila. Por esto último, las instrucciones restantes no requieren reglas específicas.

Así pues, quedan definidos por completo los cómputos que no terminan en la máquina. No obstante, al igual que en el caso de la MSECD de paso corto, nos enfrentamos una vez más al problema con la condición de guardia de Coq. Esto significa que, de forma semejante, no podemos demostrar la corrección directamente utilizando esta relación. Para resolver el problema, presentaremos una variante de la solución de Leroy adaptada a la semántica natural coinductiva. Esto quiere decir que debemos definir una relación auxiliar que sea equivalente con la relación original anterior y que nos permita demostrar la corrección. La relación auxiliar es la siguiente:



la regla ($\overset{\infty}{\underset{n}{\Rightarrow}}$-sleep) es la versión análoga mejorada de la regla ($\overset{\infty}{\underset{n}{\Rightarrow}}$-sleep) de Leroy, ya que, adicionalmente, expresa que si el código inicial c_1 diverge, el código restante c_2 divergirá sin importar cuál sea. Esta mejora permite demostrar la corrección sin la necesidad de fortalecer la hipótesis. La regla ($\overset{\infty}{\underset{n}{\Rightarrow}}$-perform) es análoga a la regla ($\overset{\infty}{\underset{n}{\Rightarrow}}$-perform) de Leroy. Podemos notar que la regla 2 desaparece (ya no es necesaria) debido a que pasó a ser un caso particular de la regla ($\overset{\infty}{\underset{n}{\Rightarrow}}$-perform). Por su parte, las reglas restantes (1 y 3-6) permanecen sin cambios, es decir, son análogas, pero en lugar de la relación para código $\overset{\infty}{\Rightarrow}$ usan la relación $\overset{\infty}{\underset{n}{\Rightarrow}}$ con un natural n cualquiera.

Cabe mencionar que la relación $\overset{\infty}{\Rightarrow}$ define los cómputos que no terminan de la máquina de la forma esperada y que sería ideal trabajar directamente con esta relación en Coq; no obstante, debido a la condición de guardia de Coq esto no es posible. Por tal razón, hemos definido la relación $\overset{\infty}{\Mapsto}$ justo para vencer a la condición de guardia de Coq. Debido a que hemos definido la relación $\overset{\infty}{\Mapsto}$ (más precisamente la relación $\overset{\infty}{\underset{n}{\Mapsto}}$ para código y la relación $\overset{\infty}{\Mapsto}$ para instrucciones) de manera similar a la relación $\overset{\infty}{\Rightarrow}$, hemos utilizado también una notación similar; sin embargo, debemos ser cuidadosos y notar la distinción entre " $\overset{\infty}{\Rightarrow}$ " y " $\overset{\infty}{\Mapsto}$ ".

El siguiente lema enuncia que las dos relaciones mutuamente dependientes originales son equivalentes con las dos relaciones mutuamente dependientes auxiliares.

Lema 4. *Sea Δ un entorno de máquina, s una pila, i una instrucción,*

$$\Delta, s \vdash i \overset{\infty}{\Rightarrow} \quad \textit{si y sólo si} \quad \Delta, s \vdash i \overset{\infty}{\Mapsto}$$

y, sea c un código, n un natural cualquiera,

$$\Delta, s \vdash c \overset{\infty}{\Rightarrow} \quad \textit{si y sólo si} \quad \Delta, s \vdash c \overset{\infty}{\underset{n}{\Mapsto}}$$

Demostración. La implicación correspondiente al *si* está compuesta por: si $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$ entonces $\Delta, s \vdash i \overset{\infty}{\Mapsto}$, y si $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$ entonces $\Delta, s \vdash c \overset{\infty}{\underset{n}{\Mapsto}}$, es decir, de los siguientes dos casos:

1. Si $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$ entonces $\Delta, s \vdash i \overset{\infty}{\Mapsto}$. Suponiendo 2, debido a que la definición de las instrucciones en $\overset{\infty}{\Rightarrow}$ y $\overset{\infty}{\Mapsto}$ son análogas, la demostración procede por un simple análisis de casos, cada uno de estos casos se demuestra directamente por definición de $\overset{\infty}{\Mapsto}$ utilizando 2 para obtener las premisas de código $\Delta, s \vdash c \overset{\infty}{\underset{n}{\Mapsto}}$ requeridas.

2. Si $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$ entonces $\Delta, s \vdash c \overset{\infty}{\underset{n}{\Mapsto}}$. Por coinducción. La idea principal es utilizar la hipótesis de coinducción junto con la regla $\overset{\infty}{\underset{n}{\Mapsto}}$-perform, y aplicar 1 para obtener las premisas $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ donde se requieran.

Por su parte, la implicación correspondiente al *sólo si* está compuesta por: si $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ entonces $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$, y si $\Delta, s \vdash c \overset{\infty}{\underset{n}{\Mapsto}}$ entonces $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$. Estos es, por los siguientes dos casos:

3. Si $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ entonces $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$. Suponiendo 4, la demostración es análoga a la de 1 pero en sentido contrario (y utilizando 4 en lugar de 2).

4. Si $\Delta, s \vdash c \overset{\infty}{\underset{n}{\Mapsto}}$ entonces $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$. Suponiendo que, si $\Delta, s \vdash i \cdot c \overset{\infty}{\underset{n}{\Mapsto}}$ entonces $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ o existe n_1, Δ_1, s_1, tal que $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ y $\Delta_1, s_1 \vdash c \overset{\infty}{\underset{n_1}{\Mapsto}}$; la demostración procede por coinducción. La idea clave es utilizar la hipótesis de coinducción junto con la definición de $\overset{\infty}{\underset{n}{\Mapsto}}$, en particular con la regla $\overset{\infty}{\underset{n}{\Mapsto}}$-perform, empleando la proposición supuesta en el paso que se requiere, y aplicando 3 para obtener las premisas $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$ donde sea necesario. □

4.2.2.2. Corrección

Para demostrar la corrección, debemos utilizar la relación auxiliar $\overset{\cong}{\Rightarrow}_n$ junto con una medida. Siguiendo un razonamiento análogo al de la MSECD de paso corto, la medida $\|d\|$ también funciona aquí. De esta manera, se puede formular la corrección para el caso de no terminación mediante el siguiente lema:

Lema 5 (Corrección en el caso de no terminación (auxiliar))**.** *Sea* Ω *un entorno sin nombres,* Δ *un entorno de máquina,* d *una expresión sin nombres,* c *un código de máquina. Si*

$$\Omega \vdash d \overset{\cong}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \Downarrow \Delta$$

entonces, para toda pila s*,*

$$\Delta, s \vdash c \overset{\cong}{\underset{\|d\|}{\Rightarrow}}$$

Demostración. Procedemos por coinducción. La idea principal es imitar la evaluación (semántica de $\mathsf{MiniML}^{\mathsf{dB}}$) de d en la máquina al evaluar c. Para las subexpresiones finitas de d (si es que las hay) empleamos el Teorema 3, mientras que para las subexpresiones infinitas de d aplicamos la hipótesis de coinducción. Adicionalmente, utilizamos la definición de $\overset{\cong}{\Rightarrow}_n$, incluyendo las reglas $\overset{\cong}{\Rightarrow}_n$-sleep y $\overset{\cong}{\Rightarrow}_n$-perform, cuando sea necesario. □

Finalmente, enunciamos el teorema de corrección para el caso de no terminación de la máquina, utilizando directamente la relación $\overset{\cong}{\Rightarrow}$ de la siguiente forma:

Teorema 6 (Corrección en el caso de no terminación)**.** *Sea* Ω *un entorno sin nombres,* Δ *un entorno de máquina,* d *una expresión sin nombres,* c *un código de máquina. Si*

$$\Omega \vdash d \overset{\cong}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \Downarrow \Delta$$

entonces, para toda pila s*,*

$$\Delta, s \vdash c \overset{\cong}{\Rightarrow}$$

Demostración. El resultado es una consecuencia directa del Lema 5 seguido del Lema 4. □

Este teorema se escribe en Coq como sigue:

```
Theorem Compilation_CNS_correctness:
∀ O,  ∀ d,  MML_dB_CNS O d →
      ∀ c,  Compilation_NS d c →
      ∀ D,  Compilation_NS_env O D →
      ∀ s,  BSMSECD_CNSC (D, s)  c.
```

Capítulo 5

Algoritmo de traducción a Coq

Durante este trabajo, a través de nuestro compilador de Mini-ML, hemos mostrado cómo a partir de la definición en semántica natural (coinductiva) de cada uno de los componentes de un compilador se puede realizar su correspondiente formalización en Coq. Ahora, nuestra intención es generalizar este método y escribirlo formalmente como un algoritmo.

5.1. Generalidades

El algoritmo 1 expresa cómo traducir una definición en semántica natural coinductiva de un compilador a su correspondiente formalización en Coq.

Podemos observar que los pasos del algoritmo poseen un nivel alto de abstracción, lo cual resulta favorable porque otorga libertad al realizar una implementación concreta; incluso se puede sacar ventaja de esta libertad si se aprovecha el trabajo relacionado previo. Por ejemplo, a partir de los resultados en [106], el paso 7 se podría realizar generando una función a partir de la definición inductiva I_N, correspondiente a la semántica natural N.

Ciertamente, el algoritmo sigue la codificación de cada uno de los componentes del compilador como la hemos ido mostrando en este trabajo. En general, el algoritmo se encarga primero de los lenguajes y luego de las traducciones.

5.2. Lenguajes

En la primera parte del algoritmo se tratan los lenguajes. En esta parte, para cada uno de los lenguajes del compilador, se indica cómo codificar su sintaxis abstracta, su semántica natural (cómputos finitos) y su semántica natural coinductiva (cómputos infinitos). Más precisamente, en el caso de la semántica natural, se indica cómo obtener además un intérprete que sea correcto y completo con respecto de la semántica para cómputos finitos; en el caso de la semántica natural coinductiva, se indica cómo hacer que este intérprete sea además correcto y completo con respecto a la semántica para cómputos infinitos.

5.2.1. Sintaxis abstracta

La sintaxis abstracta de un lenguaje, tal como se mostró en la Sección 3.1.1, se codifica como sintaxis abstracta de primer orden mediante una definición `Inductive` con tipo `Set` donde, para cada una de las construcciones del lenguaje se tiene que agregar el constructor correspondiente a dicha definición. De aquí la sintaxis abstracta queda lista para su extracción mediante el comando `Extraction`.

5.2.2. Semántica natural

En cuanto a la semántica natural, como se describió en la Sección 3.1.1 y en la Sección 3.2.2, donde en esta última se mencionan los detalles específicos de la codificación de la máquina, la semántica de un lenguaje se puede ver como una proposición lógica inductiva y por tanto codificarse como una definición `Inductive` con tipo `Prop` en Coq. Además, si se garantiza que la semántica es determinista, se puede obtener un intérprete que sigue la definición de la semántica pero que se escriba como una función recursiva `Fixpoint` agregando un parámetro natural `fuel`, este parámetro indica la profundidad de la recursión y garantiza que el intérprete necesariamente terminará. Posteriormente, se indica cómo garantizar que este intérprete sea correcto y completo con respecto a la semántica, esto es, emitiendo los lemas correspondientes a estas propiedades.

5.2.3. Semántica natural coinductiva

En analogía a la semántica natural que se codifica por medio de una definición `Inductive` con tipo `Prop`, la semántica natural coinductiva se codifica mediante una definición `CoInductive` con tipo `Prop`, tal como se describió en la Sección 4.1.1, y en específico para la máquina en la Sección 4.2.2.1. Posteriormente, el algoritmo indica la forma de garantizar, mediante la emisión de los lemas correspondientes, que el intérprete obtenido previamente a partir de la semántica natural sea correcto y completo con respecto a la semántica natural coinductiva (y no únicamente correcto y completo con respecto a la semántica natural). De esta manera, el intérprete está listo para su extracción mediante el comando `Extraction`.

5.3. Traducciones

En la segunda parte del algoritmo se tratan las traducciones. Aquí, para cada una de las traducciones del compilador, se indica cómo codificar su especificación en semántica natural, cómo obtener un compilador a partir de esta especificación y garantizar que este compilador sea correcto y completo con respecto a dicha especificación. Además, se indica cómo garantizar que la traducción sea correcta tanto para el caso de terminación como para el caso de no terminación.

5.3.1. Semántica natural

La especificación en semántica natural de una traducción, como se mostró en la Sección 3.1.3, y en particular en la Sección 3.2.2.1 en la generación de código de máquina, se puede codificar como una definición `Inductive` con tipo `Prop` donde, para cada regla de la traducción se debe agregar el constructor correspondiente, a la definición `Inductive`. Además, si la traducción es determinista entonces se indica que se debe emitir una función `Fixpoint` que siga la especificación de la traducción. En efecto, dicha función es un compilador. Se indica adicionalmente cómo, mediante los lemas respectivos, garantizar que este compilador sea correcto y completo con respecto a la especificación de la traducción.

5.3.2. Corrección

La corrección se establece como preservación semántica. Por tanto, primero se indica que se debe garantizar mediante un teorema que la traducción es correcta, es decir, que preserva la semántica, en el caso de terminación; después, se indica que se debe garantizar mediante un teorema que la traducción es correcta, es decir, que preserva la semántica, también en el caso de no terminación.

Una vez que se ha asegurado que la traducción es correcta, tanto en el caso de terminación como en el caso de no terminación, entonces se tiene por una parte que el compilador obtenido previamente es correcto y completo con respecto a la especificación de la traducción y, por otra, que es correcto total, queriendo decir esto último que preserva la semántica de ambos tipos de cómputos: finitos e infinitos. Por tanto, el compilador está listo para su extracción mediante el comando `Extraction`.

5.3.2.1. Casos especiales

No obstante, previo a garantizar la corrección en el caso de no terminación, se debe considerar que puede haber ciertos casos que requieren un tratamiento especial, como en el caso de los cómputos infinitos de la máquina que describimos en la Sección 4.2.2.1.

Si se analiza el algoritmo, como se observa en el paso 8, el caso en el que el lenguaje objetivo R de una traducción T es una representación posfija requiere un tratamiento especial y merece una explicación más detallada. Sea V el lenguaje fuente de T, d una construcción de V, si c es la traducción de d en R, entonces si R es una representación prefija al razonar sobre la evaluación de c en Coq, necesariamente se debe utilizar un constructor s asociado a la construcción (al frente) al iniciar la evaluación; de esta manera, se cumple con la condición de guardia. En cambio, si R es una representación posfija, necesariamente se debe utilizar un constructor s asociado a la construcción (pero atrás) al final del la evaluación; de esta forma, no se cumple con la condición de guardia de Coq, ya que ésta exige que se utilice necesariamente al inicio (al frente). Por ejemplo, sea `Plus e1 e2` una construcción de MiniML y `Plus_dB d1 d2` su traducción en $\mathsf{MiniML}^{\mathsf{dB}}$, entonces al evaluar `Plus_dB d1 d2` se comenzará utilizando un constructor de `Plus_dB`, asociado a la suma, al inicio, y se cumplirá con la condición

de guardia de Coq. En cambio, sea `Plus_dB d1 d2` una construcción de $\mathsf{MiniML}^{\mathsf{dB}}$ y `c1++ c2++ IAdd` su traducción en código de la MSECD de paso largo, entonces al evaluar `c1++ c2++ IAdd` no se comenzará utilizando un constructor de `IAdd`, asociado a la suma, aunque potencialmente se utilizará al final. En este último caso, se debe buscar una forma de expresarle y convencer a la condición de guardia de Coq que el constructor sí se utilizará, solo que al final de la evaluación de c. Esto es precisamente lo que hace la solución expuesta en la Sección 4.2.2.1; si se utiliza la relación auxiliar de dicha solución, se puede iniciar usando el constructor correspondiente a la regla sleep, el cual permite que se comience la evaluación de c sin usar un constructor s, asociado a la construcción, garantizando que dicho constructor s eventualmente se utilizará, lo que se expresa en el constructor correspondiente a la regla perform; la función medida $\|d\|$ indica el número de constructores que se utilizarán al final de la evaluación de c. Esto sin duda es una debilidad de la condición de guardia de Coq, que en este caso resulta demasiado inflexible. Es por esto que se recurrió a una solución indirecta.

En efecto, en nuestro compilador, en la traducción de MiniML a $\mathsf{MiniML}^{\mathsf{dB}}$ no fue necesario el uso de esta solución indirecta en lo absoluto debido a que $\mathsf{MiniML}^{\mathsf{dB}}$ es una representación prefija. En cambio, sí fue necesaria en la generación de código de la máquina MSECD de paso largo a partir de $\mathsf{MiniML}^{\mathsf{dB}}$, ya que el código de máquina es una representación posfija.

Algoritmo 1: Traducción de una definición de un compilador a Coq (primera parte)

Entrada: Una definición en semántica natural (coinductiva) de un compilador correcto total (donde todas las semánticas y traducciones son deterministas)

Salida: Una formalización del compilador en Coq (a partir de la cual se puede obtener una implementación verificada)

Para cada lenguaje L del compilador: lenguaje fuente, lenguajes intermedios y lenguaje de máquina se debe realizar lo siguiente:

1. Sea A la sintaxis abstracta de L, emitir una definición inductiva I_A con tipo `Set`
2. Para cada construcción del lenguaje $c \in A$ se debe agregar el constructor s, correspondiente a la construcción c, a la definición I_A
3. Emitir un comando de extracción con la definición I_A como argumento
4. Sea N la semántica natural de L, emitir una definición inductiva I_N con tipo `Prop`
5. Para cada regla $r \in N$ se debe agregar el constructor s, correspondiente a la regla r, a la definición I_N
6. Emitir un lema que enuncie el determinismo de N
7. Emitir una función (un intérprete) i que se comporte como la semántica natural N
8. Emitir un lema que enuncie que el intérprete i es correcto con respecto a la semántica natural N
9. Emitir un lema que enuncie que el intérprete i es completo con respecto a la semántica natural N
10. Sea CoN la semántica natural coinductiva de L, emitir una definición coinductiva C_{CoN} con tipo `Prop`
11. Para cada regla $r \in CoN$ se debe agregar el constructor s, correspondiente a la regla r, a la definición C_{CoN}
12. Emitir un lema que enuncie que el intérprete i es correcto con respecto a la semántica natural coinductiva CoN
13. Emitir un lema que enuncie que el intérprete i es completo con respecto a la semántica natural coinductiva CoN
14. Emitir un comando de extracción con el intérprete i como argumento

Algoritmo 1: Traducción de una definición de un compilador a Coq (segunda parte)

Para cada traducción T del compilador (traducciones intermedias y generación de código) se debe realizar lo siguiente:

1. Emitir una definición inductiva I_T con tipo `Prop`
2. Para cada regla de traducción $r \in T$ se debe agregar el constructor s, correspondiente a la regla r, a la definición I_T
3. Emitir un lema que enuncie el determinismo de T
4. Emitir una función (un compilador) c que se comporte como la traducción T
5. Emitir un lema que enuncie que el compilador c es correcto con respecto a la traducción T
6. Emitir un lema que enuncie que el compilador c es completo con respecto a la traducción T
7. Emitir un teorema que enuncie la corrección para el caso de terminación de la traducción T
8. Sea R el lenguaje objetivo de la traducción T, en caso de que R se trate de una representación posfija, se debe realizar lo siguiente:
   a) Emitir una definición auxiliar coinductiva C'_{CoN} análoga a C_{CoN}, pero que incluya un natural n como término adicional
   b) Agregar el constructor s, correspondiente a la regla sleep adaptada y mejorada, a la definición C'_{CoN}
   c) Agregar el constructor p, correspondiente a la regla perform adaptada, a la definición C'_{CoN}
   d) Para cada regla $r \in CoN$ verificar que r no haya sido capturada por alguna de las reglas sleep o perform; en caso de serlo, eliminar el constructor s, correspondiente a la regla r, de la definición C'_{CoN}
9. Emitir un lema que enuncie la equivalencia de C_{CoN} con C'_{CoN}
10. Sea V el lenguaje fuente de T, definir una función medida $\|d\|$ sobre las construcciones de V como sigue: si d es un átomo entonces $\|d\| = 0$, en otro caso d se trata de una construcción compuesta por subconstrucciones $d_1, \ldots, d_n$ y entonces $\|d\| = 1 + \|d_1\|$
11. Emitir una función (altura izquierda) h que se comporte como la medida $\|d\|$
12. Emitir un lema que enuncie la corrección para el caso de no terminación de la traducción T (utilizando C'_{CoN} y h)
13. Emitir un comando de extracción con el compilador c como argumento

Capítulo 6

Conclusiones y trabajo futuro

La semántica natural es un formalismo simple, fácil, intuitivo y ampliamente usado para definir la semántica de lenguajes de programación. No obstante, su uso como marco para demostrar la corrección de compiladores es mucho menos frecuente.

En este trabajo, extendimos la semántica natural (coinductiva) a fin de presentarla como un marco de verificación de la corrección total de compiladores en Coq; esta es una solución clara para realizar dicha tarea. De este modo, se puede obtener un compilador verificado para utilizarlo de manera independiente en la vida real y, si bien hemos ilustrado el uso de este marco con Mini-ML, un lenguaje funcional, todo indica que con las extensiones pertinentes podría utilizarse con otro tipo de lenguajes.

Aunque no lo hemos ilustrado aquí, con la semántica natural también se puede expresar y verificar la semántica estática de un lenguaje. Por ejemplo, en [40] se verifica la semántica estática de Mini-ML en Coq (aunque no es posible obtener un analizador semántico verificado). En trabajo futuro, pensamos extender este uso de la semántica natural para obtener un analizador semántico verificado.

Adicionalmente, a fin de tener un marco de verificación de compiladores completo, es necesario considerar el análisis léxico y el sintáctico; vislumbramos que con la "semántica" natural también se puede llevar a cabo estas tareas. La inspiración proviene de observar que, como lo menciona Kahn [55], el sistema de deducción natural está en el corazón de la semántica natural; de esta manera, se busca una estrategia de análisis sintáctico basada en deducción natural.

Por fortuna, existe una técnica de análisis sintáctico con estas características desde hace mucho tiempo. En la comunidad de la programación lógica, el análisis sintáctico como deducción (*parsing as deduction* [88, 98]) es un marco basado en deducción natural bien conocido y establecido. Por tanto, dado que ambos, la semántica natural y el análisis sintáctico como deducción, están basados en deducción natural, pensamos que podemos abstraerlos en un único formalismo capaz de expresar ambas: sintaxis y semántica. Así, se llegaría a la "semántica" natural como marco de verificación completo de compiladores en Coq.

A continuación ofrecemos una perspectiva de la semántica natural hacia el futuro en diferentes ámbitos.

6.1. Semántica natural para todos los cómputos

La semántica natural original sólo es capaz de expresar cómputos que terminan; a lo largo del tiempo, ésta ha sido una de sus debilidades principales. Pensamos que podemos extenderla para expresar todos lo cómputos posibles. Leroy [63] y Leroy y Grall [71] introducen la semántica natural coinductiva para los cómputos que no terminan; por su parte, Leroy y Grall [71] mencionan que el enfoque estándar para expresar cómputos que van mal (*goes wrong*) usando semántica natural es definir un predicado $e \overset{\lightning}{\Rightarrow}$ como lo hace Tofte en [105]. Además, Leroy y Grall señalan que esta solución no es satisfactoria por dos motivos: el primero es que se debe introducir reglas adicionales, lo que incrementa el tamaño de la semántica; el segundo es el riesgo de olvidar definir reglas para algunos casos de cómputos que vayan mal.

Dejando de lado, por el momento, estas objeciones, a primera vista se podría obtener una semántica natural que exprese todos los cómputos posibles: los finitos mediante el predicado $e \Rightarrow v$, los infinitos mediante el predicado $e \overset{\infty}{\Rightarrow}$ y los que van mal mediante el predicado $e \overset{\lightning}{\Rightarrow}$; por lo que, en principio, sería ideal demostrar en Coq el teorema que exprese que para toda expresión e se cumple que $e \Rightarrow v \vee e \overset{\infty}{\Rightarrow} \vee e \overset{\lightning}{\Rightarrow}$. Notemos que así se resolvería la segunda objeción de Leroy y Grall respecto al predicado $e \overset{\lightning}{\Rightarrow}$, puesto que así no habría forma de olvidar las reglas correspondientes a algunos casos de cómputos que van mal. El problema es que, en primera instancia y como lo señala Gimenez [43], no es posible demostrar en Coq este tipo de teoremas.

Gimenez [43] menciona que no hay esperanza de demostrar la proposición que expresa que dada una lista x cualquiera se cumple que $Fin\ x \vee Inf\ x$, donde *Fin* denota la propiedad de ser finita e *Inf* la de ser infinita. En nuestro contexto, esto equivaldría a la imposibilidad de demostrar que dada una expresión e necesariamente la evaluación de e es finita (ya sea porque llega una valor final o porque va mal) o infinita; es decir, es imposible demostrar el teorema planteado anteriormente acerca de la totalidad de la semántica natural.

Sin pensar que fuese posible, creemos haber encontrado una técnica, una manera indirecta, para demostrarlo. Nuestra solución sigue la misma estrategia explotada en las demostraciones por reflexión en Coq [21]. La inspiración proviene de observar que las funciones en Coq (en el tipo `Set`) necesariamente son totales, de modo que, si definimos la semántica natural como función (en `Set`), luego la reflejamos como una definición inductiva (en `Prop`) y finalmente establecemos una equivalencia entre ambas, podemos demostrar que la definición inductiva de la semántica natural es total apelando a la totalidad de la función por medio de la equivalencia entre la definición inductiva y la función.

En este trabajo, se escribió un intérprete que corresponde a la semántica natural (Sección 3.1.1), el cual posee un parámetro natural (al que nombramos `fuel`) para asegurar su terminación. Si afinamos este intérprete para que exprese cómputos que van mal y luego lo reflejamos en una definición inductiva, se obtiene una semántica natural con paso indexado, de la cual se puede demostrar un teorema expresando que ésta es total. Es decir, se puede demostrar que, para toda expresión e, para todo

número de pasos (combustible) i, $\exists v.e \underset{i}{\Rightarrow} v \vee e \overset{\lightning}{\underset{i}{\Rightarrow}} \vee\ e \underset{i}{\Rightarrow}$; lo cual quiere decir que e se evalúa a un valor final en i pasos, e va mal en i pasos, o la evaluación de e se queda sin combustible en i pasos.

Este resultado es sin duda importante para nuestros fines: notemos que se podría trabajar directamente con esta semántica. Luego, si se pudiera demostrar una equivalencia de $e \Rightarrow v$ con $\exists v.e \underset{i}{\Rightarrow} v$, otra de $e \overset{\lightning}{\Rightarrow}$ con $e \overset{\lightning}{\underset{i}{\Rightarrow}}$ y finalmente una tercera de $e \underset{i}{\Rightarrow}$ con $\blacksquare$, donde "$\blacksquare$" denota algo por determinar; en consecuencia también se podría demostrar que $e \Rightarrow v \vee\ e \overset{\lightning}{\Rightarrow} \vee \blacksquare$. Hasta el momento, hemos logrado demostrar las primeras dos, por lo que sólo queda la tercera por ser demostrada.

Nuestra primera tarea es determinar qué es $\blacksquare$; una primera aproximación sería establecer $\blacksquare$ como $e \overset{\infty}{\Rightarrow}$, buscando obtener $e \Rightarrow v \vee e \overset{\lightning}{\Rightarrow} \vee\ e \overset{\infty}{\Rightarrow}$. No obstante, el predicado $e \underset{i}{\Rightarrow}$ que expresa todos los cómputos que se quedan sin combustible no es equivalente a $e \overset{\infty}{\Rightarrow}$. Ciertamente, los cómputos infinitos $e \overset{\infty}{\Rightarrow}$ se quedan sin combustible sin importar cuan grande sea el valor de éste; pero, tomando el sentido contrario de la equivalencia (que es el que más nos interesa), no todos los cómputos que se quedan sin combustible $e \underset{i}{\Rightarrow}$ son cómputos infinitos.

Esto nos lleva a preguntarnos: además de los infinitos, ¿qué otros cómputos son expresados por los cómputos que se quedan sin combustible? Inmediatamente identificamos a los cómputos intermedios, es decir, aquellos en los que se acaba el combustible sin haber llegado a un valor final o sin que el cómputo haya ido mal. Bajo esta perspectiva, un cómputo infinito es un cómputo intermedio, ya que eventualmente se agotará el combustible sea cual sea su valor. No obstante, existen otros cómputos que se quedan sin combustible y que no son infinitos: son todos aquellos que de haber contado con la cantidad suficiente de combustible hubiesen llegado a un valor final o irían mal; a éstos los llamaremos cómputos estrictamente intermedios para diferenciarlos de los cómputos infinitos (que también son intermedios) y los denotaremos como $e \overset{\circ}{\Rightarrow} e'$.

Pensamos que los cómputos que se quedan sin combustible $e \underset{i}{\Rightarrow}$ son equivalentes a los cómputos infinitos unión los cómputos estrictamente intermedios. De ser esto cierto, entonces el $\blacksquare$ que estábamos buscando es $e \overset{\infty}{\Rightarrow} \vee\ e \overset{\circ}{\Rightarrow} e'$ y finalmente podríamos demostrar

$$e \Rightarrow v \vee\ e \overset{\lightning}{\Rightarrow} \vee\ e \overset{\infty}{\Rightarrow} \vee\ e \overset{\circ}{\Rightarrow} e'$$

y así llegar a una semántica natural capaz de expresar de todos los cómputos posibles con una demostración de este hecho mecanizada en Coq.

6.2. Concurrencia en semántica natural

La coinducción es una noción estrechamente relacionada con los cálculos de sistemas concurrentes, por lo que parece sensato utilizar semántica natural coinductiva para expresar primitivas de concurrencia en un lenguaje de programación; sin embargo, esta posibilidad ha sido poco explorada en la literatura (ver, por ejemplo, [109]).

En cambio, una línea de investigación más desarrollada, que ha recibido atención reciente, se basa en una lógica de separación de orden superior (*higher-order separation logic*) que utiliza pasos indexados [6], los cuales se internalizan en la lógica por medio de un operador modal (en lugar de ser tratados explícitamente como índices) para ofrecer soporte de concurrencia; luego, esta lógica se emplea para razonar sobre la semántica de paso corto de un lenguaje de programación concurrente. Esta línea de trabajo ha mostrado ser útil [54, 52, 57, 53].

Intuitivamente, los pasos indexados proveen un medio que otorga la fineza requerida para razonar sobre las transacciones atómicas presentes en concurrencia. Pensamos que podemos adoptar esta idea directamente en la semántica natural, es decir, introducir una semántica natural con pasos indexados (*step-indexed natural semantics*) en la que sea posible expresar la semántica de un lenguaje con concurrencia sin tener que usar una lógica de separación. Además, sería posible internalizar los índices mediante un operador modal, de lo que resultaría una semántica natural modal (*modal natural semantics*) capaz de expresar concurrencia. Ésta parece ser una solución limpia y elegante al tratamiento de concurrencia en semántica natural.

6.3. Nombres de las variables

En la literatura dedicada a la formalización de lenguajes en asistentes de demostración existen diversas técnicas para representar los nombres de las variables, principalmente con el fin de tener un tratamiento correcto de las variables ligadas (y evitar así, particularmente, el problema de captura de variables). Algunas de las técnicas más consolidadas son la sintaxis abstracta de orden superior (*Higher-Order Abstract Syntax*, HOAS) [89, 39], los índices de de Bruijn [37] y las técnicas nominales [93].

Sería deseable liberar al usuario de tener que tratar explícitamente esta tarea al hacer una formalización. Vislumbramos al menos dos maneras de abordar el problema en el marco de la semántica natural.

En el primer enfoque, se busca internalizar el manejo de los nombres de las variables dentro de la semántica natural misma; de esta manera, la manipulación de los nombres se realizaría en el marco, con lo cual queda oculto y por consiguiente es completamente transparente para el usuario, algo sin duda deseable. Éste es precisamente el enfoque que se sigue en la sintaxis abstracta de orden superior débil (*weak HOAS*) [28], la cual se basa en la teoría de contextos (*theory of contexts*) [97]. La idea principal de la sintaxis abstracta de orden superior débil es que unos pocos axiomas bastan para desarrollar una teoría robusta para el tratamiento de los nombres de variables en lenguajes de programación de forma completamente transparente para el usuario. De aquí que integrar la sintaxis abstracta de orden superior débil en el marco de la semántica natural parece una solución limpia, clara y elegante, puesto que la propia sintaxis de orden superior débil tiene como uno de sus orígenes, precisamente, en esta idea planteada por Burstall y Honsell en [24]. Por tanto, se trata de una solución natural que se muestra atractiva su desarrollo futuro.

En el segundo enfoque, se busca que un lenguaje dado se traduzca automáticamente a una representación fácil de manipular en un asistente de demostración,

en nuestro caso Coq; un ejemplo de de dicha representación sería la notación de de Bruijn. Existen marcos que siguen esta estrategia, tales como Autosubst [103, 102], en el cual el usuario escribe un lenguaje en HOAS y el marco se encarga automáticamente de generar el lenguaje en notación de de Bruijn junto con los lemas y teoremas relevantes para su manipulación. Tomando esta idea, podríamos hacer que el usuario escriba el lenguaje fácilmente en semántica natural usando la notación de HOAS y el marco se encargue de generar automáticamente la traducción a notación de de Bruijn especificada y verificada en semántica natural. En otras palabras, podemos tomar los resultados de Autosubst y escribirlos en términos de nuestro marco de semántica natural. Esta línea por desarrollar parece sólida, robusta y atractiva.

6.4. Semántica natural en otros asistentes de demostración

En este trabajo se mostró cómo codificar el marco de semántica natural en Coq. Independientemente, pensamos que el marco de semántica natural se puede implementar en otros asistentes de demostración, los cuales se prestan en mayor o en menor medida según el soporte de las características con las que cuentan. A continuación, discutimos la posible implementación del marco de semántica natural en algunos de estos asistentes de demostración con base en los mecanismos que ofrecen.

Agda [81, 80, 22, 110] es un asistente de demostración basado en teoría de tipos que se distingue particularmente por su innovador, flexible y experimental soporte de coinducción; ésta descansa sobre dos nociones centrales: copatrones (*copatterns*) [4, 2, 3, 104] y tipos con tamaño (*sized types*) [3, 1]. La idea general de los primeros es que al trabajar con estructuras finitas (tales como listas finitas), éstas se pueden definir vía constructores y manipular vía encaje de patrones (*pattern matching*). De aquí que se introducen los conceptos duales de observaciones (*observations*) para definir estructuras infinitas y de copatrones para poder realizar encaje de copatrones (*copattern matching*), esto último con el fin de manipular las estructuras infinitas. Por su parte, los tipos con tamaño se introducen particularmente con el fin de verificar que la definición de una función corecursiva esté bien definida; esto incluye verificar que las demostraciones de las propiedades sobre estructuras infinitas (definidas coinductivamente) estén bien definidas. Los tipos con tamaño representan una alternativa más robusta a la condición de guardia de Coq (ya se han realizado esfuerzos para su posible adopción en Coq [95]).

Por todas estas razones, la implementación de nuestro marco de semántica natural coinductiva en Agda parece ser inmediato, natural e ideal, sobre todo por su avanzado soporte de coinducción. De hecho, es posible que esto facilite la codificación de la semántica natural coinductiva de la máquina abstracta objetivo (al punto de que quizás ya no sea necesario definir las reglas sleep y perform que eran necesarias para convencer a la condición de guardia de Coq).

A cambio, Agda es un asistente de demostración menos maduro que Coq, debido en buena medida a que es más reciente; esto conlleva que no cuente con ciertas

características; por ejemplo, Agda carece de un lenguaje de tácticas, lo que torna el desarrollo de demostraciones más lento y demandante para el usuario.

Beluga [90, 91] es un lenguaje funcional con tipos dependientes que ofrece soporte nativo de sintaxis abstracta de orden superior, el cual está basado en teoría de tipos modal contextual (*contextual modal type theory*) [77] por lo que, en principio, libera al usuario del tratamiento de variables ligadas. En cuanto a su soporte de coinducción, comparte los mismos fundamentos que Agda [104] (aunque la implementación de estos principios aun está en fase de experimentación). Por ello, parece ofrecer también un buen soporte nativo para la implementación de nuestro marco de semántica natural coinductiva, lo cual sería interesante explorar en el futuro. La ventaja que Beluga tendría sobre Agda es que brinda soporte nativo de sintaxis abstracta de orden superior, por lo que, al parecer, nos liberaríamos de ofrecer un soporte propio para el tratamiento de variables ligadas dentro de nuestro marco.

Por su parte, Abella [42, 10] se puede entender como una extensión de Prolog, en otras palabras, como una extensión de lógica de primer orden a lógica de orden superior (*Higher-Order Logic*) [74]; además, cuenta con soporte nativo de sintaxis abstracta de orden superior. Al igual que en la programación lógica, al ser una extensión de ésta, si la especificación de un programa está definida como una relación, ésta puede escribirse directamente y obtener así un programa. Tal característica es atractiva para nuestros fines, puesto que en nuestro marco las definiciones en semántica natural son relaciones; de aquí que se obtienen de forma automática intérpretes o compiladores, según sea el caso. Por otro lado, Abella cuenta con una lógica aparte para razonar sobre los programas; es en ésta donde debemos expresar los correspondiente lemas y teoremas de nuestro marco. Por tanto, parece ser un demostrador de teoremas interactivo que soporta de manera limpia nuestro marco de semántica natural.

La principal ventaja de Abella es que acepta directamente relaciones como programas. En cambio, Coq, cuando se usa como lenguaje de programación, únicamente acepta funciones (y no relaciones en general); es por eso que para el presente trabajo tuvimos que demostrar que las definiciones tanto de las semánticas como las de las traducciones son deterministas (es decir, funciones) para poder obtener los intérpretes y compiladores correspondientes.

Wang y Nadathur [112] y Wang [111] estudian las verificación de ciertas transformaciones de un lenguaje funcional haciendo uso de Abella, pero recurren a semántica de paso corto y relaciones lógicas (*logical relations*) para establecer la corrección. Siguiendo esta línea, pensamos que implementaciones robustas de Prolog como CIAO Prolog [49], pueden extenderse con una lógica para razonar sobre los programas en él y se convertirían sin muchos esfuerzos en un asistente de demostración similar a Abella con la consecuente ganancia de las características adicionales soportadas por CIAO, entre ellas, su soporte de programación funcional. Un sistema así nos serviría para investigar y experimentar cuál es la mejor vía de codificación de nuestro marco de semántica natural comparando las diferentes alternativas (tales como la programación funcional y la lógica) y las relación entre ellas.

HOL es un asistente de demostración en la tradición de LCF [44], pero a diferencia de éste último está basado en la teoría de tipos simple de Church [27] y su semántica

se basa en teoría de conjuntos. Al fundarse en una teoría de tipos simple, HOL no soporta definiciones inductivas ni coinductivas de forma nativa (tiene soporte de éstas a través de bibliotecas). Quizá por esta razón, lo que se sugiere más simple e inmediato es escribir la semántica natural como una función y establecer el principio de inducción adecuado manualmente, con lo que se obtiene una semántica de paso largo funcional (*functional big-step semantics*) expuesta por Owens et al. [82]. Esta semántica es muy parecida al intérprete con el parámetro `fuel` que presentamos en la Sección 3.1.1, sólo que Owens et al. llaman reloj (*clock*) a dicho parámetro. Ellos toman esta definición funcional como especificación tanto como intérprete; por ello, existe un nivel de garantía menor respecto a nuestro marco.

En el nuestro, existe una especificación escrita como una definición inductiva en Coq; adicionalmente, hay un intérprete (definido como una función) que se demuestra que es correcto y completo respecto a la especificación. Por lo tanto, si hay un error en el intérprete (definición funcional de la semántica) éste se detectará al intentar demostrar la solidez respecto a la especificación. Por otro lado, si hay un error en la especificación se detectará al intentar demostrar la completitud del intérprete respecto de la misma. En cambio, en el trabajo de Owens et al. si hay un error en la definición funcional de la semántica (intérprete) no hay forma de detectarlo formalmente, al menos no de la forma que se hace en nuestro marco; tal parece que la única opción es la inspección manual de la definición por parte del usuario.

Por otra parte, en el trabajo de Owens et al. no se plantea el uso de coinducción ni para definir ni para razonar sobre cómputos finitos; en su lugar, se propone usar el reloj con tal efecto. Quedaría pendiente entonces mecanizar nuestro marco en HOL, particularmente las definiciones (co)inductivas por medio de las bibliotecas disponibles para comparar de forma más completa ambos enfoques. Cabe mencionar que el enfoque de Owens et al. se ha utilizado exitosamente en la verificación de una implementación de ML [58, 56], de modo que la implementación de nuestro marco en HOL podría beneficiar directamente el desarrollo de dicha implementación.

Isabelle [86, 87, 79] es un asistente de demostración genérico que tiene como fundamento una lógica de orden superior [8] que sirve como marco para definir otras lógicas; de esta manera, en principio, por cada cálculo o lógica distinta que se defina sobre el marco de Isabelle se obtiene el efecto de tener un asistente de demostración distinto. Por ejemplo, la teoría de conjuntos de Zermelo-Frankel está definida en el marco de Isabelle. Asimismo, existe una implementación de HOL en Isabelle, conocida como Isabelle/HOL, que al parecer ha recibido más atención y es la más desarrollada.

Desde luego, al trabajar en Isabelle/HOL tenemos un panorama muy similar al de HOL; dado que simplemente se trata de una implementación distinta pensamos que escribir una versión funcional de la semántica natural en Isabelle/HOL como la propuesta por Owens et al. [82] para HOL sería casi inmediato. Por otro lado, para escribir nuestro marco de semántica natural tendríamos que hacer uso de definiciones inductivas y coinductivas, las cuales, como en el caso de HOL, en Isabelle/HOL también son provistas a través de bibliotecas. Quizás una de las ventajas más significativas de trabajar con Isabelle/HOL es que cuenta con un soporte correcto de las técnicas nominales [107, 108], por lo que se podría delegar el tratamiento de las variables ligadas a dicho soporte.

6.5. Semántica natural en Maude

Maude [72, 31, 32, 30, 29, 41] es un lenguaje de especificación cuyos fundamentos parten de la teoría de categorías. Su idea principal es contar con una lógica general [73] donde la noción de deducción y cómputo coincidan. La lógica de reescritura (*rewriting logic*), fundamento de Maude, es el resultado de esta empresa. Así, se trata de un lenguaje de programación lógico en el sentido general del término. Intuitivamente, los cómputos en la lógica de reescritura se realizan mediante reescritura de términos.

Por otra parte, Maude contiene como sublenguaje al lenguaje funcional OBJ; aquí, la evaluación funcional de OBJ se realiza mediante simplificación ecuacional, es decir, OBJ está basado en lógica ecuacional. Debido a la generalidad de la lógica de reescritura, la lógica ecuacional se puede poner en términos de la primera mediante una correspondencia. Asimismo, otras lógicas, o, en términos de computación, otros tipos de lenguajes de programación, se pueden poner en términos de lógica de reescritura mediante la misma técnica (estableciendo una correspondencia), lo que hace a la lógica de reescritura un formalismo ideal para la unificación de distintos paradigmas de programación.

En particular, Maude también contiene programación orientada a objetos, que se unifica en él junto a la programación relacional y la funcional. No obstante, hay que mencionar que Maude ofrece una evaluación funcional específica para OBJ, no así para la orientación de objetos, para la cual se utiliza la evaluación de la lógica de reescritura.

Dado este panorama general de Maude, a continuación discutimos cómo se puede realizar la implementación de nuestro marco de semántica natural en él. Las definiciones en semántica natural en general se pueden escribir (como relaciones) en términos de lógica de reescritura, mientras que las definiciones en semántica natural deterministas se pueden escribir (como funciones) en OBJ. A simple vista, podemos notar que Maude se presta de forma ideal a la implementación de nuestro marco de semántica natural: sus características representan diversas ventajas.

En primer lugar, cuando en Coq escribíamos una definición en semántica natural como una proposición lógica (en el tipo `Prop`), esto es, como una relación, ésta no contaba con contenido computacional; en otras palabras, la definición no se podía utilizar para evaluar expresiones del lenguaje, o lo que es lo mismo, no se obtenía un intérprete. Esto nos obligaba a escribir la definición en semántica natural como función (en el tipo `Set`) para así poder obtener un intérprete y así nos restringía a trabajar únicamente con definiciones en semántica natural deterministas. En cambio, en Maude podemos escribir directamente una definición en semántica natural en general (no necesariamente determinista) en lógica de reescritura, la cual sirve directamente como intérprete (es decir, tiene contenido computacional), dicho de otro modo, es una especificación ejecutable de la semántica. En el caso particular de las definiciones en semántica natural deterministas, éstas se pueden escribir en OBJ, aunque también se pueden escribir en términos de lógica de reescritura. Escribir una definición en semántica natural determinista en OBJ parece sensato únicamente si su ejecución es más eficiente que escribirla en términos de lógica de reescritura. Entonces, en principio, dada una definición en semántica natural, en Maude basta con escribirla en

lógica de reescritura. Esto incluso nos lleva a postular a la lógica de reescritura como contenido computacional de las proposiciones lógicas (tipo `Prop`) en Coq, idea que resulta altamente atractiva.

Así como Coq cuenta con un mecanismo de extracción el cual para los programas funcionales (en el tipo `Set`) los extrae a código de OCaml, cabe la posibilidad de desarrollar un mecanismo de extracción que para las proposiciones lógicas (en el tipo `Prop`) genere código de Maude (lógica de reescritura). Similarmente, Coq es capaz de evaluar un programa funcional (en el tipo `Set`) de manera interna, así también sería capaz de evaluar internamente proposiciones lógicas (en el tipo `Prop`).

Por otro lado, Maude cuenta con herramientas para realizar verificación de modelos, con lo cual nos preguntamos si es posible verificar la corrección de un compilador de manera completamente automática. En Maude, esto consistiría en determinar si es posible expresar la corrección del compilador dentro de la lógica de la herramienta que se utilice y, de serlo, esperar que la herramienta verifique la corrección del compilador de forma completamente automatizada.

Por otra parte, Meseguer [73] ofrece un inspirador tratamiento categórico de traducciones entre lógicas. Puesto que, como lo menciona Kahn [55], la semántica natural puede ser vista como una lógica, entonces se puede hacer una formulación categórica del marco de semántica natural con todas las ventajas que esto conlleva. Sólo por mencionar una, es fácil pensar que dado un compilador se obtenga un decompilador de forma natural bajo este enfoque. Desde luego, obtener un decompilador tiene diversas ventajas; entre ellas, que serviría en la depuración de programas. Al ser este marco categórico sería muy general; por tanto, no solo serviría para expresar traducciones entre lenguajes de programación, sino también para expresar traducciones entre lenguajes de programación y quizás, lo que sería más interesante, para expresar traducciones entre lógicas y lenguajes de programación. Por supuesto, también serviría para expresar la semántica tanto de lógicas como de lenguajes de programación. Ciertamente, pensamos que esta marco se podría implementar de manera ideal en Maude, debido a que Maude está en plena consonancia con el enfoque categórico.

Sorprendentemente, el tratamiento categórico de traducciones de Meseguer parece no haber sido utilizado en la definición y verificación de compiladores; por tanto, para el conocimiento del autor, esta línea de investigación sería la primera en explotarlo en el ámbito de compiladores.

Por otro lado, Prolog cuenta con una amplia literatura y tradición como lenguaje de implementación de compiladores [34, 113, 114, 23, 33, 75, 49]. Como su lógica es la de primer orden (con ciertas restricciones), en principio los programas de Prolog se pueden escribir en lógica de reescritura en Maude, porque, como hemos mencionado previamente, la lógica de reescritura es más general. De aquí que la inmensa teoría y práctica para escribir compiladores en Prolog es, a primera vista, directamente trasladable para escribir compiladores en Maude. Asimismo, pensamos que muchos de estos resultados pueden ser de beneficio y utilizarse dentro del marco de semántica natural y de aquí, posiblemente, generalizarse en virtud del enfoque categórico de Meseguer.

Apéndice A

Demostraciones

Teorema 1 (Corrección en el caso de terminación)**.** *Sea* Γ *un entorno,* Ω *un entorno sin nombres,* e *una expresión,* d *una expresión sin nombres y* v *un valor. Si*

$$\Gamma \vdash e \Rightarrow v, \quad \Gamma \Downarrow \Omega, \quad \prod_1(\Gamma) \vdash e \;\Downarrow\; d$$

entonces, existe un valor sin nombres v_d *tal que* $v \Downarrow v_d$ *y*

$$\Omega \vdash d \Rightarrow v_d$$

Demostración. Por inducción sobre $\Gamma \vdash e \Rightarrow v$.
Casos base:

1. $e = n$.

   Hipótesis $\Gamma \vdash n \Rightarrow n, \quad \Gamma \Downarrow \Omega, \quad \prod_1(\Gamma) \vdash n \;\Downarrow\; d$. Afirmamos que existe $v_d = n$, $n \Downarrow n$ se demuestra por definición. Usando la hipótesis $\prod_1(\Gamma) \vdash n \;\Downarrow\; d$, por definición de $\Downarrow$ necesariamente $d = n$. Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash n \Rightarrow n$, éste queda demostrado por definición de $\Rightarrow$.

2. $e = b$. Análogo al caso 1.

   Hipótesis $\Gamma \vdash b \Rightarrow b, \quad \Gamma \Downarrow \Omega, \quad \prod_1(\Gamma) \vdash b \;\Downarrow\; d$. Afirmamos que existe $v_d = b$, $b \Downarrow b$ se demuestra por definición. Usando la hipótesis $\prod_1(\Gamma) \vdash b \;\Downarrow\; d$, por definición de $\Downarrow$ necesariamente $d = b$. Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash b \Rightarrow b$, éste queda demostrado por definición de $\Rightarrow$.

3. $e = x$.

   La demostración descansa en la proposición que enuncia que si $\Gamma \vdash x \longmapsto v$, $\Gamma \Downarrow \Omega, \quad v \Downarrow v_d, \quad \prod_1(\Gamma) \vdash x \Downarrow \underline{i}$ entonces $\Omega \vdash \underline{i} \longmapsto v_d$, la cual se demuestra por inducción sobre $\Gamma \vdash x \longmapsto v$. (También se puede demostrar por inducción sobre Γ.)

4. $e = \lambda x.e$. Análogo al caso 1.

   Hipótesis $\Gamma \vdash \lambda x.e \Rightarrow (\lambda x.e)[\Gamma]$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash \lambda x.e \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash \lambda x.e \Downarrow d$, por definición de $\Downarrow$ necesariamente $x \cdot \prod_1(\Gamma) \vdash e \Downarrow d_1$, $d = \lambda.d_1$. Afirmamos que existe $v_d = (\lambda.d_1)[\Omega]$; $(\lambda x.e)[\Gamma] \Downarrow (\lambda.d_1)[\Omega]$ se demuestra por definición, empleando las hipótesis $x \cdot \prod_1(\Gamma) \vdash e \Downarrow d_1$ y $\Gamma \Downarrow \Omega$. Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash \lambda.d_1 \Rightarrow (\lambda.d_1)[\Omega]$, éste se demuestra por definición de $\Rightarrow$.

5. $e = \mu f.\lambda x.e$. Análogo al caso 1.

   Hipótesis $\Gamma \vdash \mu f.\lambda x.e \Rightarrow (\mu f.\lambda x.e)[\Gamma]$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash \mu f.\lambda x.e \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash \mu f.\lambda x.e \Downarrow d$, por definición de $\Downarrow$ necesariamente $x \cdot f \cdot \prod_1(\Gamma) \vdash e \Downarrow d_1$, $d = \mu.\lambda.d_1$. Afirmamos que existe $v_d = (\mu.\lambda.d_1)[\Omega]$; $(\mu f.\lambda x.e)[\Gamma] \Downarrow (\mu.\lambda.d_1)[\Omega]$ se demuestra por definición, empleando la hipótesis $x \cdot f \cdot \prod_1(\Gamma) \vdash e \Downarrow d_1$ y $\Gamma \Downarrow \Omega$. Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash \mu.\lambda.d_1 \Rightarrow (\mu.\lambda.d_1)[\Omega]$, éste se demuestra por definición de $\Rightarrow$.

Casos inductivos:

1. $e = e_1 \star e_2$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow n_1$, $\Gamma \vdash e_2 \Rightarrow n_2$, $\Gamma \vdash e_1 \star e_2 \Rightarrow n_1 \star n_2$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash e_1 \star e_2 \Downarrow d$.

   Afirmamos que existe $v_d = n_1 \star n_2$, $n_1 \star n_2 \Downarrow n_1 \star n_2$ se demuestra por definición. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1 \star e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1 \star d_2$. Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash d_1 \star d_2 \Rightarrow n_1 \star n_2$.

   a) $\Omega \vdash d_1 \Rightarrow n_1$ por hipótesis de inducción.

   b) $\Omega \vdash d_2 \Rightarrow n_2$ por hipótesis de inducción.

   c) $\Omega \vdash d_1 \star d_2 \Rightarrow n_1 \star n_2$ por definición empleando $1a$ y $1b$.

2. $e =$ `if` e_1 `then` e_2 `else` e_3.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow true$, $\Gamma \vdash e_2 \Rightarrow v$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash$ `if` e_1 `then` e_2 `else` $e_3 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash$ `if` e_1 `then` e_2 `else` $e_3 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $\prod_1(\Gamma) \vdash e_3 \Downarrow d_3$, $d =$ `if` d_1 `then` d_2 `else` d_3. Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash$ `if` d_1 `then` d_2 `else` $d_3 \Rightarrow v_d$.

   a) $\Omega \vdash d_1 \Rightarrow true$ por hipótesis de inducción.

   b) $\Omega \vdash d_2 \Rightarrow v_d$ por hipótesis de inducción (tal que $v \Downarrow v_d$).

c) $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Rightarrow v_d$ por definición empleando $2a$ y $2b$.

3. $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow false$, $\Gamma \vdash e_3 \Rightarrow v$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $\prod_1(\Gamma) \vdash e_3 \Downarrow d_3$, $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$. Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Rightarrow v_d$.

   a) $\Omega \vdash d_1 \Rightarrow false$ por hipótesis de inducción.

   b) $\Omega \vdash d_3 \Rightarrow v_d$ por hipótesis de inducción (tal que $v \Downarrow v_d$).

   c) $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Rightarrow v_d$ por definición empleando $3a$ y $3b$.

4. $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow v_1$, $(x, v_1) \cdot \Gamma \vdash e_2 \Rightarrow v$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $x \cdot \prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$. Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \Rightarrow v_d$.

   a) $\Omega \vdash d_1 \Rightarrow v_{d_1}$ por hipótesis de inducción (tal que $v_1 \Downarrow v_{d_1}$).

   b) $(x, v_1) \cdot \Gamma \Downarrow v_{d_1} \cdot \Omega$ por definición empleando $v_1 \Downarrow v_{d_1}$ de $4a$ y la hipótesis $\Gamma \Downarrow \Omega$.

   c) $v_{d_1} \cdot \Omega \vdash d_2 \Rightarrow v_d$ por hipótesis de inducción (tal que $v \Downarrow v_d$). Al aplicar la hipótesis de inducción se emplea $4b$.

   d) $\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \Rightarrow v_d$ por definición empleando $4a$ y $4c$.

5. $e = e_1\ e_2$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow (\lambda x.e)[\Gamma_1]$, $\Gamma \vdash e_2 \Rightarrow v_2$, $(x, v_2) \cdot \Gamma_1 \vdash e \Rightarrow v$, $\Gamma \vdash e_1\ e_2 \Rightarrow v$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$ por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1\ d_2$.

   Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash d_1\ d_2 \Rightarrow v_d$.

   a) $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$ por hipótesis de inducción (tal que $\Gamma_1 \Downarrow \Omega_1$, $x \cdot \prod_1(\Gamma_1) \vdash e \Downarrow d$ y $(\lambda x.e)[\Gamma_1] \Downarrow (\lambda.d)[\Omega_1]$).

b) $\Omega \vdash d_2 \Rightarrow v_{d_2}$ por hipótesis de inducción (tal que $v_2 \Downarrow v_{d_2}$).

c) $v_{d_2} \cdot \Omega_1 \vdash d \Rightarrow v_d$ por hipótesis de inducción (tal que $v \Downarrow v_d$).

d) $\Omega \vdash d_1\ d_2 \Rightarrow v_d$ por definición empleando $5a$, $5b$ y $5c$.

6. $e = e_1\ e_2$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow (\mu f.\lambda x.e)[\Gamma_1]$, $\Gamma \vdash e_2 \Rightarrow v_2$, $(x, v_2) \cdot (f, (\mu f.\lambda x.e)[\Gamma_1]) \cdot \Gamma_1 \vdash e \Rightarrow v$, $\Gamma \vdash e_1\ e_2 \Rightarrow v$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$ por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1\ d_2$.

   Nos encontramos ahora en posición de demostrar el resultado principal $\Omega \vdash d_1\ d_2 \Rightarrow v_d$.

   a) $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$ por hipótesis de inducción (tal que $\Gamma_1 \Downarrow \Omega_1$, $x \cdot f \cdot \prod_1(\Gamma_1) \vdash e \Downarrow d$ y $(\mu f.\lambda x.e)[\Gamma_1] \Downarrow (\mu.\lambda.d)[\Omega_1]$).

   b) $\Omega \vdash d_2 \Rightarrow v_{d_2}$ (tal que $v_2 \Downarrow v_{d_2}$).

   c) $v_{d_2} \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \Rightarrow v_d$ por hipótesis de inducción (tal que $v \Downarrow v_d$).

   d) $\Omega \vdash d_1\ d_2 \Rightarrow v_d$ por definición empleando $6a$, $6b$ y $6c$. □

Teorema 2. *Si* $\Omega \vdash d \Rightarrow v$, *entonces* $([\![d]\!] \cdot c, [\![\Omega]\!], s) \overset{+}{\to} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ *para todo código* c *y pila* s.

Demostración. Por inducción sobre $\Omega \vdash d \Rightarrow v$.
Casos base:

(I) $d = n$. Por hipótesis $\Omega \vdash n \Rightarrow n$. Dado que por definición $[\![n]\!] = \mathsf{IConst}\ n$ y $[\![n]\!] = n$, debemos demostrar $(\mathsf{IConst}\ n \cdot c, [\![\Omega]\!], s) \overset{+}{\to} (c, [\![\Omega]\!], n \cdot s)$, lo que se tiene simplemente por definición de la semántica de paso corto de la máquina, es decir, de la transición $\to$ correspondiente a la instrucción $\mathsf{IConst}\ n$.

(II) $d = b$. Análogo al caso I. Hipótesis $\Omega \vdash b \Rightarrow b$; $[\![b]\!] = \mathsf{IConstb}\ b$ y $[\![b]\!] = b$.

(III) $d = \underline{i}$. La demostración se sigue de la proposición de que si $\Omega \vdash \underline{i} \longmapsto v$ entonces $[\![\Omega]\!](i) = [\![v]\!]$ la cual se demuestra simplemente por inducción sobre Ω.

(IV) $d = \lambda.d$. Análogo al caso I. Hipótesis $\Omega \vdash \lambda.d \Rightarrow (\lambda.d)[\Omega]$; $[\![\lambda.d]\!] = \mathsf{IClos}([\![d]\!] \cdot \mathsf{IRet})$ y $[\![(\lambda.d)[\Omega]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega]\!]]$.

(V) $d = \mu.\lambda.d$. Análogo al caso I. Hipótesis $\Omega \vdash \mu.\lambda.d \Rightarrow (\mu.\lambda.d)[\Omega]$; $[\![\mu.\lambda.d]\!] = \mathsf{IClos}_{rec}([\![d]\!] \cdot \mathsf{IRet})$ y $[\![(\mu.\lambda.d)[\Omega]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega]\!]]_{rec}$.

Casos inductivos:

(VI) $d = d_1 \star d_2$.

Por hipótesis $\Omega \vdash d_1 \Rightarrow n_1$, $\Omega \vdash d_2 \Rightarrow n_2$, $\Omega \vdash d_1 \star d_2 \Rightarrow n_1 \star n_2$. Debemos demostrar $(\llbracket d_1 \star d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket n_1 \star n_2 \rrbracket \cdot s)$.

a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, n_1 \star n_2 \cdot s)$ por definición $\llbracket d_1 \star d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp}$ y $\llbracket n_1 \star n_2 \rrbracket = n_1 \star n_2$.

b) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, n_1 \cdot s)$ por transitividad de $\stackrel{+}{\to}$, aplicando la hipótesis de inducción, y por definición $\llbracket n_1 \rrbracket = n_1$.

c) $(\llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, n_1 \cdot s) \stackrel{+}{\to} (\mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, n_2 \cdot n_1 \cdot s)$ por transitividad de $\stackrel{+}{\to}$, aplicando la hipótesis de inducción, y por definición $\llbracket n_2 \rrbracket = n_2$.

d) $(\mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, n_2 \cdot n_1 \cdot s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, n_1 \star n_2 \cdot s)$ por definición de $\to$ de la instrucción IOp.

(VII) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Por hipótesis $\Omega \vdash d_1 \Rightarrow true$, $\quad \Omega \vdash d_2 \Rightarrow v$. Por demostrar $(\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$.

a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición $\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$.

b) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to}$ $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, true \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

c) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, true \cdot s) \stackrel{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s)$ por definición de $\to$ de la instrucción ISel.

d) $(\llbracket d_2 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s) \stackrel{+}{\to} (\mathsf{IJoin}, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot (c, [\,]) \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

e) $(\mathsf{IJoin}, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot (c, [\,]) \cdot s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición de $\to$ de la instrucción IJoin.

(VIII) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Por hipótesis $\Omega \vdash d_1 \Rightarrow false$, $\quad \Omega \vdash d_3 \Rightarrow v$. Por demostrar $(\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$.

a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición $\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$.

b) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to}$
$(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, false \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

c) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, false \cdot s) \stackrel{+}{\to} (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s)$ por definición de $\to$ de la instrucción ISel.

d) $(\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s) \stackrel{+}{\to} (\mathsf{IJoin}, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot (c, [\,]) \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando hipótesis de inducción.

e) $(\mathsf{IJoin}, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot (c, [\,]) \cdot s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición de $\to$ de la instrucción IJoin.

(IX) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Por hipótesis $\Omega \vdash d_1 \Rightarrow v_1, \quad v_1 \cdot \Omega \vdash d_2 \Rightarrow v, \quad \Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \Rightarrow v$. Por demostrar $(\llbracket \texttt{let}\ d_1\ \texttt{in}\ d_2\ \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$.

a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c,\ \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$
por definición $\llbracket \texttt{let}\ \ d_1\ \ \texttt{in}\ \ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet}$.

b) $(\llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c,\ \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (\mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_1 \rrbracket \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

c) $(\mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c,\ \llbracket \Omega \rrbracket, \llbracket v_1 \rrbracket \cdot s) \stackrel{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket v_1 \rrbracket \cdot \llbracket \Omega \rrbracket, s)$ por definición de $\to$ de la instrucción ILet.

d) $(\llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket v_1 \rrbracket \cdot \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (\mathsf{IEndLet} \cdot c, \llbracket v_1 \rrbracket \cdot \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

e) $(\mathsf{IEndLet} \cdot c,\ \llbracket v_1 \rrbracket \cdot \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición de $\to$ de la instrucción IEndLet.

(X) $d = d_1\ d_2$.

Por hipótesis $\quad \Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1], \quad \Omega \vdash d_2 \Rightarrow v_2, \quad v_2 \cdot \Omega_1 \vdash d \Rightarrow v$,
$\Omega \vdash d_1\ d_2 \Rightarrow v$. Por demostrar $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$.

a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición
$\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$

b) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

c) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s) \stackrel{+}{\to} (\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

d) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s) \stackrel{+}{\to} (\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket] \cdot s)$ por definición $\llbracket (\lambda.d)[\Omega_1] \rrbracket = (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]$.

e) $(\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket] \cdot s) \stackrel{+}{\to} (\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s)$ por definición de $\to$ de la instrucción IApp.

f) $(\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s) \stackrel{+}{\to} (\mathsf{IRet}, \llbracket v_2 \rrbracket \cdot \llbracket \Omega_1 \rrbracket, \llbracket v \rrbracket \cdot (c, \llbracket \Omega \rrbracket) \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

g) $(\mathsf{IRet}, \llbracket v_2 \rrbracket \cdot \llbracket \Omega_1 \rrbracket, \llbracket v \rrbracket \cdot (c, \llbracket \Omega \rrbracket) \cdot s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición de $\to$ de la instrucción IRet.

(XI) $d = d_1\ d_2$.

Por hipótesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \Rightarrow v$, $\Omega \vdash d_1\ d_2 \Rightarrow v$. Por demostrar $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$.

a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición $\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$

b) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \stackrel{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

c) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s) \stackrel{+}{\to} (\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

d) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s) \stackrel{+}{\to} (\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot s)$ por definición $\llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket = (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec}$.

e) $(\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot s) \stackrel{+}{\to}$ $(\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s)$ por definición de $\to$ de la instrucción IApp.

f) $(\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s) \stackrel{+}{\to}$ $(\mathsf{IRet}, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot \llbracket \Omega_1 \rrbracket, \llbracket v \rrbracket \cdot (c, \llbracket \Omega \rrbracket) \cdot s)$ por transitividad de $\stackrel{+}{\to}$, y aplicando la hipótesis de inducción.

g) $(\mathsf{IRet}, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot \llbracket \Omega_1 \rrbracket, \llbracket v \rrbracket \cdot (c, \llbracket \Omega \rrbracket) \cdot s) \stackrel{+}{\to} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ por definición de $\to$ de la instrucción IRet. □

Lema 1. *Sean Δ, Δ_1, Δ_2 entornos de máquina; s, s_1, s_2 pilas; c_1, c_2 códigos de máquina. Si*

$$\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1), \qquad \Delta_1, s_1 \vdash c_2 \Rightarrow (\Delta_2, s_2)$$

entonces

$$\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta_2, s_2)$$

Demostración. Por inducción sobre $\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)$.
Caso base:

(I) $c_1 = []$.

Hipótesis $\Delta, s \vdash [] \Rightarrow (\Delta, s), \quad \Delta, s \vdash c_2 \Rightarrow (\Delta_2, s_2)$.
Por demostrar $\Delta, s \vdash [] \cdot c_2 \Rightarrow (\Delta_2, s_2)$, debido a que $[] \cdot c_2 = c_2$ nuestro objetivo se reduce a $\Delta, s \vdash c_2 \Rightarrow (\Delta_2, s_2)$ al que se llega por hipótesis.

Caso inductivo:

(II) $c_1 \neq []$ es decir $c_1 = i \cdot c$.

Hipótesis $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1), \quad \Delta_1, s_1 \vdash c \Rightarrow (\Delta_2, s_2), \quad \Delta_2, s_2 \vdash c_2 \Rightarrow (\Delta_3, s_3)$.
Por demostrar $\Delta, s \vdash i \cdot c \cdot c_2 \Rightarrow (\Delta_3, s_3)$.

a) Aplicando la hipótesis de inducción sobre $\Delta_1, s_1 \vdash c \Rightarrow (\Delta_2, s_2)$ y $\Delta_2, s_2 \vdash c_2 \Rightarrow (\Delta_3, s_3)$ tenemos $\Delta_1, s_1 \vdash c \cdot c_2 \Rightarrow (\Delta_3, s_3)$.

b) Usando $(\Delta, s) \vdash i \Rightarrow (\Delta_1, s_1)$ y $\Delta_1, s_1 \vdash c \cdot c_2 \Rightarrow (\Delta_3, s_3)$ por definición de $\Rightarrow$ se concluye $\Delta, s \vdash i \cdot c \cdot c_2 \Rightarrow (\Delta_3, s_3)$. □

Teorema 3 (Corrección en el caso de terminación)**.** *Sea Ω un entorno sin nombres, Δ un entorno de máquina, d una expresión sin nombres, c un código de máquina, v un valor sin nombres. Si*

$$\Omega \vdash d \Rightarrow v, \quad d \Downarrow c, \quad \Omega \Downarrow \Delta$$

entonces, existe un valor de máquina v_m tal que $v \Downarrow v_m$ y para toda pila s,

$$\Delta, s \vdash c \Rightarrow (\Delta, v_m \cdot s)$$

Demostración. Por inducción sobre $\Omega \vdash d \Rightarrow v$.

Casos base:

(I) $d = n$.

Hipótesis $\Omega \vdash n \Rightarrow n, \quad n \Downarrow c, \quad \Omega \Downarrow \Delta$. Afirmamos que existe $v_m = n$, $n \Downarrow n$ se demuestra por definición. Usando la hipótesis $n \Downarrow c$, por definición de $\Downarrow$ necesariamente $c = \mathsf{IConst}\ n$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash \mathsf{IConst}\ n \Rightarrow (\Delta, n \cdot s)$, éste se demuestra por definición de $\Rightarrow$ de la instrucción IConst.

(II) $d = b$. Análogo al caso I.

Hipótesis $\Omega \vdash b \Rightarrow b, \quad b \Downarrow c, \quad \Omega \Downarrow \Delta$. Afirmamos que existe $v_m = b$, $b \Downarrow b$ se demuestra por definición. Usando la hipótesis $b \Downarrow c$, por definición de $\Downarrow$ necesariamente $c = \mathsf{IConstb}\ b$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash \mathsf{IConstb}\ b \Rightarrow (\Delta, b \cdot s)$, éste se demuestra por definición de $\Rightarrow$ de la instrucción IConstb.

(III) $d = \underline{i}$.

La demostración descansa en la proposición que enuncia que si $\Omega \vdash \underline{i} \longmapsto v$, $\Omega \Downarrow \Delta$, $v \Downarrow v_m$ entonces $\Delta \vdash \underline{i} \longmapsto v_m$, la cual se demuestra simplemente por inducción sobre $\Omega \vdash \underline{i} \longmapsto v$. (También se puede demostrar por inducción sobre Ω.)

(IV) $d = \lambda.d$. Análogo al caso I.

Hipótesis $\Omega \vdash \lambda.d \Rightarrow (\lambda.d)[\Omega], \quad \lambda.d \Downarrow c, \quad \Omega \Downarrow \Delta$. Usando la hipótesis $\lambda.d \Downarrow c$, por definición de $\Downarrow$ necesariamente $d \Downarrow c_1, \quad c = \mathsf{IClos}\ c_1$. Afirmamos que existe $v_m = c_1[\Delta]$; $(\lambda.d)[\Omega] \Downarrow c_1[\Delta]$ se demuestra por definición, empleando las hipótesis $d \Downarrow c_1$ y $\Omega \Downarrow \Delta$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash \mathsf{IClos}\ c_1 \Rightarrow (\Delta, c_1[\Delta] \cdot s)$, éste se demuestra por definición de $\Rightarrow$ de la instrucción IClos.

(V) $d = \mu.\lambda.d$. Análogo al caso I.

Hipótesis $\Omega \vdash \mu.\lambda.d \Rightarrow (\mu.\lambda.d)[\Omega], \quad \mu.\lambda.d \Downarrow c, \quad \Omega \Downarrow \Delta$. Usando la hipótesis $\mu.\lambda.d \Downarrow c$, por definición de $\Downarrow$ necesariamente $d \Downarrow c_1, \quad c = \mathsf{IClos}_{rec}\ c_1$. Afirmamos que existe $v_m = c_1[\Delta]_{rec}$; $(\mu.\lambda.d)[\Omega] \Downarrow c_1[\Delta]_{rec}$ se demuestra por definición, empleando las hipótesis $d \Downarrow c_1$ y $\Omega \Downarrow \Delta$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash \mathsf{IClos}_{rec}\ c_1 \Rightarrow (\Delta, c_1[\Delta]_{rec} \cdot s)$, éste se demuestra por definición de $\Rightarrow$ de la instrucción IClos_{rec}.

Casos inductivos:

(VI) $d = d_1 \star d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow n_1, \quad \Omega \vdash d_2 \Rightarrow n_2, \quad \Omega \vdash d_1 \star d_2 \Rightarrow n_1 \star n_2, \quad d_1 \star d_2 \Downarrow c$, $\Omega \Downarrow \Delta$. Afirmamos que existe $v_m = n_1 \star n_2, \quad n_1 \star n_2 \Downarrow n_1 \star n_2$ se demuestra por definición. Usando la hipótesis $d_1 \star d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IOp}$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \Rightarrow (\Delta, n_1 \star n_2 \cdot s)$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, n_1 \cdot s)$ por hipótesis de inducción.

b) $\Delta, n_1 \cdot s \vdash c_2 \Rightarrow (\Delta, n_2 \cdot n_1 \cdot s)$ por hipótesis de inducción.

c) $\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta, n_2 \cdot n_1 \cdot s)$ por Lema 1 sobre VI.*a* y VI.*b*.

d) $\Delta, n_2 \cdot n_1 \cdot s \vdash \mathsf{IOp} \Rightarrow (\Delta, n_1 \star n_2 \cdot s)$ por definición.

e) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \Rightarrow (\Delta, n_1 \star n_2 \cdot s)$ por Lema 1 sobre VI.*c* y VI.*d*.

(VII) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hipótesis $\Omega \vdash d_1 \Rightarrow \mathit{true}, \quad \Omega \vdash d_2 \Rightarrow v, \quad \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c, \quad \Omega \Downarrow \Delta$. Usando la hipótesis $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad d_3 \Downarrow c_3, \quad c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, \mathit{true} \cdot s)$ por hipótesis de inducción.

b) $\Delta, s \vdash c_2 \Rightarrow (\Delta, v_m \cdot s)$ por hipótesis de inducción (tal que $v \Downarrow v_m$).

c) $\Delta, \mathit{true} \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$ por definición empleando VIII.*b*.

d) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$ por Lema 1 sobre VIII.*a* y VIII.*c*.

(VIII) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hipótesis $\Omega \vdash d_1 \Rightarrow \mathit{false}, \quad \Omega \vdash d_3 \Rightarrow v, \quad \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c, \quad \Omega \Downarrow \Delta$. Usando la hipótesis $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$, por definición $\Downarrow$ necesariamente $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad d_3 \Downarrow c_3, \quad c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, \mathit{false} \cdot s)$ por hipótesis de inducción.

b) $\Delta, s \vdash c_3 \Rightarrow (\Delta, v_m \cdot s)$ por hipótesis de inducción (tal que $v \Downarrow v_m$).

c) $\Delta, \mathit{false} \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$ por definición empleando VIII.*b*.

d) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$ por Lema 1 sobre VIII.*a* y VIII.*c*.

(IX) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow v_1, \quad v_1 \cdot \Omega \vdash d_2 \Rightarrow v, \quad \Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \Rightarrow v, \quad \texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet}$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \Rightarrow (\Delta, v_m \cdot s)$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, v_{m_1} \cdot s)$ por hipótesis de inducción (tal que $v_1 \Downarrow v_{m_1}$).

b) $\Delta, v_{m_1} \cdot s \vdash \mathsf{ILet} \Rightarrow (v_{m_1} \cdot \Delta, s)$ por definición.

c) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \Rightarrow (v_{m_1} \cdot \Delta, s)$ por Lema 1 sobre IX.*a* y IX.*b*.

d) $v_{m_1} \cdot \Delta, s \vdash c_2 \Rightarrow (v_{m_1} \cdot \Delta, v_m \cdot s)$ por hipótesis de inducción (tal que $v \Downarrow v_m$).

e) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \Rightarrow (v_{m_1} \cdot \Delta, v_m \cdot s)$ por Lema 1 sobre IX.c y IX.d.

f) $v_{m_1} \cdot \Delta, v_m \cdot s \vdash \mathsf{IEndLet} \Rightarrow (\Delta, v_m \cdot s)$ por definición.

g) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \Rightarrow (\Delta, v_m \cdot s)$ por Lema 1 sobre IX.e y IX.f.

(X) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot \Omega_1 \vdash d \Rightarrow v$, $\Omega \vdash d_1\ d_2 \Rightarrow v$, $d_1\ d_2 \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $d_1\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IApp}$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Rightarrow (\Delta, m_v \cdot s)$.

a) $d \Downarrow c$, $\Omega_1 \Downarrow \Delta_1$, $\Delta, s \vdash c_1 \Rightarrow (\Delta, c[\Delta_1] \cdot s)$ por hipótesis de inducción (tal que $(\lambda.d)[\Omega_1] \Downarrow c[\Delta_1]$).

b) $\Delta, c[\Delta_1] \cdot s \vdash c_2 \Rightarrow (\Delta, v_{m_2} \cdot c[\Delta_1] \cdot s)$ por hipótesis de inducción (tal que $v_2 \Downarrow v_{m_2}$).

c) $\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta, v_{m_2} \cdot c[\Delta_1] \cdot s)$ por Lema 1 sobre X.a y X.b.

d) $v_{m_2} \cdot \Delta_1, s \vdash c \Rightarrow (v_{m_2} \cdot \Delta_1, v_m \cdot s)$ por hipótesis de inducción (tal que $v \Downarrow v_m$).

e) $\Delta, v_{m_2} \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Rightarrow (\Delta, v_m \cdot s)$ por definición empleando X.d.

f) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Rightarrow (\Delta, v_m \cdot s)$ por Lema 1 sobre X.c y X.e.

(XI) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \Rightarrow v$, $\Omega \vdash d_1\ d_2 \Rightarrow v$, $d_1\ d_2 \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $d_1\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IApp}$. Nos encontramos ahora en posición de demostrar el resultado principal $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Rightarrow (\Delta, m_v \cdot s)$.

a) $d \Downarrow c$, $\Omega_1 \Downarrow \Delta_1$, $\Delta, s \vdash c_1 \Rightarrow (\Delta, c[\Delta_1]_{rec} \cdot s)$ por hipótesis de inducción (tal que $(\mu.\lambda.d)[\Omega_1] \Downarrow c[\Delta_1]_{rec}$).

b) $\Delta, c[\Delta_1]_{rec} \cdot s \vdash c_2 \Rightarrow (\Delta, v_{m_2} \cdot c[\Delta_1]_{rec} \cdot s)$ por hipótesis de inducción (tal que $v_2 \Downarrow v_{m_2}$).

c) $\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta, v_{m_2} \cdot c[\Delta_1]_{rec} \cdot s)$ por Lema 1 sobre XI.a y XI.b.

d) $v_{m_2}{\cdot}c[\Delta_1]_{rec}{\cdot}\Delta_1, s \vdash c \Rightarrow (v_{m_2}{\cdot}c[\Delta_1]_{rec}{\cdot}\Delta_1, v_m{\cdot}s)$ por hipótesis de inducción (tal que $v \Downarrow v_m$).

e) $\Delta, v_{m_2} \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Rightarrow (\Delta, v_m \cdot s)$ por definición empleando XI.*d*.

f) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Rightarrow (\Delta, v_m \cdot s)$ por Lema 1 sobre XI.*c* y XI.*e*. □

Teorema 4 (Corrección en el caso de no terminación)**.** *Sea* Γ *un entorno,* Ω *un entorno sin nombres, e una expresión, d una expresión sin nombres. Si*

$$\Gamma \vdash e \mathrel{\widehat{\Rightarrow}}, \quad \Gamma \Downarrow \Omega, \quad \prod_1(\Gamma) \vdash e \Downarrow d$$

entonces

$$\Omega \vdash d \mathrel{\widehat{\Rightarrow}}$$

Demostración. Por coinducción.

1. $e = e_1 \star e_2$.

   Hipótesis $\Gamma \vdash e_1 \mathrel{\widehat{\Rightarrow}}, \quad \Gamma \vdash e_1 \star e_2 \mathrel{\widehat{\Rightarrow}}, \quad \Gamma \Downarrow \Omega, \quad \prod_1(\Gamma) \vdash e_1 \star e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1 \star e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1 \star d_2$. Por demostrar $\Omega \vdash d_1 \star d_2 \mathrel{\widehat{\Rightarrow}}$.

   a) $\Omega \vdash d_1 \mathrel{\widehat{\Rightarrow}}$ por hipótesis de coinducción sobre $\Gamma \vdash e_1 \mathrel{\widehat{\Rightarrow}}$.

   b) $\Omega \vdash d_1 \star d_2 \mathrel{\widehat{\Rightarrow}}$ por definición empleando $1a$.

2. $e = e_1 \star e_2$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow n_1, \quad \Gamma \vdash e_2 \mathrel{\widehat{\Rightarrow}}, \quad \Gamma \vdash e_1 \star e_2 \mathrel{\widehat{\Rightarrow}}, \quad \Gamma \Downarrow \Omega, \quad \prod_1(\Gamma) \vdash e_1 \star e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1 \star e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1 \star d_2$. Por demostrar $\Omega \vdash d_1 \star d_2 \mathrel{\widehat{\Rightarrow}}$.

   a) $\Omega \vdash d_1 \Rightarrow n_1$ por Teorema 1 sobre $\Gamma \vdash e_1 \Rightarrow n_1$.

   b) $\Omega \vdash d_2 \mathrel{\widehat{\Rightarrow}}$ por hipótesis de coinducción sobre $\Gamma \vdash e_2 \mathrel{\widehat{\Rightarrow}}$.

   c) $\Omega \vdash d_1 \star d_2 \mathrel{\widehat{\Rightarrow}}$ por definición empleando $2a$ y $2b$.

3. $e = \texttt{let } x = e_1 \texttt{ in } e_2$.

   Hipótesis $\Gamma \vdash e_1 \mathrel{\widehat{\Rightarrow}}, \quad \Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \mathrel{\widehat{\Rightarrow}}, \quad \Gamma \Downarrow \Omega,$
   $\prod_1(\Gamma) \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Downarrow d$.
   Usando la hipótesis $\prod_1(\Gamma) \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $x \cdot \prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = \texttt{let } d_1 \texttt{ in } d_2$. Por demostrar $\Omega \vdash \texttt{let } d_1 \texttt{ in } d_2 \mathrel{\widehat{\Rightarrow}}$.

a) $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$ por hipótesis de coinducción sobre $\Gamma \vdash e_1 \overset{\infty}{\Rightarrow}$.

b) $\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \overset{\infty}{\Rightarrow}$ por definición empleando $3a$.

4. $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow v_1$, $(x, v_1) \cdot \Gamma \vdash e_2 \overset{\infty}{\Rightarrow}$, $\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \overset{\infty}{\Rightarrow}$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow d$.
   Usando la hipótesis $\prod_1(\Gamma) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $x \cdot \prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$. Por demostrar $\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \overset{\infty}{\Rightarrow}$.

   a) $\Omega \vdash d_1 \Rightarrow v_{d_1}$ por Teorema 1 sobre $\Gamma \vdash e_1 \Rightarrow v_1$ (tal que $v_1 \Downarrow v_{d_1}$).

   b) $v_{d_1} \cdot \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$ por hipótesis de coinducción sobre $(x, v_1) \cdot \Gamma \vdash e_2 \overset{\infty}{\Rightarrow}$.

   c) $\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \overset{\infty}{\Rightarrow}$ por definición empleando $4a$ y $4b$.

5. $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.

   Hipótesis $\Gamma \vdash e_1 \overset{\infty}{\Rightarrow}$, $\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \overset{\infty}{\Rightarrow}$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow d$.
   Usando la hipótesis $\prod_1(\Gamma) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $\prod_1(\Gamma) \vdash e_3 \Downarrow d_3$, $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$. Por demostrar $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}$.

   a) $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$ por hipótesis de coinducción sobre $\Gamma \vdash e_1 \overset{\infty}{\Rightarrow}$.

   b) $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}$ por definición empleando $5a$.

6. $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow \mathit{true}$, $\Gamma \vdash e_2 \overset{\infty}{\Rightarrow}$, $\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \overset{\infty}{\Rightarrow}$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow d$.
   Usando la hipótesis $\prod_1(\Gamma) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $\prod_1(\Gamma) \vdash e_3 \Downarrow d_3$, $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$. Por demostrar $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}$.

   a) $\Omega \vdash d_1 \Rightarrow \mathit{true}$ por Teorema 1 sobre $\Gamma \vdash e_1 \Rightarrow \mathit{true}$.

   b) $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$ por hipótesis de coinducción sobre $\Gamma \vdash e_2 \overset{\infty}{\Rightarrow}$.

   c) $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}$ por definición empleando $6a$ y $6b$.

7. $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow false$, $\Gamma \vdash e_3 \mathrel{\widehat{\Rightarrow}}$, $\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mathrel{\widehat{\Rightarrow}}$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow d$.
   Usando la hipótesis $\prod_1(\Gamma) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $\prod_1(\Gamma) \vdash e_3 \Downarrow d_3$,
   $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$. Por demostrar $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \mathrel{\widehat{\Rightarrow}}$.

   a) $\Omega \vdash d_1 \Rightarrow false$ por Teorema 1 sobre $\Gamma \vdash e_1 \Rightarrow false$.

   b) $\Omega \vdash d_3 \mathrel{\widehat{\Rightarrow}}$ por hipótesis de coinducción sobre $\Gamma \vdash e_3 \mathrel{\widehat{\Rightarrow}}$.

   c) $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \mathrel{\widehat{\Rightarrow}}$ por definición empleando $7a$ y $7b$.

8. $e = e_1\ e_2$.

   Hipótesis $\Gamma \vdash e_1 \mathrel{\widehat{\Rightarrow}}$, $\Gamma \vdash e_1\ e_2 \mathrel{\widehat{\Rightarrow}}$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1\ d_2$. Por demostrar $\Omega \vdash d_1\ d_2 \mathrel{\widehat{\Rightarrow}}$.

   a) $\Omega \vdash d_1 \mathrel{\widehat{\Rightarrow}}$ por hipótesis de coinducción sobre $\Gamma \vdash e_1 \mathrel{\widehat{\Rightarrow}}$.

   b) $\Omega \vdash d_1\ d_2 \mathrel{\widehat{\Rightarrow}}$ por definición empleando $8a$.

9. $e = e_1\ e_2$.

   Hipótesis $\Gamma \vdash e_1 \Rightarrow (\lambda x.e)[\Gamma_1]$, $\Gamma \vdash e_2 \mathrel{\widehat{\Rightarrow}}$, $\Gamma \vdash e_1\ e_2 \mathrel{\widehat{\Rightarrow}}$, $\Gamma \Downarrow \Omega$,
   $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1\ d_2$. Por demostrar $\Omega \vdash d_1\ d_2 \mathrel{\widehat{\Rightarrow}}$.

   a) $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$ por Teorema 1 sobre $\Gamma \vdash e_1 \Rightarrow (\lambda x.e)[\Gamma_1]$ (tal que $(\lambda x.e)[\Gamma_1] \Downarrow (\lambda.d)[\Omega_1]$).

   b) $\Omega \vdash d_2 \mathrel{\widehat{\Rightarrow}}$ por hipótesis de coinducción sobre $\Gamma \vdash e_2 \mathrel{\widehat{\Rightarrow}}$.

   c) $\Omega \vdash d_1\ d_2 \mathrel{\widehat{\Rightarrow}}$ por definición empleando $9a$ y $9b$.

10. $e = e_1\ e_2$.

    Hipótesis $\Gamma \vdash e_1 \Rightarrow (\mu f.\lambda x.e)[\Gamma_1]$, $\Gamma \vdash e_2 \mathrel{\widehat{\Rightarrow}}$, $\Gamma \vdash e_1\ e_2 \mathrel{\widehat{\Rightarrow}}$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1\ d_2$. Por demostrar $\Omega \vdash d_1\ d_2 \mathrel{\widehat{\Rightarrow}}$.

    a) $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$ por Teorema 1 sobre $\Gamma \vdash e_1 \Rightarrow (\mu f.\lambda x.e)[\Gamma_1]$ (tal que $(\mu f.\lambda x.e)[\Gamma_1] \Downarrow (\mu.\lambda.d)[\Omega_1]$).

b) $\Omega \vdash d_2 \stackrel{\infty}{\Rightarrow}$ por hipótesis de coinducción sobre $\Gamma \vdash e_2 \stackrel{\infty}{\Rightarrow}$.

c) $\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}$ por definición empleando $10a$ y $10b$.

11. $e = e_1\ e_2$.

Hipótesis $\Gamma \vdash e_1 \Rightarrow (\lambda x.e)[\Gamma_1]$, $\Gamma \vdash e_2 \Rightarrow v_2$, $(x, v_2)\cdot\Gamma_1 \vdash e \stackrel{\infty}{\Rightarrow}$, $\Gamma \vdash e_1\ e_2 \stackrel{\infty}{\Rightarrow}$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1\ d_2$. Por demostrar $\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}$.

a) $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$ por Teorema 1 sobre $\Gamma \vdash e_1 \Rightarrow (\lambda x.e)[\Gamma_1]$ (tal que $(\lambda x.e)[\Gamma_1] \Downarrow (\lambda.d)[\Omega_1]$).

b) $\Omega \vdash d_2 \Rightarrow v_{d_2}$ por Teorema 1 sobre $\Gamma \vdash e_2 \Rightarrow v_2$ (tal que $v_2 \Downarrow v_{d_2}$).

c) $v_{d_2} \cdot \Omega_1 \vdash d \stackrel{\infty}{\Rightarrow}$ por hipótesis de coinducción sobre $(x, v_2) \cdot \Gamma_1 \vdash e \stackrel{\infty}{\Rightarrow}$.

d) $\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}$ por definición empleando $11a$, $11b$ y $11c$.

12. $e = e_1\ e_2$.

Hipótesis $\Gamma \vdash e_1 \Rightarrow (\mu f.\lambda x.e)[\Gamma_1]$, $\Gamma \vdash e_2 \Rightarrow v_2$, $(x, v_2)\cdot(f, (\mu f.\lambda x.e)[\Gamma_1])\cdot\Gamma_1 \vdash e \stackrel{\infty}{\Rightarrow}$, $\Gamma \vdash e_1\ e_2 \stackrel{\infty}{\Rightarrow}$, $\Gamma \Downarrow \Omega$, $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$. Usando la hipótesis $\prod_1(\Gamma) \vdash e_1\ e_2 \Downarrow d$, por definición de $\Downarrow$ necesariamente $\prod_1(\Gamma) \vdash e_1 \Downarrow d_1$, $\prod_1(\Gamma) \vdash e_2 \Downarrow d_2$, $d = d_1\ d_2$. Por demostrar $\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}$.

a) $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$ por Teorema 1 sobre $\Gamma \vdash e_1 \Rightarrow (\mu f.\lambda x.e)[\Gamma_1]$ (tal que $(\mu f.\lambda x.e)[\Gamma_1] \Downarrow (\mu.\lambda.d)[\Omega_1]$).

b) $\Omega \vdash d_2 \Rightarrow v_{d_2}$ por Teorema 1 sobre $\Gamma \vdash e_2 \Rightarrow v_2$ (tal que $v_2 \Downarrow v_{d_2}$).

c) $v_{d_2} \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \stackrel{\infty}{\Rightarrow}$ por hipótesis de coinducción sobre $(x, v_2) \cdot (f, (\mu f.\lambda x.e)[\Gamma_1]) \cdot \Gamma_1 \vdash e \stackrel{\infty}{\Rightarrow}$.

d) $\Omega \vdash d_1\ d_2 \stackrel{\infty}{\Rightarrow}$ por definición empleando $12a$, $12b$ y $12c$. □

Lema 2. *Sea m una configuración, n un natural cualquiera,*

$$m \stackrel{\infty}{\Rightarrow} \quad \textit{si y sólo si} \quad m \stackrel{\infty}{\underset{n}{\Rightarrow}}$$

Demostración.

(I) Si $m \overset{\infty}{\Rightarrow}$ entonces $m \overset{\infty}{\underset{n}{\Rightarrow}}$. Por coinducción.

Hipótesis $m \overset{\infty}{\Rightarrow}$. Usando la hipótesis $m \overset{\infty}{\Rightarrow}$ por definición de $\overset{\infty}{\Rightarrow}$ necesariamente $m \to m_1$, $m_1 \overset{\infty}{\Rightarrow}$. Nos encontramos ahora en posición de demostrar $m \overset{\infty}{\underset{n}{\Rightarrow}}$.

a) $m_1 \overset{\infty}{\underset{n}{\Rightarrow}}$ por hipótesis de coinducción sobre $m_1 \overset{\infty}{\Rightarrow}$.

b) $m \overset{\infty}{\underset{n}{\Rightarrow}}$ por la regla $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform sobre $m \to m_1$ y $m_1 \overset{\infty}{\underset{n}{\Rightarrow}}$.

(II) Si $m \overset{\infty}{\underset{n}{\Rightarrow}}$ entonces $m \overset{\infty}{\Rightarrow}$. Por coinducción.

Hipótesis $m \overset{\infty}{\underset{n}{\Rightarrow}}$. Por demostrar $m \overset{\infty}{\Rightarrow}$.

a) La demostración usa la proposición que enuncia que si $m \overset{\infty}{\underset{n}{\Rightarrow}}$ entonces existe m_1 tal que $m \to m_1$ y $m_1 \overset{\infty}{\underset{n_1}{\Rightarrow}}$, la cual se demuestra por inducción sobre el natural n.

b) $m \to m_1$ y $m_1 \overset{\infty}{\underset{n_1}{\Rightarrow}}$ por II.*a* sobre $m \overset{\infty}{\underset{n}{\Rightarrow}}$.

c) $m_1 \overset{\infty}{\Rightarrow}$ por hipótesis de coinducción sobre $m_1 \overset{\infty}{\underset{n_1}{\Rightarrow}}$.

d) $m \overset{\infty}{\Rightarrow}$ por definición de $\overset{\infty}{\Rightarrow}$ sobre $m \to m_1$ y $m_1 \overset{\infty}{\Rightarrow}$. □

Lema 3. *Si* $\Omega \vdash d \overset{\infty}{\Rightarrow}$, *entonces* $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d\|}{\Rightarrow}}$ *para todo código* c *y pila* s.

Demostración. Por coinducción.

(I) $d = d_1 \star d_2$.

Hipótesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$. Por demostrar $(\llbracket d_1 \star d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1 \star d_2\|}{\Rightarrow}}$.

a) para todo c_h, s_h, $(\llbracket d_1 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

b) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ por I.*a* con $c_h = \llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c$ y $s_h = s$.

c) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|+1}{\Rightarrow}}$ por la regla $\overset{\infty}{\underset{n}{\Rightarrow}}$-sleep sobre I.*b*.

d) $(\llbracket d_1 \star d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1 \star d_2\|}{\Rightarrow}}$ por definición $\llbracket d_1 \star d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp}$ y $\|d_1 \star d_2\| = \|d_1\| + 1$.

(II) $d = d_1 \star d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow n_1, \quad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$. Por demostrar $(\llbracket d_1 \star d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \star d_2 \rVert}{\overset{\infty}{\Rightarrow}}$.

a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, \llbracket n_1 \rrbracket \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_1 \Rightarrow n_1$.

b) para todo c_h, s_h, $(\llbracket d_2 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \underset{\lVert d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por hipótesis de inducción sobre $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

c) $(\llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, \llbracket n_1 \rrbracket \cdot s) \underset{\lVert d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por II.*b* con $c_h = \mathsf{IOp} \cdot c$ y $s_h = \llbracket n_1 \rrbracket \cdot s$.

d) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \star d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-perform sobre II.*a* y II.*c*.

e) $(\llbracket d_1 \star d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \star d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por definición $\llbracket d_1 \star d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IOp}$.

(III) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Hipótesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$. Por demostrar $(\llbracket \texttt{let}\ d_1\ \texttt{in}\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert \texttt{let}\ d_1\ \texttt{in}\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$.

a) para todo c_h, s_h, $(\llbracket d_1 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \underset{\lVert d_1 \rVert}{\overset{\infty}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

b) $(\llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \rVert}{\overset{\infty}{\Rightarrow}}$ por III.*a* con $c_h = \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c$ y $s_h = s$.

c) $(\llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \rVert + 1}{\overset{\infty}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-sleep sobre III.*b*.

d) $(\llbracket \texttt{let}\ d_1\ \texttt{in}\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert \texttt{let}\ d_1\ \texttt{in}\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por definición $\llbracket \texttt{let}\ d_1\ \texttt{in}\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet}$ y $\lVert \texttt{let}\ d_1\ \texttt{in}\ d_2 \rVert = \lVert d_1 \rVert + 1$.

(IV) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow v_1, \quad v_1 \cdot \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

Por demostrar $(\llbracket \texttt{let}\ d_1\ \texttt{in}\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert \texttt{let}\ d_1\ \texttt{in}\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$.

a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to} (\mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_1 \rrbracket \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_1 \Rightarrow v_1$.

b) $(\mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_1 \rrbracket \cdot s) \to (\llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket v_1 \rrbracket \cdot \llbracket \Omega \rrbracket, s)$ por definición de $\to$ de la instrucción ILet.

c) para todo c_h, s_h, $(\llbracket d_2 \rrbracket \cdot c_h, \llbracket v_1 \rrbracket \cdot \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\|d_2\|}{\Rightarrow}}$ por hipótesis de coinducción sobre $v_1 \cdot \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

d) $(\llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket v_1 \rrbracket \cdot \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_2\|}{\Rightarrow}}$ por IV.*c* con $c_h = \mathsf{IEndLet} \cdot c$ y $s_h = s$.

e) $(\mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_1 \rrbracket \cdot s) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-perform sobre IV.*b* y IV.*d*.

f) $(\llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|+1}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-perform sobre IV.*a* y IV.*e*.

g) $(\llbracket \texttt{let } d_1 \texttt{ in } d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|\texttt{let } d_1 \texttt{ in } d_2\|}{\Rightarrow}}$ por definición $\llbracket \texttt{let } d_1 \texttt{ in } d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet}$ y $\|\texttt{let } d_1 \texttt{ in } d_2\| = \|d_1\| + 1$.

(V) $d = \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3$.

Hipótesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

Por demostrar $(\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\Rightarrow}}$.

a) para todo c_h, s_h, $(\llbracket d_1 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

b) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ por V.*a* con $c_h = \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c$ y $s_h = s$.

c) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|+1}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-sleep sobre V.*b*.

d) $(\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\Rightarrow}}$ por definición $\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$ y $\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\| = \|d_1\| + 1$.

(VI) $d = \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3$.

Hipótesis $\Omega \vdash d_1 \Rightarrow true$, $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

Por demostrar $(\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\Rightarrow}}$.

a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow}$ $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, true \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_1 \Rightarrow true$.

b) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, true \cdot s) \to (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s)$ por definición de $\to$ de la instrucción ISel.

c) para todo c_h, s_h, $(\llbracket d_2 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\lVert d_2 \rVert}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

d) $(\llbracket d_2 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s) \overset{\infty}{\underset{\lVert d_2 \rVert}{\Rightarrow}}$ por VI.*c* con $c_h = \mathsf{IJoin}$ y $s_h = (c, [\,]) \cdot s$.

e) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, true \cdot s) \overset{\infty}{\underset{\lVert d_1 \rVert}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-perform sobre VI.*b* y VI.*d*.

f) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\lVert d_1 \rVert + 1}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-perform sobre VI.*a* y VI.*e*.

g) $(\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\lVert \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rVert}{\Rightarrow}}$ por definición $\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$ y $\lVert \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rVert = \lVert d_1 \rVert + 1$.

(VII) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hipótesis $\Omega \vdash d_1 \Rightarrow false, \quad \Omega \vdash d_3 \overset{\infty}{\Rightarrow}$.

Por demostrar $(\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\lVert \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rVert}{\Rightarrow}}$.

a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to}$ $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, false \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_1 \Rightarrow false$.

b) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, false \cdot s) \to (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s)$ por definición de $\to$ de la instrucción ISel.

c) para todo c_h, s_h, $(\llbracket d_3 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\lVert d_3 \rVert}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_3 \overset{\infty}{\Rightarrow}$.

d) $(\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s) \overset{\infty}{\underset{\lVert d_3 \rVert}{\Rightarrow}}$ por VII.*c* con $c_h = \mathsf{IJoin}$ y $s_h = (c, [\,]) \cdot s$.

e) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, false \cdot s) \overset{\infty}{\underset{\lVert d_1 \rVert}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-perform sobre VII.*b* y VII.*d*.

f) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\lVert d_1 \rVert + 1}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}_n$-perform sobre VII.*a* y VII.*e*.

g) $(\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rVert}{\overset{\infty}{\Rightarrow}}$ por definición $\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$ y $\lVert \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rVert = \lVert d_1 \rVert + 1$.

(VIII) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$. Por demostrar $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$.

a) para todo c_h, s_h, $(\llbracket d_1 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \underset{\lVert d_1 \rVert}{\overset{\infty}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

b) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \rVert}{\overset{\infty}{\Rightarrow}}$ por VIII.*a* con $c_h = \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c$ y $s_h = s$.

c) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \rVert + 1}{\overset{\infty}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}$-sleep sobre VIII.*b*.

d) $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por definición $\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$ y $\lVert d_1\ d_2 \rVert = \lVert d_1 \rVert + 1$.

(IX) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$. Por demostrar $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$.

a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$.

b) para todo c_h, s_h, $(\llbracket d_2 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \underset{\lVert d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

c) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s) \underset{\lVert d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por IX.*b* con $c_h = \mathsf{IApp} \cdot c$ y $s_h = \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s$.

d) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \rVert + 1}{\overset{\infty}{\Rightarrow}}$ por la regla $\overset{\infty}{\Rightarrow}$-perform sobre IX.*a* y IX.*c*.

e) $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por definición $\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$ y $\lVert d_1\ d_2 \rVert = \lVert d_1 \rVert + 1$.

(X) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$. Por demostrar $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$.

a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$.

b) para todo c_h, s_h, $(\llbracket d_2 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\|d_2\|}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

c) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s) \overset{\infty}{\underset{\|d_2\|}{\Rightarrow}}$ por X.*b* con $c_h = \mathsf{IApp} \cdot c$ y $s_h = \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s$.

d) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|+1}{\Rightarrow}}$ por la regla $\overset{\infty}{n}$-perform sobre X.*a* y X.*c*.

e) $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$ por definición $\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$ y $\|d_1\ d_2\| = \|d_1\| + 1$.

(XI) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}$.
Por demostrar $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$.

a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$.

b) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s) \overset{+}{\to} (\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_2 \Rightarrow v_2$.

c) $(\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket] \cdot s) \to (\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s)$ por definición de $\to$ de la instrucción IApp.

d) para todo c_h, s_h, $(\llbracket d \rrbracket \cdot c_h, \llbracket v_2 \rrbracket \cdot \llbracket \Omega_1 \rrbracket, s_h) \overset{\infty}{\underset{\|d\|}{\Rightarrow}}$ por hipótesis de coinducción sobre $v_2 \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}$.

e) $(\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s) \overset{\infty}{\underset{\|d\|}{\Rightarrow}}$ por XI.*d* con $c_h = \mathsf{IRet}$ y $s_h = (c, \llbracket \Omega \rrbracket) \cdot s$.

f) $(\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket] \cdot s) \overset{\infty}{\underset{\|d_2\|}{\Rightarrow}}$ por la regla $\overset{\infty}{n}$-perform sobre XI.*c* y XI.*e*.

g) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ por la regla $\overset{\infty}{n}$-perform sobre XI.*b* y XI.*f*, y por definición $\llbracket (\lambda.d)[\Omega_1] \rrbracket = (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]$.

h) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|+1}{\Rightarrow}}$ por la regla $\overset{\infty}{n}$-perform sobre XI.*a* y XI.*g*.

i) $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$ por definición $\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$ y $\|d_1\ d_2\| = \|d_1\| + 1$.

(XII) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}$.
Por demostrar $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$.

a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$.

b) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s) \overset{+}{\to} (\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s)$ por Teorema 2 sobre $\Omega \vdash d_2 \Rightarrow v_2$.

c) $(\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot s) \to$
$(\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s)$ por definición de $\to$ de la instrucción IApp.

d) para todo c_h, s_h, $(\llbracket d \rrbracket \cdot c_h, \llbracket v_2 \rrbracket \cdot \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot \llbracket \Omega_1 \rrbracket, s_h) \underset{\lVert d \rVert}{\overset{\infty}{\Rightarrow}}$ por hipótesis de coinducción sobre $v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}$.

e) $(\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s) \underset{\lVert d \rVert}{\overset{\infty}{\Rightarrow}}$ por XII.d con $c_h = \mathsf{IRet}$ y $s_h = (c, \llbracket \Omega \rrbracket) \cdot s$, y por definición $\llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket = (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec}$.

f) $(\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot s) \underset{\lVert d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por la regla $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform sobre XII.$c$ y XII.e.

g) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s) \underset{\lVert d_1 \rVert}{\overset{\infty}{\Rightarrow}}$ por la regla $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform sobre XII.$b$ y XII.f, y por definición $\llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket = (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec}$.

h) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1 \rVert + 1}{\overset{\infty}{\Rightarrow}}$ por la regla $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform sobre XII.$a$ y XII.g.

i) $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d_1\ d_2 \rVert}{\overset{\infty}{\Rightarrow}}$ por definición $\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$ y $\lVert d_1\ d_2 \rVert = \lVert d_1 \rVert + 1$. □

Teorema 5. *Si* $\Omega \vdash d \overset{\infty}{\Rightarrow}$, *entonces* $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\to}$ *para todo código* c *y pila* s.

Demostración. Hipótesis $\Omega \vdash d \overset{\infty}{\Rightarrow}$. Por demostrar $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\to}$.

(I) $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\lVert d \rVert}{\overset{\infty}{\Rightarrow}}$ por Lema 3 sobre $\Omega \vdash d \overset{\infty}{\Rightarrow}$.

(II) $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\to}$ por Lema 2 sobre I. □

Lema 4. *Sea Δ un entorno de máquina, s una pila, i una instrucción,*

$$\Delta, s \vdash i \Rrightarrow \qquad \textit{si y sólo si} \qquad \Delta, s \vdash i \Mapsto$$

y, sea c un código, n un natural cualquiera,

$$\Delta, s \vdash c \Rrightarrow \qquad \textit{si y sólo si} \qquad \Delta, s \vdash c \overset{n}{\Mapsto}$$

Demostración.

1. Si $\Delta, s \vdash i \Rrightarrow$ entonces $\Delta, s \vdash i \Mapsto$ y si $\Delta, s \vdash c \Rrightarrow$ entonces $\Delta, s \vdash c \overset{n}{\Mapsto}$.

   a) Si $\Delta, s \vdash i \Rrightarrow$ entonces $\Delta, s \vdash i \Mapsto$. Suponiendo 1b, la demostración es por análisis de casos.

      (I) $i = \mathsf{ISel}\ c_1\ c_2$.
      Hipótesis $\Delta, s \vdash c_1 \Rrightarrow$, $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Rrightarrow$.
      Por demostrar $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Mapsto$.
      A. $\Delta, s \vdash c_1 \overset{n}{\Mapsto}$ por 1b sobre $\Delta, s \vdash c_1 \Rrightarrow$.

      B. $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Mapsto$ por definición empleando 1aIA.

      (II) $i = \mathsf{ISel}\ c_1\ c_2$.
      Hipótesis $\Delta, s \vdash c_1 \Rrightarrow$, $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Rrightarrow$.
      Por demostrar $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Mapsto$.
      A. $\Delta, s \vdash c_1 \overset{n}{\Mapsto}$ por 1b sobre $\Delta, s \vdash c_1 \Rrightarrow$.

      B. $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Mapsto$ por definición empleando 1aIIA.

      (III) $i = \mathsf{IApp}$.
      Hipótesis $v \cdot \Delta_1, s \vdash c \Rrightarrow$, $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Rrightarrow$.
      Por demostrar $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Mapsto$.
      A. $v \cdot \Delta_1, s \vdash c \overset{n}{\Mapsto}$ por 1b sobre $v \cdot \Delta_1, s \vdash c \Rrightarrow$.

      B. $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Mapsto$ por definición empleando 1aIIIA.

      (IV) $i = \mathsf{IApp}$.
      Hipótesis $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Rrightarrow$, $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Rrightarrow$.
      Por demostrar $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Mapsto$.
      A. $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \overset{n}{\Mapsto}$ por 1b sobre $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Rrightarrow$.

      B. $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Mapsto$ por definición empleando 1aIVA.

b) Si $\Delta, s \vdash c \Subset$ entonces $\Delta, s \vdash c \Subset_n$. Por coinducción.

(I) $c = i \cdot c$'.
Hipótesis $\Delta, s \vdash i \Subset$, $\Delta, s \vdash i \cdot c' \Subset$. Por demostrar $\Delta, s \vdash i \cdot c' \Subset_n$.

A. $\Delta, s \vdash i \Subset_n$ por 1a sobre $\Delta, s \vdash i \Subset$.

B. $\Delta, s \vdash i \cdot c' \Subset_n$ por definición empleando 1bIA.

(II) $c = i \cdot c$'.
Hipótesis $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$, $\Delta_1, s_1 \vdash c' \Subset$, $\Delta, s \vdash i \cdot c' \Subset$.
Por demostrar $\Delta, s \vdash i \cdot c' \Subset_n$.

A. $\Delta_1, s_1 \vdash c' \Subset_n$ por hipótesis de coinducción sobre $\Delta_1, s_1 \vdash c' \Subset$.

B. $\Delta, s \vdash i \cdot c' \Subset_n$ por la regla $\Subset_n$-perform sobre $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ y 1bIIA.

2. Si $\Delta, s \vdash i \Subset_n$ entonces $\Delta, s \vdash i \Subset$ y si $\Delta, s \vdash c \Subset_n$ entonces $\Delta, s \vdash c \Subset$.

a) Si $\Delta, s \vdash i \Subset_n$ entonces $\Delta, s \vdash i \Subset$. Suponiendo 2b, la demostración es por análisis de casos.

(I) $i = \mathsf{ISel}\ c_1\ c_2$.
Hipótesis $\Delta, s \vdash c_1 \Subset_n$, $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Subset$.
Por demostrar $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Subset$.
A. $\Delta, s \vdash c_1 \Subset$ por 2b sobre $\Delta, s \vdash c_1 \Subset_n$.

B. $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Subset$ por definición empleando 2aIA.

(II) $i = \mathsf{ISel}\ c_1\ c_2$.
Hipótesis $\Delta, s \vdash c_1 \Subset_n$, $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Subset$.
Por demostrar $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Subset$.
A. $\Delta, s \vdash c_1 \Subset$ por 2b sobre $\Delta, s \vdash c_1 \Subset_n$.

B. $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Subset$ por definición empleando 2aIIA.

(III) $i = \mathsf{IApp}$.
Hipótesis $v \cdot \Delta_1, s \vdash c \Subset_n$, $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Subset$.
Por demostrar $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Subset$.
A. $v \cdot \Delta_1, s \vdash c \Subset$ por 2b sobre $v \cdot \Delta_1, s \vdash c \Subset_n$.

B. $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Subset$ por definición empleando 2aIIIA.

(IV) $i = \mathsf{IApp}$.
Hipótesis $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Downarrow_n$, $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Downarrow$.
Por demostrar $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Uparrow$.
A. $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Uparrow$ por 2b sobre $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Downarrow_n$.

B. $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Uparrow$ por definición empleando 2aIVA.

b) Si $\Delta, s \vdash c \Downarrow_n$ entonces $\Delta, s \vdash c \Uparrow$. Por coinducción.

(I) $c = i \cdot c$.
Hipótesis $\Delta, s \vdash i \Downarrow$, $\Delta, s \vdash i \cdot c \Downarrow_n$. Por demostrar $\Delta, s \vdash i \cdot c \Uparrow$.
A. $\Delta, s \vdash i \Uparrow$ por 2a sobre $\Delta, s \vdash i \Downarrow$.

B. $\Delta, s \vdash i \cdot c \Uparrow$ por definición empleando 2bIA.

(II) $c = c_1 \cdot c_2$.
Hipótesis $\Delta, s \vdash c_1 \Downarrow_n$, $\Delta, s \vdash c_1 \cdot c_2 \Downarrow_{n+1}$.
Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \Uparrow$.
A. La demostración usa la proposición que enuncia que si $\Delta, s \vdash i \cdot c \Downarrow_n$ entonces $\Delta, s \vdash i \Downarrow$ o existen n_1, Δ_1, s_1, tal que $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ y $\Delta_1, s_1 \vdash c \Downarrow_{n_1}$, dicha proposición se demuestra por inducción sobre n.

B. $c_1 \neq []$ es decir $c_1 = i \cdot c_1'$ por definición de $\Downarrow_n$ y $\Delta, s \vdash c_1 \Downarrow_n$.

C. $\Delta, s \vdash i \Downarrow$, o $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ y $\Delta_1, s_1 \vdash c_1' \cdot c_2 \Downarrow_{n_1}$ por 2bIIA sobre $\Delta, s \vdash i \cdot c_1' \cdot c_2 \Downarrow_{n+1}$.

- $\Delta, s \vdash i \Downarrow$.
  - $\Delta, s \vdash i \Uparrow$ por 2a sobre $\Delta, s \vdash i \Downarrow$.
  - $\Delta, s \vdash i \cdot c_1' \cdot c_2 \Uparrow$ por definición empleando $\Delta, s \vdash i \Uparrow$.
- $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ y $\Delta_1, s_1 \vdash c_1' \cdot c_2 \Downarrow_{n_1}$.
  - $\Delta_1, s_1 \vdash c_1' \cdot c_2 \Uparrow$ por hipótesis de coinducción sobre $\Delta_1, s_1 \vdash c_1' \cdot c_2 \Downarrow_{n_1}$.
  - $\Delta, s \vdash i \cdot c_1' \cdot c_2 \Uparrow$ por definición empleando $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ y $\Delta_1, s_1 \vdash c_1' \cdot c_2 \Uparrow$.

(III) $c = c_1 \cdot c_2$.
Hipótesis $c_1 \neq []$, $\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)$, $\Delta_1, s_1 \vdash c_2 \Downarrow_{n'}$,
$\Delta, s \vdash c_1 \cdot c_2 \Downarrow_n$. Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \Uparrow$.

A. $c_1 \neq [\,]$ es decir $c_1 = i \cdot c_1'$ por hipótesis.

B. $\Delta, s \vdash i \Rightarrow (\Delta_t, s_t)$ y $\Delta_t, s_t \vdash c_1' \Rightarrow (\Delta_1, s_1)$ por la hipótesis $\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)$.

- $c_1' = [\,]$.
  - $\Delta_t = \Delta_1$ y $s_t = s_1$ por $\Delta_t, s_t \vdash c_1' \Rightarrow (\Delta_1, s_1)$ y $c_1' = [\,]$.
  - $\Delta_1, s_1 \vdash c_2 \overset{\cong}{\Rightarrow}$ por hipótesis de coinducción sobre $\Delta_1, s_1 \vdash c_2 \overset{\cong}{\underset{n'}{\Rightarrow}}$.
  - $\Delta, s \vdash i \cdot c_2 \overset{\cong}{\Rightarrow}$ por definición empleando $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ y $\Delta_1, s_1 \vdash c_2 \overset{\cong}{\Rightarrow}$.
  - $\Delta, s \vdash i \cdot c_1' \cdot c_2 \overset{\cong}{\Rightarrow}$ por $c_1' = [\,]$.
- $c_1' \neq [\,]$
  - $\Delta_t, s_t \vdash c_1' \cdot c_2 \overset{\cong}{\underset{n'}{\Rightarrow}}$ por la regla $\overset{\cong}{\underset{n}{\Rightarrow}}$-perform sobre $\Delta_t, s_t \vdash c_1' \Rightarrow (\Delta_1, s_1)$ y $\Delta_1, s_1 \vdash c_2 \overset{\cong}{\underset{n'}{\Rightarrow}}$.
  - $\Delta_t, s_t \vdash c_1' \cdot c_2 \overset{\cong}{\Rightarrow}$ por hipótesis de coinducción sobre $\Delta_t, s_t \vdash c_1' \cdot c_2 \overset{\cong}{\underset{n'}{\Rightarrow}}$.
  - $\Delta, s \vdash i \cdot c_1' \cdot c_2 \overset{\cong}{\Rightarrow}$ por definición empleando $\Delta, s \vdash i \Rightarrow (\Delta_t, s_t)$ y $\Delta_t, s_t \vdash c_1' \cdot c_2 \overset{\cong}{\Rightarrow}$. □

Lema 5 (Corrección en el caso de no terminación (auxiliar))**.** *Sea* Ω *un entorno sin nombres,* Δ *un entorno de máquina,* d *una expresión sin nombres,* c *un código de máquina. Si*

$$\Omega \vdash d \overset{\cong}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \Downarrow \Delta$$

entonces, para toda pila s*,*

$$\Delta, s \vdash c \overset{\cong}{\underset{\|d\|}{\Rightarrow}}$$

Demostración. Por coinducción.

(I) $d = d_1 \star d_2$.

Hipótesis $\Omega \vdash d_1 \overset{\cong}{\Rightarrow}, \quad \Omega \vdash d_1 \star d_2 \overset{\cong}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \Downarrow \Delta$. Usando la hipótesis $d_1 \star d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IOp}$. Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \overset{\cong}{\underset{\|d_1 \star d_2\|}{\Rightarrow}}$.

a) $\Delta, s \vdash c_1 \overset{\cong}{\underset{\|d_1\|}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_1 \overset{\cong}{\Rightarrow}$.

b) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \mathrel{\Mapsto}_{\|d_1\|+1}$ por la regla $\mathrel{\Mapsto}_n$-sleep sobre I.*a*.

c) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \mathrel{\Mapsto}_{\|d_1 \star d_2\|}$ por definición $\|d_1 \star d_2\| = \|d_1\| + 1$.

(II) $d = d_1 \star d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow n_1$, $\Omega \vdash d_2 \mathrel{\Mapsto}$, $\Omega \vdash d_1 \star d_2 \mathrel{\Mapsto}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $d_1 \star d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IOp}$. Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \mathrel{\Mapsto}_{\|d_1 \star d_2\|}$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, n_1 \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_1 \Rightarrow n_1$.

b) $\Delta, n_1 \cdot s \vdash c_2 \mathrel{\Mapsto}_{\|d_2\|}$ por hipótesis de coinducción sobre $\Omega \vdash d_2 \mathrel{\Mapsto}$.

c) $\Delta, n_1 \cdot s \vdash c_2 \cdot \mathsf{IOp} \mathrel{\Mapsto}_{\|d_2\|+1}$ por la regla $\mathrel{\Mapsto}_n$-sleep sobre II.*b*.

d) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \mathrel{\Mapsto}_{\|d_1\|+1}$ por la regla $\mathrel{\Mapsto}_n$-perform sobre II.*a* y II.*c*.

e) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \mathrel{\Mapsto}_{\|d_1 \star d_2\|}$ por definición $\|d_1 \star d_2\| = \|d_1\| + 1$.

(III) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Hipótesis $\Omega \vdash d_1 \mathrel{\Mapsto}$, $\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \mathrel{\Mapsto}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet}$. Por demostrar $\Omega \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \mathrel{\Mapsto}_{\|\texttt{let}\ d_1\ \texttt{in}\ d_2\|}$.

a) $\Delta, s \vdash c_1 \mathrel{\Mapsto}_{\|d_1\|}$ por hipótesis de coinducción sobre $\Omega \vdash d_1 \mathrel{\Mapsto}$.

b) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \mathrel{\Mapsto}_{\|d_1\|+1}$ por la regla $\mathrel{\Mapsto}_n$-sleep sobre III.*a*.

c) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \mathrel{\Mapsto}_{\|\texttt{let}\ d_1\ \texttt{in}\ d_2\|}$ por definición $\|\texttt{let}\ d_1\ \texttt{in}\ d_2\| = \|d_1\| + 1$.

(IV) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow v_1$, $v_1 \cdot \Omega \vdash d_2 \mathrel{\Mapsto}$, $\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \mathrel{\Mapsto}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet}$. Por demostrar $\Omega \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \mathrel{\Mapsto}_{\|\texttt{let}\ d_1\ \texttt{in}\ d_2\|}$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, v_{m_1} \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_1 \Rightarrow v_1$ (tal que $v_1 \Downarrow v_{m_1}$).

b) $\Delta, v_{m_1} \cdot s \vdash \mathsf{ILet} \Rightarrow (v_{m_1} \cdot \Delta, s)$ por definición.

c) $v_{m_1} \cdot \Delta, s \vdash c_2 \underset{\|d_2\|}{\overset{\cong}{\Rightarrow}}$ por hipótesis de coinducción sobre $v_1 \cdot \Omega \vdash d_2 \overset{\cong}{\Rightarrow}$.

d) $v_{m_1} \cdot \Delta, s \vdash c_2 \cdot \mathsf{IEndLet} \underset{\|d_2\|+1}{\overset{\cong}{\Rightarrow}}$ por la regla $\overset{\cong}{\Rightarrow}_n$-sleep sobre IV.*c*.

e) $\Delta, v_{m_1} \cdot s \vdash \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \underset{\|d_2\|+1}{\overset{\cong}{\Rightarrow}}$ por la regla $\overset{\cong}{\Rightarrow}_n$-perform sobre IV.*b* y IV.*d*.

f) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \underset{\|d_1\|+1}{\overset{\cong}{\Rightarrow}}$ por la regla $\overset{\cong}{\Rightarrow}_n$-perform sobre IV.*a* y IV.*e*.

g) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \underset{\|\texttt{let } d_1 \texttt{ in } d_2\|}{\overset{\cong}{\Rightarrow}}$ por definición $\|\texttt{let } d_1 \texttt{ in } d_2\| = \|d_1\|+1$.

(V) $d = \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3$.

Hipótesis $\Omega \vdash d_1 \overset{\cong}{\Rightarrow}$, $\Omega \vdash \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \overset{\cong}{\Rightarrow}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $d_3 \Downarrow c_3$, $c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$.
Por demostrar $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\overset{\cong}{\Rightarrow}}$.

a) $\Delta, s \vdash c_1 \underset{\|d_1\|}{\overset{\cong}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_1 \overset{\cong}{\Rightarrow}$.

b) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|d_1\|+1}{\overset{\cong}{\Rightarrow}}$ por la regla $\overset{\cong}{\Rightarrow}_n$-sleep sobre V.*a*.

c) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\overset{\cong}{\Rightarrow}}$
por definición $\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\| = \|d_1\| + 1$.

(VI) $d = \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3$.

Hipótesis $\Omega \vdash d_1 \Rightarrow \mathit{true}$, $\Omega \vdash d_2 \overset{\cong}{\Rightarrow}$, $\Omega \vdash \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \overset{\cong}{\Rightarrow}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $d_3 \Downarrow c_3$, $c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. Por demostrar $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\overset{\cong}{\Rightarrow}}$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, \mathit{true} \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_1 \Rightarrow \mathit{true}$.

b) $\Delta, s \vdash c_2 \underset{\|d_2\|}{\overset{\cong}{\Rightarrow}}$ por hipótesis de coinducción sobre $\Omega \vdash d_2 \overset{\cong}{\Rightarrow}$.

c) $\Delta, \mathit{true} \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \Rightarrow \overset{\cong}{\Rightarrow}$ por definición empleando VI.*b*.

d) $\Delta, \mathit{true} \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \underset{\|d_2\|}{\overset{\cong}{\Rightarrow}}$ por definición empleando VI.*c*.

e) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|d_1\|+1}{\overset{\cong}{\Rightarrow}}$ por la regla $\overset{\cong}{\Rightarrow}_n$-perform sobre VI.*a* y VI.*d*.

f) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3\ {}_{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}\!\!\Mapsto$
por definición $\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\| = \|d_1\| + 1$.

(VII) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hipótesis $\Omega \vdash d_1 \Rightarrow false$, $\Omega \vdash d_3 \mathrel{\overset{\cong}{\Rightarrow}}$, $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \mathrel{\overset{\cong}{\Rightarrow}}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $d_3 \Downarrow c_3$, $c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. Por demostrar $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3\ {}_{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}\!\!\Mapsto$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, false \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_1 \Rightarrow false$.

b) $\Delta, s \vdash c_3\ {}_{\|d_3\|}\!\!\Mapsto$ por hipótesis de coinducción sobre $\Omega \vdash d_3 \mathrel{\overset{\cong}{\Rightarrow}}$.

c) $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \Rightarrow \Mapsto$ por definición empleando VII.*b*.

d) $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_2\ c_3\ {}_{\|d_3\|}\!\!\Mapsto$ por definición empleando VII.*c*.

e) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3\ {}_{\|d_1\|+1}\!\!\Mapsto$ por la regla $\Mapsto_n$-perform sobre VII.*a* y VII.*d*.

f) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3\ {}_{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}\!\!\Mapsto$
por definición $\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\| = \|d_1\| + 1$.

(VIII) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \mathrel{\overset{\cong}{\Rightarrow}}$, $\Omega \vdash d_1\ d_2 \mathrel{\overset{\cong}{\Rightarrow}}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $d_1\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IApp}$. Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp}\ {}_{\|d_1\ d_2\|}\!\!\Mapsto$.

a) $\Delta, s \vdash c_1\ {}_{\|d_1\|}\!\!\Mapsto$ por hipótesis de coinducción sobre $\Omega \vdash d_1 \mathrel{\overset{\cong}{\Rightarrow}}$.

b) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp}\ {}_{\|d_1\|+1}\!\!\Mapsto$ por la regla $\Mapsto_n$-sleep sobre VIII.*a*.

c) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp}\ {}_{\|d_1\ d_2\|}\!\!\Mapsto$ por definición $\|d_1\ d_2\| = \|d_1\| + 1$.

(IX) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\lambda.d')[\Omega_1]$, $\Omega \vdash d_2 \mathrel{\overset{\cong}{\Rightarrow}}$, $\Omega \vdash d_1\ d_2 \mathrel{\overset{\cong}{\Rightarrow}}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $d_1\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IApp}$. Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp}\ {}_{\|d_1\ d_2\|}\!\!\Mapsto$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, v_m \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_1 \Rightarrow (\lambda.d')[\Omega_1]$ (tal que $(\lambda.d')[\Omega_1] \Downarrow v_m$).

b) $\Delta, v_m \cdot s \vdash c_2 \Subset_{\|d_2\|}$ por hipótesis de coinducción sobre $\Omega \vdash d_2 \mathrel{\widehat{\Rightarrow}}$.

c) $\Delta, v_m \cdot s \vdash c_2 \cdot \mathsf{IApp} \Subset_{\|d_2\|+1}$ por la regla $\mathrel{\widehat{\Rightarrow}}_n$-sleep sobre IX.*b*.

d) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Subset_{\|d_1\|+1}$ por la regla $\mathrel{\widehat{\Rightarrow}}_n$-perform sobre IX.*a* y IX.*c*.

e) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Subset_{\|d_1\ d_2\|}$ por definición $\|d_1\ d_2\| = \|d_1\| + 1$.

(X) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d')[\Omega_1]$, $\Omega \vdash d_2 \mathrel{\widehat{\Rightarrow}}$, $\Omega \vdash d_1\ d_2 \mathrel{\widehat{\Rightarrow}}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $d_1\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IApp}$. Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Subset_{\|d_1\ d_2\|}$.

a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, v_m \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d')[\Omega_1]$ (tal que $(\mu.\lambda.d')[\Omega_1] \Downarrow v_m$).

b) $\Delta, v_m \cdot s \vdash c_2 \Subset_{\|d_2\|}$ por hipótesis de coinducción sobre $\Omega \vdash d_2 \mathrel{\widehat{\Rightarrow}}$.

c) $\Delta, v_m \cdot s \vdash c_2 \cdot \mathsf{IApp} \Subset_{\|d_2\|+1}$ por la regla $\mathrel{\widehat{\Rightarrow}}_n$-sleep sobre X.*b*.

d) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Subset_{\|d_1\|+1}$ por la regla $\mathrel{\widehat{\Rightarrow}}_n$-perform sobre X.*a* y X.*c*.

e) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Subset_{\|d_1\ d_2\|}$ por definición $\|d_1\ d_2\| = \|d_1\| + 1$.

(XI) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\lambda.d')[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot \Omega_1 \vdash d' \mathrel{\widehat{\Rightarrow}}$, $\Omega \vdash d_1\ d_2 \mathrel{\widehat{\Rightarrow}}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $d_1\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IApp}$. Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Subset_{\|d_1\ d_2\|}$.

a) $\Omega_1 \Downarrow \Delta_1$, $d' \Downarrow c'$, $\Delta, s \vdash c_1 \Rightarrow (\Delta, c'[\Delta_1] \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_1 \Rightarrow (\lambda.d')[\Omega_1]$ (tal que $(\lambda.d')[\Omega_1] \Downarrow c'[\Delta_1]$).

b) $\Delta, c'[\Delta_1] \cdot s \vdash c_2 \Rightarrow (\Delta, v_{m_2} \cdot c'[\Delta_1] \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_2 \Rightarrow v_2$ (tal que $v_2 \Downarrow v_{m_2}$).

c) $v_{m_2} \cdot \Delta_1, s \vdash c' \Subset_{\|d'\|}$ por hipótesis de coinducción sobre $v_2 \cdot \Omega_1 \vdash d' \mathrel{\widehat{\Rightarrow}}$.

d) $\Delta, v_{m_2} \cdot c'[\Delta_1] \cdot s \vdash \mathsf{IApp} \mathrel{\widehat{\Rightarrow}}$ por definición empleando XI.*c*.

e) $\Delta, v_{m_2} \cdot c'[\Delta_1] \cdot s \vdash \mathsf{IApp} \overset{\cong}{\underset{\|d'\|}{\Rightarrow}}$ por definición empleando XI.d.

f) $\Delta, c'[\Delta_1] \cdot s \vdash c_2 \cdot \mathsf{IApp} \overset{\cong}{\underset{\|d_2\|}{\Rightarrow}}$ por la regla $\overset{\cong}{\underset{n}{\Rightarrow}}$-perform sobre XI.$b$ y XI.e.

g) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \overset{\cong}{\underset{\|d_1\|+1}{\Rightarrow}}$ por la regla $\overset{\cong}{\underset{n}{\Rightarrow}}$-perform sobre XI.$a$ y XI.f.

h) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \overset{\cong}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$ por definición $\|d_1\ d_2\| = \|d_1\| + 1$.

(XII) $d = d_1\ d_2$.

Hipótesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d')[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot (\mu.\lambda.d')[\Omega_1] \cdot \Omega_1 \vdash d' \overset{\cong}{\Rightarrow}$, $\Omega \vdash d_1\ d_2 \overset{\cong}{\Rightarrow}$, $d \Downarrow c$, $\Omega \Downarrow \Delta$. Usando la hipótesis $d_1\ d_2 \Downarrow c$, por definición de $\Downarrow$ necesariamente $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IApp}$.
Por demostrar $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \overset{\cong}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$.

a) $\Omega_1 \Downarrow \Delta_1$, $d' \Downarrow c'$, $\Delta, s \vdash c_1 \Rightarrow (\Delta, c'[\Delta_1]_{rec} \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d')[\Omega_1]$ (tal que $(\mu.\lambda.d')[\Omega_1] \Downarrow c'[\Delta_1]_{rec}$).

b) $\Delta, c'[\Delta_1]_{rec} \cdot s \vdash c_2 \Rightarrow (\Delta, v_{m_2} \cdot c'[\Delta_1]_{rec} \cdot s)$ por Teorema 3 sobre $\Omega \vdash d_2 \Rightarrow v_2$ (tal que $v_2 \Downarrow v_{m_2}$).

c) $v_{m_2} \cdot c'[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c' \overset{\cong}{\underset{\|d'\|}{\Rightarrow}}$ por hipótesis de coinducción sobre $v_2 \cdot (\mu.\lambda.d')[\Omega_1] \cdot \Omega_1 \vdash d' \overset{\cong}{\Rightarrow}$.

d) $\Delta, v_{m_2} \cdot c'[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\cong}{\Rightarrow}$ por definición empleando XII.c.

e) $\Delta, v_{m_2} \cdot c'[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\cong}{\underset{\|d'\|}{\Rightarrow}}$ por definición empleando XII.d.

f) $\Delta, c'[\Delta_1]_{rec} \cdot s \vdash c_2 \cdot \mathsf{IApp} \overset{\cong}{\underset{\|d_2\|}{\Rightarrow}}$ por la regla $\overset{\cong}{\underset{n}{\Rightarrow}}$-perform sobre XII.$b$ y XII.e.

g) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \overset{\cong}{\underset{\|d_1\|+1}{\Rightarrow}}$ por la regla $\overset{\cong}{\underset{n}{\Rightarrow}}$-perform sobre XII.$a$ y XII.f.

h) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \overset{\cong}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$ por definición $\|d_1\ d_2\| = \|d_1\| + 1$. □

Teorema 6 (Corrección en el caso de no terminación)**.** *Sea Ω un entorno sin nombres, Δ un entorno de máquina, d una expresión sin nombres, c un código de máquina. Si*

$$\Omega \vdash d \overset{\cong}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \Downarrow \Delta$$

entonces, para toda pila s,

$$\Delta, s \vdash c \overset{\cong}{\Rightarrow}$$

Demostración. Hipótesis $\Omega \vdash d \mathrel{\widetilde{\Rightarrow}}$, $\quad d \Downarrow c$, $\quad \Omega \Downarrow \Delta$. Por demostrar $\Delta, s \vdash c \mathrel{\widetilde{\Rightarrow}}$.

(I) $\Delta, s \vdash c \underset{\|d\|}{\mathrel{\widetilde{\Rightarrow}}}$ por Lema 5 sobre las hipótesis.

(II) $\Delta, s \vdash c \mathrel{\widetilde{\Rightarrow}}$ por Lema 4 sobre I. $\square$

Bibliografía

[1] Andreas Abel. "A Polymorphic Lambda-Calculus with Sized Higher-Order Types". Tesis doct. Ludwig-Maximilians-Universität München, 2006.

[2] Andreas M. Abel y Brigitte Pientka. "Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity". En: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, págs. 185-196. ISBN: 9781450323260. DOI: 10.1145/2500365.2500591.

[3] Andreas Abel y Brigitte Pientka. "Well-founded recursion with copatterns and sized types". En: *Journal of Functional Programming* 26 (2016), e2. DOI: 10.1017/S0956796816000022.

[4] Andreas Abel y col. "Copatterns: Programming Infinite Structures by Observations". En: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: Association for Computing Machinery, 2013, págs. 27-38. ISBN: 9781450318327. DOI: 10.1145/2429069.2429075.

[5] Mads Sig Ager. "From Natural Semantics to Abstract Machines". En: *Logic Based Program Synthesis and Transformation*. Ed. por Sandro Etalle. Vol. 3573. LNCS. LOPSTR 2004. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, págs. 245-261. ISBN: 978-3-540-31683-1. DOI: 10.1007/11506676_16.

[6] Amal Ahmed. "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types". En: *Programming Languages and Systems*. Ed. por Peter Sestoft. Vol. 3924. LNCS. ESOP 2006. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, págs. 69-83. ISBN: 978-3-540-33096-7. DOI: 10.1007/11693024_6.

[7] Abhishek Anand y col. *CertiCoq: A verified compiler for Coq*. CoqPL 2017. Paris. Ene. de 2017.

[8] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Ed. por Dov M. Gabbay y Jon Barwise. Second. Vol. 27. Applied Logic Series. Springer, Dordrecht, 2002. DOI: 10.1007/978-94-015-9934-4.

[9] Casper Bach Poulsen y Peter D. Mosses. "Flag-based big-step semantics". En: *Journal of Logical and Algebraic Methods in Programming* 88 (2017), págs. 174-190. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2016.05.001.

[10] David Baelde y col. "Abella: A System for Reasoning about Relational Specifications". En: *Journal of Formalized Reasoning* 7.2 (dic. de 2014), págs. 1-89. DOI: 10.6092/issn.1972-5787/4650.

[11] Bruno Barras. "Auto-validation d'un système de preuves avec familles inductives". Tesis doct. Université Paris 7 - Denis Diderot, 1999.

[12] Olivier Savary Bélanger y Andrew W Appel. "Shrink Fast Correctly!" En: *Proc. 19th Int. Symp. Princ. Pract. Declar. Program.* PPDP '17. New York, NY, USA: Association for Computing Machinery, 2017, págs. 49-60. ISBN: 9781450352918. DOI: 10.1145/3131851.3131859.

[13] Nick Benton y Chung-Kil Hur. "Biorthogonality, Step-Indexing and Compiler Correctness". En: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. Edinburgh, Scotland: Association for Computing Machinery, 2009, págs. 97-108. ISBN: 9781605583327. DOI: 10.1145/1596550.1596567.

[14] Malgorzata Biernacka y Dariusz Biernacki. "Formalizing Constructions of Abstract Machines for Functional Languages in Coq". En: *7th International Workshop on Reduction Strategies in Rewriting and Programming*. Ed. por Jürgen Giesl. WRS 2007. Paris, France: Preeliminary Proceedings, jun. de 2007, págs. 84-99.

[15] Malgorzata Biernacka y Witold Charatonik. "Deriving an Abstract Machine for Strong Call by Need". En: *4th International Conference on Formal Structures for Computation and Deduction*. Ed. por Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). FSCD 2019. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 8:1-8:20. ISBN: 978-3-95977-107-8. DOI: 10.4230/LIPIcs.FSCD.2019.8.

[16] Malgorzata Biernacka, Witold Charatonik y Klara Zielinska. "Generalized Refocusing: From Hybrid Strategies to Abstract Machines". En: *2nd International Conference on Formal Structures for Computation and Deduction*. Ed. por Dale Miller. Vol. 84. Leibniz International Proceedings in Informatics (LIPIcs). FSCD 2017. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1-10:17. ISBN: 978-3-95977-047-7. DOI: 10.4230/LIPIcs.FSCD.2017.10.

[17] Małgorzata Biernacka y Olivier Danvy. "A Concrete Framework for Environment Machines". En: *ACM Transactions on Computational Logic* 9.1 (dic. de 2007), 6-es. ISSN: 1529-3785. DOI: 10.1145/1297658.1297664.

[18] Sandrine Blazy, Zaynah Dargaye y Xavier Leroy. "Formal Verification of a C Compiler Front-End". En: *F*. Ed. por Jayadev Misra, Tobias Nipkow y Emil Sekerinski. Vol. 4085. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, págs. 460-475. ISBN: 978-3-540-37216-5. DOI: 10.1007/11813040_31.

[19] Sandrine Blazy y Xavier Leroy. "Mechanized semantics for the Clight subset of the C language". En: *Journal of Automated Reasoning* 43.3 (2009), págs. 263-288. DOI: 10.1007/s10817-009-9148-3.

[20] Samuel Boutin. *Proving Correctness of the Translation from Mini-ML to the CAM with the Coq Proof Development System*. Research Report RR-2536. INRIA, 1995. URL: https://hal.inria.fr/inria-00074142/file/RR-2536.pdf.

[21] Samuel Boutin. "Using reflection to build efficient and certified decision procedures". En: *Theoretical Aspects of Computer Software*. Ed. por Martín Abadi y Takayasu Ito. TACS 1997. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, págs. 515-529. ISBN: 978-3-540-69530-1. DOI: 10.1007/BFb0014565.

[22] Ana Bove, Peter Dybjer y Ulf Norell. "A Brief Overview of Agda – A Functional Language with Dependent Types". En: *Theorem Proving in Higher Order Logics*. Ed. por Stefan Berghofer y col. TPHOLs 2009. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, págs. 73-78. ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_6.

[23] D. L. Bowen, L. M. Bird y W. F. Clocksin. "A Portable Prolog Compiler". En: *Proceedings of the Logic Programming Workshop*. Albufeira, Portugal, 1983.

[24] Rod Burstall y Furio Honsell. "Operational Semantics in a Natural Deduction Setting". En: *Logical Frameworks*. USA: Cambridge University Press, 1991, págs. 185-214. ISBN: 0521413001.

[25] Arthur Charguéraud. "Pretty-Big-Step Semantics". En: *Programming Languages and Systems*. Ed. por Matthias Felleisen y Philippa Gardner. Vol. 7792. LNCS. ESOP 2013. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, págs. 41-60. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_3.

[26] Adam Chlipala. "A Verified Compiler for an Impure Functional Language". En: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: Association for Computing Machinery, 2010, págs. 93-106. ISBN: 9781605584799. DOI: 10.1145/1706299.1706312.

[27] Alonzo Church. "A formulation of the simple theory of types". En: *Journal of Symbolic Logic* 5.2 (1940), págs. 56-68. DOI: 10.2307/2266170.

[28] Alberto Ciaffaglione y Ivan Scagnetto. "Mechanizing type environments in weak HOAS". En: *Theoretical Computer Science* 606 (2015), págs. 57-78. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2015.07.019.

[29] Manuel Clavel y col. *All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. LNCS. Springer Berlin Heidelberg, 2007. ISBN: 978-3-540-71940-3. DOI: 10.1007/978-3-540-71999-1.

[30] M. Clavel y col. "Maude: specification and programming in rewriting logic". En: *Theoretical Computer Science* 285.2 (2002). Rewriting Logic and its Applications, págs. 187-243. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(01)00359-0.

[31] M. Clavel y col. “Principles of Maude”. En: *Electronic Notes in Theoretical Computer Science* 4 (1996). RWLW96, First International Workshop on Rewriting Logic and its Applications, págs. 65-89. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)00034-9.

[32] M. Clavel y col. “The Maude System”. En: *Rewriting Techniques and Applications*. Ed. por Paliath Narendran y Michael Rusinowitch. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, págs. 240-243. ISBN: 978-3-540-48685-5. DOI: /10.1007/3-540-48685-2_18.

[33] Jacques Cohen y Timothy J. Hickey. “Parsing and Compiling Using Prolog”. En: *ACM Transactions on Programming Languages and Systems* 9.2 (mar. de 1987), págs. 125-163. ISSN: 0164-0925. DOI: 10.1145/22719.22946.

[34] A. Colmerauer. “Metamorphosis grammars”. En: *Natural Language Communication with Computers*. Ed. por Leonard Bolc. Vol. 63. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, págs. 133-188. ISBN: 978-3-540-35765-0. DOI: 10.1007/BFb0031371.

[35] Olivier Danvy y Lasse R Nielsen. *Refocusing in Reduction Semantics*. Inf. téc. RS-04-26. BRICS, Department of Computer Science, University of Aarhus, nov. de 2004.

[36] Zaynah Dargaye. “Vérification formelle d’un compilateur optimisant pour langages fonctionnels”. Tesis doct. Université Paris-Diderot - Paris VII, jul. de 2009. URL: https://tel.archives-ouvertes.fr/tel-00452440/file/these_Zaynah_Dargaye.pdf.

[37] N.G de Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. En: *Indagationes Mathematicae* 75.5 (1972), págs. 381-392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0.

[38] Joëlle Despeyroux. *Proof of translation in natural semantics*. Research Report RR-0514. INRIA, 1986, pág. 13. URL: https://hal.inria.fr/inria-00076040/file/RR-0514.pdf.

[39] Joëlle Despeyroux, Amy Felty y André Hirschowitz. “Higher-order abstract syntax in Coq”. En: *Typed Lambda Calculi and Applications*. Ed. por Mariangiola Dezani-Ciancaglini y Gordon Plotkin. TLCA 1995. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, págs. 124-138. ISBN: 978-3-540-49178-1. DOI: 10.1007/BFb0014049.

[40] Catherine Dubois y Valérie Ménissier-Morain. “Certification of a Type Inference Tool for ML: Damas–Milner within Coq”. En: *Journal of Automated Reasoning* 23.3 (nov. de 1999), págs. 319-346. ISSN: 1573-0670. DOI: 10.1023/A:1006285817788.

[41] Francisco Durán y col. “Programming and symbolic computation in Maude”. En: *Journal of Logical and Algebraic Methods in Programming* 110 (2020), pág. 100497. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2019.100497.

[42] Andrew Gacek. "The Abella Interactive Theorem Prover (System Description)". En: *Automated Reasoning*. Ed. por Alessandro Armando, Peter Baumgartner y Gilles Dowek. IJCAR 2008. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, págs. 154-161. ISBN: 978-3-540-71070-7. DOI: 10.1007/978-3-540-71070-7_13.

[43] Eduardo Giménez. "Un Calcul de Constructions Infinies et son application a la vérification de systemes communicants". Tesis doct. École Normale Supérieure de Lyon, dic. de 1996.

[44] Michael J. Gordon, Robin Milner y Christopher P. Wadsworth. *Edinburgh LCF. A Mechanised Logic of Computation*. Ed. por G. Goos y J. Hartmanis. Vol. 78. LNCS. Springer-Verlag, 1979. ISBN: 0-387-09724-4.

[45] Benjamin Grégoire. "Compilation des termes de preuves: un (nouveau) mariage entre Coq et OCaml". Thése de doctorat, spécialité informatique. École Polytechnique, France: Université Paris 7 - Denis Diderot, dic. de 2003.

[46] Benjamin Grégoire y Xavier Leroy. "A Compiled Implementation of Strong Reduction". En: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. Pittsburgh, PA, USA: Association for Computing Machinery, 2002, págs. 235-246. ISBN: 1581134878. DOI: 10.1145/581478.581501.

[47] Thérèse Hardin, Luc Maranget y Bruno Pagano. "Functional runtime systems within the lambda-sigma calculus". En: *Journal of Functional Programming* 8.2 (1998), págs. 131-176. DOI: 10.1017/S0956796898002986.

[48] Peter Henderson. *Functional Programming Application and Implementation*. Ed. por C. A. R. Hoare. Computer Science. Prentice-Hall International, 1980. ISBN: 0-13-331579-7.

[49] M. V. Hermenegildo y col. "An overview of Ciao and its design philosophy". En: *Theory and Practice of Logic Programming* 12.1–2 (2012), págs. 219-252. DOI: 10.1017/S1471068411000457.

[50] Jacques-Henri Jourdan, François Pottier y Xavier Leroy. "Validating LR(1) Parsers". En: *Programming Languages and Systems*. Ed. por Helmut Seidl. ESOP 2012. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 397-416. ISBN: 978-3-642-28869-2. DOI: 10.1007/978-3-642-28869-2_20.

[51] Jacques-Henri Jourdan y col. "A Formally-Verified C Static Analyzer". En: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: Association for Computing Machinery, 2015, págs. 247-259. ISBN: 9781450333009. DOI: 10.1145/2676726.2676966.

[52] Ralf Jung y col. "Higher-Order Ghost State". En: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. New York, NY, USA: Association for Computing Machinery, 2016, págs. 256-269. ISBN: 9781450342193. DOI: 10.1145/2951913.2951943.

[53] Ralf Jung y col. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". En: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151.

[54] Ralf Jung y col. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". En: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. New York, NY, USA: Association for Computing Machinery, 2015, págs. 637-650. ISBN: 9781450333009. DOI: 10.1145/2676726.2676980.

[55] Gilles Kahn. "Natural semantics". En: *STACS 87*. Ed. por Franz Josef Brandenburg, Guy Vidal-Naquet y Martin Wirsing. Vol. 247. LNCS. Berlin, Heidelberg: Springer, 1987, págs. 22-39. ISBN: 978-3-540-47419-7. DOI: 10.1007/BFb0039592.

[56] Yong Kiam Tan y col. "The verified CakeML compiler backend". En: *Journal of Functional Programming* 29 (2019), e2. DOI: 10.1017/S0956796818000229.

[57] Robbert Krebbers y col. "The Essence of Higher-Order Concurrent Separation Logic". En: *Programming Languages and Systems*. Ed. por Hongseok Yang. Vol. 10201. LNCS. ESOP 2017. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, págs. 696-723. ISBN: 978-3-662-54434-1. DOI: 10.1007/978-3-662-54434-1_26.

[58] Ramana Kumar y col. "CakeML: A Verified Implementation of ML". En: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, págs. 179-191. ISBN: 9781450325448. DOI: 10.1145/2535838.2535841.

[59] Fabian Kunze, Gert Smolka y Yannick Forster. "Formal Small-Step Verification of a Call-by-Value Lambda Calculus Machine". En: *Programming Languages and Systems*. Ed. por Sukyoung Ryu. Vol. 11275. LNCS. APLAS 2018. Cham: Springer International Publishing, 2018, págs. 264-283. ISBN: 978-3-030-02768-1. DOI: 10.1007/978-3-030-02768-1_15.

[60] Peter John Landin. "The Mechanical Evaluation of Expressions". En: *The Computer Journal* 6.4 (ene. de 1964), págs. 308-320. ISSN: 0010-4620. DOI: 10.1093/comjnl/6.4.308.

[61] Xavier Leroy. Comunicación personal. Mar. de 2020.

[62] Xavier Leroy. "A formally verified compiler back-end". En: *Journal of Automated Reasoning* 43.4 (2009), págs. 363-446. DOI: 10.1007/s10817-009-9155-4.

[63] Xavier Leroy. "Coinductive Big-Step Operational Semantics". En: *Programming Languages and Systems*. Ed. por Peter Sestoft. Vol. 3924. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, págs. 54-68. ISBN: 978-3-540-33096-7. DOI: 10.1007/11693024_5.

[64] Xavier Leroy. "Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant". En: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. New York, NY, USA: Association for Computing Machinery, 2006, págs. 42-54. ISBN: 1595930272. DOI: 10.1145/1111037.1111042.

[65] Xavier Leroy. "Formal Verification of a Realistic Compiler". En: *Communications of the ACM* 52.7 (jul. de 2009), págs. 107-115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814.

[66] Xavier Leroy. *Functional programming languages Part II: abstract machines*. MPRI 2-4, INRIA. INRIA Paris-Rocquencourt. 2015-2016. URL: https://xavierleroy.org/mpri/2-4/machines.pdf.

[67] Xavier Leroy. *Le système Caml Special Light: modules et compilation efficace en Caml*. Research Report RR-2721. INRIA, 1995.

[68] Xavier Leroy. *L'éternité c'est long ... Sémantiques de la divergence: théorie des domaines et approches coinductives*. Sixième cours. Collège de France. Ene. de 2020. URL: https://www.college-de-france.fr/media/xavier-leroy/UPL2331162062302586657_leroy_cours_6.pdf.

[69] Xavier Leroy. *The formal verification of compilers*. DeepSpec Summer School 2017. Inria Paris. 2017. URL: https://deepspec.org/event/dsss17/leroy-dsss17.pdf.

[70] Xavier Leroy. *The ZINC experiment : an economical implementation of the ML language*. Inf. téc. RT-0117. INRIA, feb. de 1990, pág. 100. URL: https://hal.inria.fr/inria-00070049.

[71] Xavier Leroy y Hervé Grall. "Coinductive big-step operational semantics". En: *Information and Computation* 207.2 (2009), págs. 284-304. ISSN: 0890-5401. DOI: 10.1016/j.ic.2007.12.004.

[72] José Meseguer. "A Logical Theory of Concurrent Objects and Its Realization in the Maude Language". En: *Research Directions in Concurrent Object-Oriented Programming*. Cambridge, MA, USA: MIT Press, 1993, págs. 314-390. ISBN: 0262011395.

[73] José Meseguer. "General Logics". En: *Logic Colloquium'87*. Ed. por H.-D. Ebbinghaus y col. Vol. 129. Studies in Logic and the Foundations of Mathematics. Elsevier, 1989, págs. 275-329. DOI: 10.1016/S0049-237X(08)70132-0.

[74] Dale Miller y Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. DOI: 10.1017/CBO9781139021326.

[75] José Morales. "Técnicas Avanzadas de Compilación para Programación Lógica". Tesis doct. Universidad Politécnica de Madrid, 2010.

[76] Keiko Nakata y Tarmo Uustalu. "Trace-Based Coinductive Operational Semantics for While". En: *Theorem Proving in Higher Order Logics*. Ed. por Stefan Berghofer y col. Vol. 5674. LNCS. TPHOLs 2009. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, págs. 375-390. ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_26.

[77] Aleksandar Nanevski, Frank Pfenning y Brigitte Pientka. "Contextual Modal Type Theory". En: *ACM Transactions on Computational Logic* 9.3 (jun. de 2008). ISSN: 1529-3785. DOI: 10.1145/1352582.1352591.

[78] Hanne Riis Nielson y Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer London, 2007. ISBN: 978-1-84628-691-9. DOI: 10.1007/978-1-84628-692-6.

[79] Tobias Nipkow, Lawrence C. Paulson y Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[80] Ulf Norell. "Dependently Typed Programming in Agda". En: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. por Pieter Koopman, Rinus Plasmeijer y Doaitse Swierstra. AFP 2008. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, págs. 230-266. ISBN: 978-3-642-04652-0. DOI: 10.1007/978-3-642-04652-0_5.

[81] Ulf Norell. "Towards a practical programming language based on dependent type theory". Tesis doct. Department of Computer Science y Engineering, Chalmers University of Technology, sep. de 2007.

[82] Scott Owens y col. "Functional Big-Step Semantics". En: *Programming Languages and Systems*. Ed. por Peter Thiemann. ESOP 2016. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, págs. 589-615. ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_23.

[83] Zoe Paraskevopoulou. "Verified Optimizations for Functional Languages". Tesis doct. Princeton University, nov. de 2020.

[84] Zoe Paraskevopoulou y Andrew W Appel. "Closure Conversion is Safe for Space". En: *Proceedings of the ACM on Programming Languages* 3.ICFP (jul. de 2019). DOI: 10.1145/3341687.

[85] Christine Paulin-Mohring. "Extraction de programmes dans le Calcul des Constructions". Tesis doct. Université Paris-Diderot - Paris VII, ene. de 1989. URL: https://tel.archives-ouvertes.fr/tel-00431825/file/new.pdf.

[86] Lawrence C. Paulson. *Isabelle: The Next 700 Theorem Provers*. Inf. téc. UCAM-CL-TR-143. University of Cambridge, Computer Laboratory, ago. de 1988. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-143.pdf.

[87] Lawrence C. Paulson. "The foundation of a generic theorem prover". En: *Journal of Automated Reasoning* 5.3 (1989), págs. 363-397. DOI: 10.1007/BF00248324.

[88] Fernando C. N. Pereira y David H. D. Warren. "Parsing as Deduction". En: *Proceedings of the 21st Annual Meeting on Association for Computational Linguistics*. ACL '83. Cambridge, Massachusetts, USA: Association for Computational Linguistics, 1983, págs. 137-144. DOI: 10.3115/981311.981338.

[89] F. Pfenning y C. Elliott. "Higher-Order Abstract Syntax". En: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, págs. 199-208. ISBN: 0897912691. DOI: 10.1145/53990.54010.

[90] Brigitte Pientka. "Beluga: Programming with Dependent Types, Contextual Data, and Contexts". En: *Functional and Logic Programming*. Ed. por Matthias Blume, Naoki Kobayashi y Germán Vidal. FLOPS 2010. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, págs. 1-12. ISBN: 978-3-642-12251-4. DOI: 10.1007/978-3-642-12251-4_1.

[91] Brigitte Pientka y Joshua Dunfield. "Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)". En: *Automated Reasoning*. Ed. por Jürgen Giesl y Reiner Hähnle. IJCAR 2010. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, págs. 15-21. ISBN: 978-3-642-14203-1. DOI: 10.1007/978-3-642-14203-1_2.

[92] Maciej Pirog y Dariusz Biernacki. "A Systematic Derivation of the STG Machine Verified in Coq". En: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. New York, NY, USA: Association for Computing Machinery, 2010, págs. 25-36. ISBN: 9781450302524. DOI: 10.1145/1863523.1863528.

[93] Andrew M. Pitts. "Nominal logic, a first order theory of names and binding". En: *Information and Computation* 186.2 (2003). Theoretical Aspects of Computer Software (TACS 2001), págs. 165-193. ISSN: 0890-5401. DOI: 10.1016/S0890-5401(03)00138-X.

[94] Dag Prawitz. *Natural Deduction. A Proof-Theoretical Study*. Stockholm Studies in Philosophy. Stockholm: Almqvist & Wiksell, 1965.

[95] Jorge Luis Sacchini. "On type-based termination and dependent pattern matching in the calculus of inductive constructions". Tesis doct. École Nationale Supérieure des Mines de Paris, jun. de 2011. URL: https://pastel.archives-ouvertes.fr/pastel-00622429.

[96] Olivier Savary Bélanger. "Verified Extraction for Coq". Tesis doct. Princeton University, nov. de 2019.

[97] Ivan Scagnetto. "Reasoning about names in Higher-Order Abstract Syntax". Tesis doct. University of Udine, 2001.

[98] Stuart M. Shieber, Yves Schabes y Fernando C.N. Pereira. "Principles and implementation of deductive parsing". En: *The Journal of Logic Programming* 24.1 (1995). Computational Linguistics and Logic Programming, págs. 3-36. ISSN: 0743-1066. DOI: 10.1016/0743-1066(95)00035-I.

[99] Filip Sieczkowski, Małgorzata Biernacka y Dariusz Biernacki. "Automating Derivations of Abstract Machines from Reduction Semantics: A Generic Formalization of Refocusing in Coq". En: *Implementation and Application of Functional Languages*. Ed. por Jurriaan Hage y Marco T Morazán. Vol. 6647. LNCS. IFL 2010. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, págs. 72-88. ISBN: 978-3-642-24276-2. DOI: 10.1007/978-3-642-24276-2_5.

[100] Matthieu Sozeau y col. "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq". En: *Proceedings of the ACM on Programming Languages* 4.POPL (dic. de 2019). DOI: 10.1145/3371076.

[101] Matthieu Sozeau y col. "The MetaCoq Project". En: *Journal of Automated Reasoning* 64.5 (2020), págs. 947-999. ISSN: 1573-0670. DOI: 10.1007/s10817-019-09540-0.

[102] Kathrin Stark. "Mechanising Syntax with Binders in Coq". Tesis doct. Saarland University, 2020, pág. 206.

[103] Kathrin Stark, Steven Schäfer y Jonas Kaiser. "Autosubst 2: Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions". En: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019. Cascais, Portugal: Association for Computing Machinery, 2019, págs. 166-180. ISBN: 9781450362221. DOI: 10.1145/3293880.3294101. URL: https://doi.org/10.1145/3293880.3294101.

[104] David Thibodeau. "An Intensional Type Theory of Coinduction using Copatterns". Tesis doct. McGill University, dic. de 2020.

[105] Mads Tofte. "Operational Semantics and Polymorphic Type Inference". Tesis doct. University of Edinburgh, 1987.

[106] Pierre-Nicolas Tollitte, David Delahaye y Catherine Dubois. "Producing Certified Functional Code from Inductive Specifications". En: *Certified Programs and Proofs*. Ed. por Chris Hawblitzel y Dale Miller. Vol. 7679. LNCS. CPP 2012. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 76-91. ISBN: 978-3-642-35308-6. DOI: 10.1007/978-3-642-35308-6_9.

[107] Christian Urban. "Nominal Techniques in Isabelle/HOL". En: *Journal of Automated Reasoning* 40.4 (2008), págs. 327-356. DOI: 10.1007/s10817-008-9097-2.

[108] Christian Urban y Christine Tasson. "Nominal Techniques in Isabelle/HOL". En: *Automated Deduction – CADE-20*. Ed. por Robert Nieuwenhuis. Vol. 3632. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, págs. 38-53. ISBN: 978-3-540-31864-4. DOI: 10.1007/11532231_4.

[109] Tarmo Uustalu. "Coinductive Big-Step Semantics for Concurrency". En: Proceedings 5th Workshop on *Programming Language Approaches to Concurrency and Communication-cEntric Software,* Rome, Italy, 23rd March 2013. Ed. por Nobuko Yoshida y Wim Vanderbauwhede. Vol. 137. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2013, págs. 63-78. DOI: 10.4204/EPTCS.137.6.

[110] Philip Wadler. "Programming Language Foundations in Agda". En: *Formal Methods: Foundations and Applications*. Ed. por Tiago Massoni y Mohammad Reza Mousavi. Vol. 11254. LNCS. SBMF 2018. Cham: Springer International Publishing, 2018, págs. 56-73. ISBN: 978-3-030-03044-5. DOI: 10.1007/978-3-030-03044-5_5.

[111] Yuting Wang. "A Higher-Order Abstract Syntax Approach to the Verified Compilation of Functional Programs". Tesis doct. University of Minnesota, 2016.

[112] Yuting Wang y Gopalan Nadathur. "A Higher-Order Abstract Syntax Approach to Verified Transformations on Functional Programs". En: *Programming Languages and Systems*. Ed. por Peter Thiemann. ESOP 2016. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, págs. 752-779. ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_29.

[113] David H. D. Warren. "Applied Logic. its use and implementation as a programming tool". Tesis doct. University of Edinburgh, 1977. URL: https://www.era.lib.ed.ac.uk/handle/1842/6648.

[114] David H. D. Warren. "Logic programming and compiler writing". En: *Software: Practice and Experience* 10.2 (1980), págs. 97-125. DOI: 10.1002/spe.4380100203.

[115] Benjamin Werner. "Une Théorie des Constructions Inductives". Tesis doct. Université Paris-Diderot - Paris VII, mayo de 1994. URL: https://tel.archives-ouvertes.fr/tel-00196524/file/main.pdf.

[116] Angel Zúñiga. *Coinductive Natural Semantics for Compiler Verification in Coq: the Coq Development*. Ago. de 2020. URL: https://sites.google.com/a/ciencias.unam.mx/zuniga/repository/cnsvcompiler.tgz.

[117] Angel Zúñiga. "Un compilador correcto verificado de Mini-ML a la máquina SECD en Coq". Tesis de mtría. Posgrado en Ciencia e Ingeniería de la Computación, Universidad Nacional Autónoma de México, 2016.