UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

UN COMPILADOR CORRECTO VERIFICADO DE MINI-ML A LA MÁQUINA SECD EN COQ

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIAS (COMPUTACIÓN)

PRESENTA:
ANGEL FRANCISCO ZÚÑIGA CHÁVEZ

TUTOR:
DR. FRANCISCO HERNÁNDEZ QUIROZ
FACULTAD DE CIENCIAS, UNAM

COTUTOR:
DR. FAVIO EZEQUIEL MIRANDA PEREA
FACULTAD DE CIENCIAS, UNAM

MÉXICO, D.F. MAYO 2016

Agradecimientos

A Dios

A mis padres y a mi hermano

A mi abuela y a mi tía

A mis abuelos y a mis tías que Dios me dió

A mi padrino y familia

A mi florecita hermosa

Quisiera agradecer a mi tutores. A Francisco por su trato siempre amable, tranquilo, paciente y cordial, por confiar en mí, en él encontré cualidades que esperaba de un tutor. A Favio por aceptar ser mi tutor con todo lo que eso implicó, por aceptar brindarme libertad en mis ideas, trabajo y estudio.

A mis sinodales Sofía, Gerardo y Carlos. Por su trato amable, cordial y generoso. Sus comentarios, observaciones y correcciones enriquecieron este trabajo.

A mis muy queridos y estimados profesores: Elisa, Francisco, Amparo, Jorge, Lucy, Lupita, Sergio, Hanna, Favio, David, Salvador, Adrián, Sofía, Gerardo, Carlos y todos los demás. Muchos de ellos han sido mis profesores, unos además de ser mis profesores me han brindado su amistad, consejos y calor humano más allá de las aulas y otros simplemente esto último (pero no por ello menos valioso). Pero lo más importante pienso que caminamos hacia el respeto a la libertad de creencias, de pensamientos, de ideas, hacia la búsqueda del conocimiento, aceptando con humildad nuestros errores y aprendiendo los unos de los otros. Cualidades que considero nos deben distinguir como universitarios y nos hacen más humanos.

A todas las personas que en silencio han trabajado por el bien común, son el corazón de causas justas y nobles, de alegrías y de esperanza.

A todas las personas que a lo largo de todos estos años han creído y puesto su confianza en mí, se los agradezco mucho.

A Sandra Ramírez por todo, sin ella tal vez esto no hubiese sido posible.

A Lulú, Amalia y Cecilia por brindarme siempre atención con amabilidad y respeto.

Al CONACYT por la beca otorgada durante mis estudios correspondientes a esta maestría.

Índice general

CAPÍTULO 1

Introducción

En el caso más usual un programador escribe programas escritos en un lenguaje de alto nivel y hace uso de un compilador para obtener una versión ejecutable de éstos.

Un error en el compilador significaría que éste introdujera errores en las versiones ejecutables de los programas y esto puede tener diversas consecuencias finales adversas. Dos de las más significativas son pérdidas económicas y de salud.

Un ejemplo de la primera se da cuando suponemos que el compilador que se utiliza para el administrador de base de datos *Oracle* tiene un error. Este error puede tener como consecuencia que al ejecutar el manejador bajo ciertas condiciones éste aborte de manera inesperada. Es claro que esto ocasionaría pérdidas económicas para los clientes de *Oracle* y hasta pérdidas millonarias para esta compañía.

Por su parte, un ejemplo de la segunda se da en el contexto de un compilador que se utilice para generar un ejecutable de una aplicación que controla un instrumento médico, pensemos en un láser. Es claro que, esto puede tener consecuencias catastróficas en la salud de un paciente e incluso puede ocasionar la pérdida de vidas humanas.

Por tanto es imprescindible contar con compiladores correctos, es decir, que no contengan ningún tipo de error.[1]

En consecuencia, es nuestro objetivo en el presente trabajo desarrollar un pequeño compilador correcto que nos permita conocer (y comprender) cuáles son los pasos necesarios para desarrollarlo y cómo realizarlos. Adicionalmente que sirva para experimentar y como base para futuras extensiones.

A manera de introducción, de forma abstracta podemos pensar en un compilador de la siguiente manera.

[1] Esto desde luego toma una mayor relevancia en el contexto de aplicaciones de misión crítica.

Sean L y L' dos lenguajes,[2] un compilador es una función f con dominio L y codominio L', es decir $f : L \to L'$.[3]

Por ejemplo, podemos considerar L como el conjunto de cadenas que representan a los números decimales y L' como el conjunto de cadenas que representan a los números binarios. De esta manera un compilador f sería una función que a cada número decimal le asignaría un (y sólo un) número binario. Es decir, el compilador realizaría una conversión de números decimales a binarios.[4] Por ejemplo $f(9) = 101$.[5]

Abordando nuestro objetivo como primer paso es necesario elegir nuestro lenguaje fuente, es decir, nuestro lenguaje a compilar.

Idealmente todos los programas que se escriben deben ser correctos. En vías de lograr este fin se debe poder razonar sobre ellos y, las propiedades que éstos cumplen. Por tanto, es deseable que el lenguaje en el que se escriben facilite estas tareas.

Un lenguaje funcional es aquel que suele basarse en el cálculo lambda [HS08; Bar12; Han04; Chu41]. Un programa escrito en un lenguaje funcional[6] goza de referencia transparencial: la única característica que es importante de una expresión es su valor, cualquier subexpresión se puede reemplazar por cualquier otra de igual valor; más aún el valor de una expresión es (con ciertos límites) el mismo donde quiera que esta aparezca. Es posible entonces realizar *razonamiento ecuacional* (reemplazar (sub)-expresiones por (sub)-expresiones de igual valor) sobre el programa. Lo anterior debido a que el cálculo lambda es un formalismo teórico el cual cuenta con bases matemáticas sólidas.

Las anteriores y otras características de la programación funcional se han discutido ya en [Bac78; Hug89]. En oposición una de sus desventajas ha resultado ser la eficiencia [Veg84; Hud89].

Por lo anterior, hemos elegido como lenguaje fuente un pequeño lenguaje funcional, el cual presentaremos más adelante.

Elijamos ahora nuestro lenguaje objetivo.

Un compilador en principio debe ser capaz de generar código de máquina, es sobre esta máquina sobre la cual el programa finalmente se ejecutará.

Esta máquina puede tratarse de una arquitectura real o de una máquina abstracta.

En general las ventajas de trabajar con una máquina abstracta son:

- Que en principio se puede trabajar únicamente con las características esenciales de una máquina, es decir, aquellas que en principio están presentes en cualquier máquina real, por lo que se evita tener que casarse con los detalles específicos de una arquitectura real dada.

[2] Aquí hablamos de lenguajes en el sentido formal de la teoría de autómatas. Para una introducción a la teoría de autómatas puede consultarse por ejemplo [Mar10; Koz97; HU79].

[3] En el capítulo 2 daremos una definición de un compilador como se suele utilizar en el área de compiladores.

[4] En el capítulo 2 estudiaremos cómo un compilador lleva a cabo la traducción correspondiente.

[5] Debemos tener claro, que de manera estricta L es un conjunto de cadenas que representan a los números decimales y no el conjunto de los números decimales. Lo análogo sucede para L'.

[6] Al no tener efectos laterales.

- Permite un estudio y razonamiento más simple y claro (del programa a ejecutar y de la máquina).
- Sirve como punto de referencia para comparar su comportamiento con arquitecturas reales.
- Da mayor libertad de experimentar al añadir, eliminar o modificar elementos de la máquina o al modificar el comportamiento de ésta.
- Como consecuencia de lo anterior una máquina virtual[7] puede influir o servir como base para incluir características en una máquina real, como en efecto ha sucedido en diversas ocasiones.
- Su código puede ser visto como una representación intermedia, a partir de la cual, se puede generar código de una (o varias[8]) máquina(s) real(es).

Sus desventajas más relevantes son:

1. Tener que agregar una capa de software a la ejecución del programa. Lo anterior significa trabajo de implementación para codificarla.
2. Degrada el desempeño en tiempo de ejecución.

De éstas, la segunda es su principal desventaja en comparación con una máquina real, aunque ésta en principio se puede remediar con la última de sus ventajas mencionadas, es decir, el código de la máquina abstracta puede ser visto como una representación intermedia y como en principio este código es similar al de una máquina real, se puede expandir al de esta última y así mejorar el desempeño en tiempo de ejecución.

Aquí como nuestros principales intereses son: el estudio, experimentación, claridad y punto de partida para futuras extensiones; es claro que, nos es conveniente trabajar con una máquina abstracta y así lo haremos.

Para verificar la corrección de un compilador se debe formular un teorema de corrección y presentar una demostración de este teorema.

Hacer una demostración sólo tiene sentido dentro de un sistema formal.

Un sistema formal[9] está conformado por:

- Un alfabeto de símbolos

[7] En la literatura el término *máquina virtual* suele utilizarse de distintas maneras [DHS00], una de ellas es como la implementación de una máquina abstracta; por otro lado también se utiliza de manera intercambiable con el término máquina abstracta. Como en el presente trabajo las máquinas abstractas con las que trabajaremos son máquinas de las que se puede hacer (y se han hecho implementaciones) utilizaremos ambos términos de manera indistinta.

[8] Un compilador que es capaz de generar código de diferentes máquinas se conoce como *reobjetivizable* (*retargetable*).

[9] En el presente trabajo utilizaremos los términos sistema formal y sistema de demostración de manera indistinta.

- Una gramática
- Un conjunto de axiomas
- Un conjunto de reglas de inferencia

Una demostración consiste entonces en una serie de pasos, en cada uno de los cuales a partir de los axiomas (o los teoremas previamente probados) se utiliza una regla de deducción sobre éstos hasta llegar al teorema enunciado.

En primer término podemos enunciar el teorema y hacer una demostración en papel y lápiz como es usual en matemáticas.

En matemáticas el sistema formal que se utiliza usualmente consiste en un lenguaje de primer orden y las reglas de deducción natural [Pra06].

No obstante, es fácil notar que al hacer una prueba en papel y lápiz se pueden cometer errores de varias formas, como son: aplicar una regla de manera incorrecta, dar por concedido que un teorema es válido aun cuando no lo es, entre muchos otros. Estos son errores que suceden en la vida real.

Podemos entonces ser ambiciosos y vislumbrar que podemos mecanizar este proceso. Si codificamos un sistema formal en un programa de computadora, entonces podemos utilizar esta aplicación para realizar la prueba indicando qué regla usar en cada uno de los pasos; bajo este esquema el programa se encargaría de verificar el buen uso de las reglas y que se usen sólo aquellos teoremas que han sido probados. En general el programa se encargaría de permitir un paso sólo cuando éste es correcto.[10] A este tipo de programas se les conoce como *asistente de pruebas* (*proof assistants*).

Es fácil notar que este enfoque es más seguro y confiable que el de utilizar papel y lápiz y por tanto es el que nosotros utilizaremos aquí.

1.1. Ejemplo motivacional

En el contexto de lenguajes de programación una estrategia de evaluación se refiere al mecanismo de paso de parámetros que un determinado lenguaje utiliza, dos de estos mecanismos son: llamada por valor (*call by value*) en el que los argumentos de una función se evalúan antes que el cuerpo de la función y llamada por necesidad (*call by need*) en el que un argumento sólo se evalúa cuando es necesario al evaluar el cuerpo de la función. Por otra parte el cálculo lambda se ha utilizado para modelar de forma abstracta las diferentes características presentes en los lenguajes de programación. Formalmente al utilizar el cálculo lambda como lenguaje de programación se habla de estrategias de reducción.[11] Intuitivamente una estrategia de reducción consiste en dado un término del

[10] Podemos ser aun más ambiciosos y enunciar al teorema y esperar que la aplicación encuentre la demostración por nosotros, lo anterior es posible para ciertos sistemas formales y no lo es para otros, esto es campo de estudio de la teoría de demostración (*proof theory*) [Dow99; Dow13]. En efecto existen este tipo de programas y se conocen como demostradores automáticos de teoremas (*automatic theorem provers*).

[11] Estudiaremos formalmente el cálculo lambda en la sección 2.2. Por otra parte veremos el cálculo lambda como lenguaje de programación y estrategias de reducción en el capítulo 2.2.

cálculo lambda elegir de manera determinista el siguiente subtérmino a evaluar.

Mencionamos que nuestro lenguaje fuente será un lenguaje funcional, hemos elegido que tenga como estrategia de evaluación llamada por valor (*call by value*), esto porque en primera instancia es más claro e intuitivo razonar con esta estrategia (para la mayoría de las personas) debido a que ésta la incluyen la mayor parte de los lenguajes de programación de amplio uso y también como primer paso en vías de que nuestro lenguaje sirva como referencia para extensiones futuras, en las que se pudiera incluir soporte para otro tipo de estrategias como llamada por necesidad (*call by need*).

Por tanto tenemos un lenguaje funcional con llamada por valor.

Además elegimos como punto de referencia el lenguaje ML ya que precisamente cuenta con estas características. Como motivación, deseamos que el lenguaje fuente de nuestro compilador sea un subconjunto de ML en el que se pueda expresar la función recursiva factorial.

Seleccionamos la función factorial debido a que es una de las funciones más representativas de las funciones recursivas y es ampliamente utilizada en la literatura de programación como ejemplo clásico al iniciar el estudio de recursión. Por otra parte la función factorial es simple y sencilla desde el punto de vista de programación ya que hace uso únicamente de operaciones aritméticas y usualmente se define como una relación de recurrencia en la que hay un único caso base. Dado lo anterior entre otras cosas se debe contar con enunciados condicionales para poder definir una función por casos.

```
let val rec fact =
     fn  x =>
        if  x = 0 then
            1
        else
            x * fact(x-1)
in
    fact(5)
end
```

Ejemplo 1.1: Factorial en ML.

Con base en lo anterior, nuestro lenguaje debe incluir: constantes (booleanas y enteras), variables, operaciones aritméticas y booleanas de comparación, declaraciones locales,[12] condicionales, funciones, funciones recursivas y aplicación.

[12]En realidad se puede prescindir de éstas pero se incluyen porque es muy común que los lenguajes funcionales cuenten con soporte para éstas. Por otro lado, también es común que se escriban programas haciendo uso de declaraciones locales.

1.2. Nuestro lenguaje

Habiendo fijado los enunciados con los que debe contar nuestro lenguaje a continuación definimos su sintaxis concreta por medio de una gramática en forma BNF.[13]

<exp>	::=	<eac>	Expresiones aritméticas y de comparación, declaraciones locales
	|	**if** <exp> **then** <exp> **else** <exp>	Enunciados condicionales
	|	**fn** <pat> **=>** <exp>	Abstracciones (funciones anónimas)
	|	**fix** ***name*** <pat> **=>** <exp>	Puntos fijos (funciones recursivas)
	|	<appexp> <fact>	Aplicaciones
	|	***bool***	Constantes booleanas
<pat>	::=	**(** ***name*** **)**	
	|	***name***	
<appexp>	::=	<fact>	
	|	<appexp> <fact>	
<eac>	::=	<comp>	
	|	<eac> **=** <comp>	Comparador de igualdad
<comp>	::=	<term>	
	|	<comp> **+** <term>	Adición
	|	<comp> **-** <term>	Substracción
<term>	::=	<fact>	
	|	<term> * <fact>	Multiplicación
<fact>	::=	***nat***	Constantes naturales
	|	***name***	Variables
	|	**let val name =** <exp> **in** <exp> **end**	Declaraciones locales
	|	**let val rec name =** <exp> **in** <exp> **end**	Declaraciones locales recursivas
	|	**(** <exp> **)**	

Figura 1.1: Gramática en BNF que genera la sintaxis concreta de nuestro lenguaje fuente.

Por otra parte también damos la sintaxis abstracta correspondiente a las expresiones aritméticas y de comparación y delegamos la presentación de la sintaxis abstracta de todo nuestro lenguaje para el capítulo 5 (una vez que hayamos estudiado los conceptos previos necesarios para definirla).

[13] Las palabras en negritas corresponden a los símbolos terminales de la gramática, es decir, como veremos en la sección 2.1.1 a los tokens que genera el analizador léxico.

e	::=	Const n	$n \in \mathbb{N}$	Constantes naturales
	|	Constb b	$b \in \mathbb{B}$	Constantes booleanas
	|	Op e e	Op$\in$\{Plus, Minus, Times, Eq\}	Expresiones aritméticas
v	::=	Num n	$n \in \mathbb{N}$	Valores naturales
	|	Bool b	$b \in \mathbb{B}$	Valores booleanos

Figura 1.2: Sintaxis abstracta de un subconjunto de nuestro lenguaje fuente.

en donde, e representa una expresión y v un valor (el resultado final de evaluar una expresión); por su parte Op representa a los operadores binarios aritméticos y booleanos de comparación.

En cuanto a la máquina abstracta que utilizaremos, mencionamos ya que hemos elegido un lenguaje con llamada por valor y debemos por tanto elegir una máquina abstracta que siga esta estrategia de evaluación. Dos de las máquinas más conocidas (y más estudiadas) que siguen esta estrategia son: la máquina SECD [Lan64] y la CAM (*Categorical Abstract Machine*) [CCM85]. De éstas hemos elegido la SECD[14] por su claridad y simplicidad y en particular porque permite estudiar conceptos fundamentales de manera simple y clara, como lo son la estrategia de llamada por valor y cerraduras. En efecto fue con esta máquina que Landin introdujo estos conceptos[15] en [Lan64]. En otras palabras, esta máquina es ideal para estudiar, experimentar y hacer extensiones; adicionalmente sirve como punto de referencia para compararla con otras máquinas, que es justo lo que deseamos en este trabajo.

Por otra parte, en lo que al asistente de pruebas se refiere, Edinburgh LCF[16] [GMW79] fue uno de los primeros y más influyentes. El sistema formal de LCF está basado en (*Logic for Computable Functions*) un lógica introducida por Danna Scott en [Sco93].

Más tarde Thierry Coquand y Gerard Huet [CH86] en el INRIA desarrollaron el cálculo de construcciones, que es un cálculo lambda tipificado que extiende el sistema F de Girard [Gir71; Gir72; GTL89] y el sistema de la teoría de tipos de Per Martin Löf [Löf71], por tanto dio como resultado un cálculo con una expresividad amplia en el que entre otras cosas se podían expresar proposiciones lógico matemáticas por un lado y escribir programas por el otro. Pero más aún como tanto proposiciones como programas se podían expresar en dicho cálculo, también se pueden expresar directamente propiedades acerca de los programas y en su caso sus respectivas demostraciones. Y luego, debido al isomorfismo de Curry-Howard verificar que una demostración es correcta consiste en realizar verificación de tipos en el cálculo de construcciones. Después Huet y colaboradores [Dow+93; Hue89] llevaron a la práctica dicho cálculo utilizándolo como sistema

[14]En realidad trabajaremos con una variación introducida por Leroy [Ler16b] a la que denomina *Modern SECD* y que estudiaremos en el capítulo 5.

[15]La estrategia de llamada por valor había estado presente en los lenguajes de la época pero no se había formalizado para su estudio, Landin la introduce implícitamente porque es la forma de evaluar en la máquina.

[16]Que fue el sucesor de Stanford LCF, en adelante nos referiremos a Edinburgh LCF simplemente como LCF.

formal de un nuevo asistente de pruebas basado en la tradición de LCF al que nombraron Coq [Tea14; BC04; Chl13; Pie+15]. Desde entonces Coq ha sido desarrollado en el INRIA, añadiéndole a lo largo de los años numerosas características y mejoras. Una de las mejoras más significativas fue el agregar definiciones inductivas al cálculo de construcciones [Pau96; Pau14; Wer94] y de esta manera nació el cálculo de construcciones inductivas (*Calculus of Inductive Constructions* CIC) que es el cálculo que utiliza actualmente Coq. Estudiaremos más a detalle Coq en el capítulo 3.

1.3. Estrategia de estudio

A continuación delineamos la estrategia de estudio que iremos desarrollando durante los siguientes capítulos.

- Primero en la sección 2.1 diremos qué es un compilador y cómo trabaja, para ello presentaremos algunas definiciones y conceptos básicos. Adicionalmente daremos una descripción breve de cada una de las etapas de un compilador tradicional y las herramientas que en ellas se utilizan. Para ejemplificar el proceso de compilación utilizaremos un pequeño lenguaje imperativo.

- Por otra parte como el lenguaje fuente de nuestro compilador es un lenguaje funcional para poder comprender su funcionamiento y las propiedades que posee es necesario partir del cálculo lambda y luego ver cómo es que se puede utilizar el cálculo lambda como lenguaje de programación. Por otro lado para poder realizar la demostración de la corrección de nuestro compilador, es de ayuda saber qué es un sistema de demostración y cuál es el que nosotros utilizaremos. El sistema de demostración que se utiliza comúnmente es el sistema de deducción natural, sin embargo el cálculo lambda también se puede utilizar como sistema de demostración.[17] En efecto el sistema de demostración que utiliza Coq es el cálculo lambda adicionado con tipos y otras extensiones al que se le llama cálculo de construcciones inductivas.[18] Entonces el cálculo lambda se puede utilizar por una parte como lenguaje de programación y por otra como sistema de demostración es por ello que nosotros lo introduciremos en la sección 2.2 y daremos un indicio de cómo se puede utilizar como sistema de demostración.[19] Más adelante en la sección 4.1 estudiaremos su uso como lenguaje de programación. De hecho al cálculo de construcciones

[17] De hecho este era su objetivo original ser utilizado como una lógica para el fundamento de las matemáticas.

[18] Podemos ver con esto que esta evolución del cálculo lambda no está alejada de su objetivo original y que de cierta forma y con cierta medida lo ha cumplido.

[19] No es nuestra intención adentrarnos en el estudio del cálculo lambda y sus diversas extensiones para ser utilizado como sistema de demostración, pues esto queda fuera de nuestros objetivos. Simplemente deseamos tener una noción de la teoría que está detrás al utilizar Coq que nos ayudará a comprender el por qué Coq cuenta con algunas de sus características con las que nos encontraremos al desarrollar nuestro compilador.

inductivas se le dan estos dos usos, es por ello que en Coq por una parte se pueden escribir programas y por otra se pueden realizar demostraciones y más aún se pueden realizar demostraciones acerca de programas, es decir ambos usos están mezclados.

- En el ámbito de los compiladores tradicionales de lenguajes imperativos de amplio uso en la vida real (como C o C++) usualmente el escritor de compiladores consulta la especificación del lenguaje y con base en ello desarrolla el compilador. En esta especificación es común que la sintaxis del lenguaje se defina formalmente mediante una gramática libre del contexto. En cuanto a la semántica lo usual es que se den descripciones informales en lenguaje natural, esto sin duda es indeseable porque puede dar lugar a ambigüedades o vacíos en la especificación y esto puede conducir a que diferentes compiladores de un mismo lenguaje generen código que se comporte distinto para un mismo programa. Lamentablemente este es el caso para lenguajes como C o C++. Lo ideal entonces es contar con un medio que permita especificar la semántica de un lenguaje de manera formal que se utilice en la especificación del lenguaje.[20] Se han desarrollado diferentes métodos para especificar formalmente la semántica de un lenguaje de programación, los más consolidados son: semántica operacional, semántica denotativa y semántica axiomática. De éstos la semántica operacional dada su naturaleza es la de más ayuda para guiar una implementación, es por eso que es la que se esperaría que se incluyera en una especificación de un lenguaje de programación para que los escritores de compiladores pudieran consultarla.

  Originalmente la semántica operacional se concibió con la idea de mostrar el comportamiento de cada uno de los enunciados que forman parte de un lenguaje, por ejemplo un enunciado if o una secuencia. Como en principio para este fin se tendría que especificar el comportamiento del lenguaje para cada una de las máquinas reales, entonces se vislumbró abstraer las características que tiene toda máquina real en una máquina abstracta y entonces bastaría con mostrar el comportamiento de cada uno de los enunciados soportados por un lenguaje en esa máquina abstracta. Luego Plotkin vislumbró que se podía especificar el comportamiento de un enunciado directamente en términos del enunciado sin utilizar una máquina abstracta, así llegó a lo que se conoce como semántica operacional de paso pequeño. Después Khan con base en el trabajo de Plotkin descubrió que la semántica se puede especificar como una lógica de manera similar a como se especifica el sistema de deducción natural y con eso dio origen a llamada semántica natural o semántica de paso grande.

  Nosotros desde luego como deseamos formalizar la corrección de nuestro compilador necesitamos especificar formalmente la semántica del lenguaje fuente y la de la máquina objetivo. Es por ello que estudiaremos la semántica formal en la sección

[20] Desafortunadamente al parecer la única especificación de un lenguaje en amplio uso cuya semántica está definida formalmente es la de ML [MTM97].

2.3, en particular estudiaremos la semántica de paso pequeño y la semántica natural. Por otro lado, también debemos especificar formalmente la compilación, para hacerlo es fácil notar que podemos especificarla mediante una función. Sin embargo vale la pena resaltar que descubriremos que podemos utilizar la semántica natural no solo para especificar la semántica de un lenguaje o de una máquina sino también para formalizar la compilación de un lenguaje a otro. De hecho, nosotros usaremos la semántica natural para especificar una de las etapas de nuestro compilador a saber la compilación de nuestro lenguaje fuente a notación de De Bruijn.

- El capítulo 3 lo dedicaremos a estudiar los detalles técnicos de Coq y algunas de sus características con las que nos encontramos al desarrollar nuestro compilador, dos de las más relevantes son:

    1. Recursión estructural basada en sintaxis. El cálculo lambda simplemente tipificado cumple con el teorema de normalización fuerte, lo que intuitivamente significa que si utilizamos este cálculo como lenguaje de programación entonces todo programa que escribamos en él necesariamente termina, el cálculo de construcciones inductivas que es una extensión del cálculo lambda simplemente tipificado también cumple con este teorema. Por esta razón todo programa que se escribe en Coq necesariamente debe terminar. Entonces, en particular cuando se escribe una función recursiva en Coq debe haber un mecanismo que permita garantizar que dicha función terminará. El mecanismo con el que cuenta Coq por omisión es recursión estructural basada en sintaxis, esto es siempre que se define una función recursiva en Coq se debe especificar para todos casos posibles de entrada y las llamadas recursivas deben ser estrictamente sobre una subestructura del argumento de entrada sobre el cual se está haciendo la recursión. Este mecanismo en la práctica no siempre es el adecuado pues existen diversas funciones recursivas que en efecto terminan pero que no están basadas en recursión estructural, en estos casos es un reto poder expresar en Coq que la función termina. Es por eso que se han investigado otros posibles mecanismos para expresar que una función recursiva en Coq termina, sin embargo a la fecha ninguno de éstos ha sido plenamente satisfactorio ni adoptado ampliamente. La solución pragmática que se utiliza en estos casos es simplemente agregar un número natural como argumento de la función y hacer la recursión sobre él, de esta manera se garantiza que la función siempre terminará pues cuando se ejecute la función la profundidad de las llamadas recursivas que se realicen siempre será a lo más el número natural que se pase como argumento. Sin embargo esta solución no es plenamente satisfactoria entre otras cosas porque se debe agregar un nuevo argumento que nada tiene que ver con la función y que al hacerlo se tienen que realizar cálculos innecesarios con la correspondiente sobrecarga que eso genera. Esta solución se conoce como recursión acotada y también sirve para escribir funciones en Coq que originalmente no siempre terminan, pues de esta manera se garantiza que siempre terminarán, esto en

el contexto de las máquinas de Turing burdamente equivaldría a acotar el número de pasos que pueda dar una máquina, con esto aunque se garantiza que siempre termina no implica que sea con el resultado que se espera. Aunque si en efecto si la función siempre que termina es con el resultado que se espera se puede probar en Coq mediante un lema aparte.

2. Mecanismo de extracción de programas. Este mecanismo se refiere a que en ciertos casos se puede obtener a partir de un desarrollo en Coq un programa escrito en un lenguaje de programación convencional como Haskell u OCaml listo para ser utilizado en la vida real. El funcionamiento de la extracción de programas depende de cómo se utilice Coq. A continuación damos una breve explicación:

   a) Si se usa como sistema de demostración entonces se debe enunciar una proposición y con ayuda de las tácticas de Coq construir la demostración de ésta. Bajo este esquema no se extrae ningún programa, o decimos que el desarrollo no tiene contenido computacional. En efecto, un usuario no esperaría obtener un programa en este caso, pues su objetivo únicamente radica en realizar una demostración de que se cumple una proposición, por ejemplo, la asociatividad de los números naturales.

   b) Si se usa como lenguaje de programación entonces lo que se espera es que un usuario escriba un programa en Coq por ejemplo la función doble que calcula el doble de un número natural. En este caso si se utiliza el mecanismo de extracción lo que se obtiene es un programa equivalente al escrito en Coq pero expresado en un lenguaje de programación convencional, esto es, para nuestro ejemplo se obtendría la función doble escrita en el lenguaje digamos OCaml. Bajo este escenario podemos decir que sí se tiene contenido computacional.[21] Con este enfoque lo que se está haciendo es únicamente utilizar Coq como si se tratase de un lenguaje de programación y luego pedir que convierta nuestro programa de Coq a OCaml.
      Aunque utilizar Coq meramente como lenguaje de programación es posible, es más sensato que se utilice para demostrar que un programa cumple con cierta propiedad. Entonces, con esta finalidad lo que se haría es:

      I. escribir un programa (que tiene contenido computacional)

      II. enunciar la propiedad y realizar la demostración de ésta (que no tiene contenido computacional).

      De esta forma se garantiza que el programa escrito cumple con la propiedad enunciada. Ahora con miras de utilizarse en la vida real se quisiera contar con el programa pero escrito en lenguaje de programación convencional, que es justo lo que podemos obtener utilizando el mecanismo de extracción sobre el programa, esto es, el programa que el usuario escribió

[21] O sea el contenido computacional se refiere al hecho de que se puede obtener un programa escrito en un lenguaje de programación convencional a partir de un desarrollo de Coq.

en Coq ahora se obtendría en OCaml. Nótese cómo no nos interesa obtener en absoluto algún contenido computacional de la demostración de la propiedad, lo único relevante es que ya se tiene la certeza que el programa la cumple. En este caso nosotros diremos que se cuenta con un programa verificado.
Nosotros seguiremos esta estrategia para realizar la verificación de nuestro compilador.

c) Como sistema de demostración e implícitamente como lenguaje de programación (al mismo tiempo). Este uso se puede dar cuando se desea demostrar que existe un objeto que cumple cierta propiedad. En este caso primero se debe enunciar que existe un objeto tal que cumple cierta propiedad.[22] Entonces podemos considerar que la demostración constará de dos partes:

   I. Una en la que implícitamente se debe dar un algoritmo que calcule el objeto que se está afirmando que existe (esta parte tiene contenido computacional) y

   II. otra dedicada a demostrar que el objeto cumple la propiedad (esta parte no tiene contenido computacional)

   esta demostración se lleva a cabo con ayuda de las tácticas de Coq. A la parte 2.c.II. se le conoce como certificado pues como su nombre lo indica certifica que el objeto en efecto cumple con la propiedad. Cuando se realiza la extracción de programas sobre uno de estos desarrollos lo que se obtiene es el programa en OCaml correspondiente al algoritmo que calcula el objeto que cumple la propiedad, es decir, el mecanismo de extracción obtiene el contenido computacional de la primera parte y la segunda simplemente la descarta pues no tiene contenido computacional. En este caso se dice que se cuenta con un programa certificado.
   Vale la pena mencionar algunas diferencias entre un programa certificado y uno verificado:

   I. Para obtener un programa verificado primero se escribe por una parte el programa y por otra se enuncia y se demuestra la propiedad que el programa cumple. En contraposición para obtener un programa certificado se enuncia una propiedad de la forma existe un objeto tal que cumple la propiedad y al realizarse la demostración podemos considerar que se está escribiendo el programa y la demostración juntos.

   II. Para el programa verificado se escribe un programa en Coq explícitamente (sin ayuda de las tácticas de Coq) utilizando la sintaxis específica para este fin. En cambio para un programa certificado se puede considerar que se escribe el programa de forma implícita al realizar la demostración con ayuda de las tácticas de Coq.

[22] Utilizando una característica con la que cuenta Coq que es el uso de tipos dependientes.

III. El programa escrito explícitamente en Coq en el caso de verificación Coq lo puede ejecutar directamente, es decir, se puede utilizar Coq como si fuese un intérprete.[23] En comparación cuando se realiza certificación Coq también es capaz de ejecutar el programa implícito y calcular su resultado pero además pone información correspondiente a la parte de la demostración.

Nosotros decidimos utilizar verificación por la siguientes razones:

a) Es más directo e intuitivo escribir explícitamente el programa a hacerlo implícitamente con ayuda de las tácticas de Coq.
b) Nos parece más limpio y claro escribir por una parte el programa y por otra la demostración de la propiedad que éste cumple.
c) Se puede ejecutar el programa dentro de Coq obteniendo únicamente el resultado (sin información adicional de la demostración) del programa que es lo que nos interesa.
d) Realizar certificación implica utilizar una característica avanzada de Coq conocida como tipos dependientes. Esta característica ha resultado difícil de dominar para los usuarios y la documentación de ésta es escasa y muy específica.

En la sección 3.7 presentamos un ejemplo de un compilador verificado para un pequeño lenguaje de expresiones aritméticas en Coq, esto con el objetivo de introducir e ilustrar de manera simple todos los pasos necesarios para realizar la verificación de un compilador en Coq. Conocimientos que posteriormente damos por conocidos al realizar la verificación de nuestro compilador principal en el capítulo 5.

- En el capítulo 4 estudiaremos cómo a partir del cálculo lambda se puede llegar a un lenguaje de programación. A continuación describimos algunos de los pasos que se deben seguir con este fin:

  1. Se deben agregar primitivamente al cálculo las constantes con las que se desea que cuente el lenguaje por ejemplo naturales y booleanos. También se debe extender con los operadores y funciones primitivas que se deseen considerar por ejemplo, operadores aritméticos y binarios de comparación.
  2. En el cálculo lambda intuitivamente realizar una contracción β significa dar un paso en la ejecución de un programa. Por otra parte el cálculo lambda cumple con el teorema de estandarización lo que intuitivamente significa que si un programa siempre termina (eligiendo de forma no determinista la siguiente contracción β a realizar) entonces también termina siguiendo una estrategia de reducción conocida como reducción en forma normal (en la cual se elige de forma determinista la siguiente contracción β a realizar que es la de la extrema

[23] En realidad Coq lo puede compilar a una máquina virtual y ejecutarlo sobre ésta con la única finalidad de que se ejecute de manera más eficiente.

izquierda y más externa). El que se siga una estrategia de reducción determinista es de gran ayuda a los programadores para poder razonar y saber cómo se comportará un programa. Landin fue el primero en formalizar (por medio de una máquina) la estrategia de evaluación conocida como llamada por valor, luego Plotkin la formalizó puramente en términos del cálculo lambda.[24]

Cabe mencionar que diferentes estrategias de evaluación puedan dar diferentes resultados en un mismo programa, por ejemplo para un programa determinado una puede terminar devolviendo un valor final en pocos pasos mientras que si se sigue otra la ejecución del programa no terminará.

Entonces establecer una estrategia de evaluación es necesario en el camino de usar el cálculo lambda como lenguaje de programación. Nosotros utilizaremos llamada por valor porque es simple e intuitiva.

3. Uno de los criterios claves en el ámbito de los compiladores es la eficiencia. En efecto uno de los criterios más importantes para decidir hacer un compilador de un lenguaje y no un intérprete es la eficiencia. En el caso general el ejecutable de un programa generado por un compilador será más eficiente que interpretar el programa. Por otra parte hoy en día la mayoría de los compiladores son o buscan ser compiladores optimizadores, es decir, buscan realizar una o varias etapas de optimización con el objetivo de obtener un ejecutable más eficiente.

   Tomando en cuenta este criterio nosotros siempre que se nos presente la oportunidad de elegir una solución seguiremos aquella que sea más eficiente.

   Veremos que cuando se evalúa un programa se puede hacer mediante sustituciones o mediante entornos y cerraduras, de estas dos opciones, utilizar entornos y cerraduras es más eficiente, por eso es la que nosotros utilizaremos.

4. Cuando se hace uso de entornos y cerraduras para evaluar un programa (en el que se usa notación convencional para las variables, es decir, utilizando nombres para éstas), es necesario hacer búsquedas en el entorno cuando se requiere obtener o modificar el valor de una variable. Para el cálculo lambda existe una notación en la que no se utilizan nombres paras las variables en su lugar se utilizan índices, esta notación se conoce como índices de De Bruijn. Cuando se utiliza la notación de De Bruijn (junto con entornos y cerraduras) para evaluar un programa no es necesario realizar búsquedas en el entorno para manejar los valores de las variables, esto porque el índice de De Bruijn que representa a una variable coincide exactamente con el índice que ocupa dicha variable en el entorno, por tanto cuando es necesario obtener o modificar el valor de la variable simplemente se consulta la posición del entorno señalada por el índice. Desde luego como no es necesario hacer búsquedas esta solución es más eficiente. Por lo que nosotros decidimos utilizarla con base en nuestro criterio de eficiencia.

   No obstante resulta insensato exigir que un programador escriba un programa utilizando índices (que siguen ciertas reglas) para expresar las variables.

[24] También formalizó la estrategia de llamada por nombre.

Por tanto lo que sí podemos hacer es agregar una etapa a nuestro compilador[25] que convierta (compile) cualquier programa que utilice la notación convencional basada en nombres a índices de De Bruijn. Con esto el programador podrá escribir un programa con la notación convencional y la ejecución será más eficiente.

Por otra parte vale la pena mencionar que un problema muy común que se puede presentar al realizar sustituciones en el cálculo lambda es el problema de captura de variables. Existen diferentes métodos posibles para evitar este problema dos de éstos son:

a) Utilizar una estrategia de reducción débil
b) Utilizar índices de De Bruijn

Nuestro desarrollo es inmune a este problema porque utilizamos la estrategia de llamada por valor que es un caso particular de reducción débil. Así, aunque nosotros no utilizamos sustituciones podríamos preguntarnos si se da un problema análogo al de captura de variables al utilizar entornos y cerraduras, pero como no existe posibilidad de este problema en nuestro trabajo esta pregunta simplemente no nos ocupa. Obsérvese que nosotros no utilizamos índices de De Bruijn para evitar este problema ya que nuestro desarrollo no es susceptible de tenerlo, utilizamos índices de De Bruijn para tener una evaluación más eficiente de los programas generados por nuestro compilador.

5. El cálculo lambda es Turing computable (de acuerdo a la tesis de Church-Turing), lo que llama la atención es que a priori el cálculo lambda debido a su simpleza no pareciera contar con soporte de recursión. Originalmente lo que se hace en el cálculo lambda para obtener recursión es formular un término que cumple cierta propiedad que garantiza que se puede utilizar como auxiliar en la definición de una función recursiva, a éste se le conoce como combinador Y. Nótese que bajo este escenario no se ha agregado nada primitivamente al cálculo, el combinador Y puede considerarse como un programa que cualquier programador puede escribir. No obstante evaluar un programa que hace uso del combinador Y conlleva la sobrecarga introducida debida a la evaluación de éste.

   Existe otra posible solución a saber agregar primitivamente al cálculo un operador de punto fijo que se encargue de ofrecer soporte nativo de funciones recursivas, este operador se conoce como operador de punto fijo μ. La evaluación de este operador involucra menos cálculos y por tanto es más eficiente que utilizar el combinador Y. Es por eso que nosotros utilizaremos esta solución.

- En este punto contaremos ya con los conocimientos necesarios para construir nuestro lenguaje fuente a partir del cálculo lambda y expresar su semántica haciendo uso de entornos y cerraduras.

[25] Que puede ser considerada en sí misma un compilador de notación basada en nombres para las variables a índices de De Bruijn.

Por otra parte sabemos que nuestro compilador consistirá de dos etapas:

1. Compilación del lenguaje fuente a índices de De Bruijn.
2. Compilación de índices de De Bruijn a código de máquina de la SECD moderna.

Adicionalmente conoceremos también los detalles técnicos y los pasos necesarios para realizar la verificación de la corrección de un compilador con base en el ejemplo del compilador correcto que desarrollamos en la sección 3.7.

La idea entonces es seguir la misma estrategia de nuestro ejemplo de la sección 3.7 pero ahora en nuestro compilador principal.

Tomando en cuenta lo anterior en el capítulo 5 presentaremos lo siguiente:

1. Sintaxis abstracta de nuestro lenguaje fuente.
2. Semántica natural de nuestro lenguaje haciendo uso de entornos y cerraduras.
3. Sintaxis abstracta de nuestro lenguaje fuente utilizando índices de De Bruijn.
4. Semántica natural de nuestro lenguaje con índices de De Bruijn haciendo uso de entornos y cerraduras.
5. Compilación de nuestro lenguaje fuente a índices de De Bruijn.
6. Teorema de corrección de la compilación de nuestro lenguaje fuente a índices de De Bruijn.
7. Sintaxis abstracta de las instrucciones de la máquina SECD moderna.
8. Semántica de paso pequeño de la máquina SECD moderna.
9. Semántica de paso grande de la máquina SECD moderna.
10. Lema de equivalencia entre ambas semánticas de la máquina.
11. Compilación de nuestro lenguaje fuente con notación de De Bruijn a código de máquina de la SECD moderna.
12. Teorema de corrección de la compilación de nuestro lenguaje fuente a código de máquina de la SECD moderna utilizando la semántica de paso pequeño de la máquina.
13. Teorema de corrección de la compilación de nuestro lenguaje fuente a código de máquina de la SECD moderna utilizando la semántica de paso grande de la máquina.

En cada uno de los puntos anteriores además se muestra cómo se expresa dicho punto en Coq. Por ejemplo, se presenta la sintaxis abstracta del lenguaje e inmediatamente después cómo se escribe ésta en Coq. Para el caso de los teoremas de

corrección además de mostrar cómo se enuncian en Coq también se dan sus demostraciones en éste.[26]

Notemos además que se ofrecen dos semánticas de las máquina SECD moderna esto no era estrictamente necesario pues bastaría con utilizar una. En la literatura es más común que se utilice la semántica de paso pequeño para especificar la semántica de una máquina. No obstante durante el desarrollo del presente trabajo seguimos la estrategia de utilizar la semántica natural o de paso grande siempre que fuera de ayuda y se tuviera como alternativa, entre otras cosas por considerarla simple e intuitiva y por su concepción unificadora (que explicaremos en la sección 2.3.2), es por eso que desarrollamos una semántica natural para la máquina SECD moderna. Originalmente la semántica de la máquina SECD moderna era una semántica de paso pequeño, así en vías de mostrar que nuestra nueva semántica era equivalente a la especificación original realizamos la demostración de que esta equivalencia se cumple.

- Por último en el capítulo 6 presentamos nuestras conclusiones y el trabajo relacionado.

Habiendo mostrado nuestra estrategia de estudio comenzamos con ella en el siguiente capítulo.

[26] En realidad por cuestiones de espacio sólo se muestra en este texto parte del desarrollo en Coq para cada uno de los puntos, para ver el desarrollo completo consúltese [Zúñ15b].

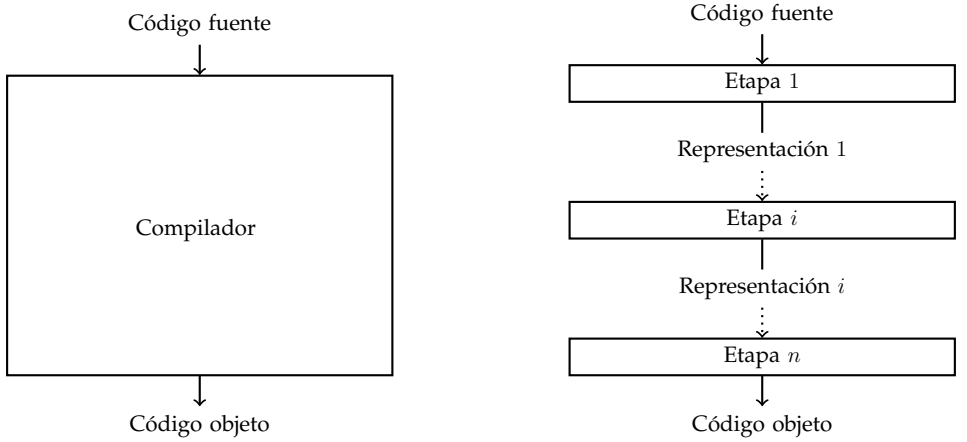
CAPÍTULO 2

Conceptos preliminares

2.1. Compiladores

Un *compilador* es un programa que toma como entrada un programa p escrito en un lenguaje L y produce como salida un programa p' escrito en un lenguaje L' donde p' corresponde a la traducción de p en L'.

A L se le denomina *lenguaje fuente*, a L' *lenguaje objetivo*, a p *código fuente* y a p' *código objeto*.



(a) Proceso de compilación visto como una caja negra.

(b) Proceso de compilación como secuencia de etapas.

Figura 2.1: Proceso de compilación.

Podemos visualizar el proceso de compilación como una caja negra tal como se muestra en la figura 2.1a. Sin embargo, como este proceso es lo suficientemente complejo para ser realizado como una única operación, se suele dividir en fases o etapas (como se ilustra en la figura 2.1b).

Una *fase o etapa* es una operación de alto nivel de abstracción que toma como entrada una representación del código fuente y produce como salida otra representación (que refleja el resultado de haber realizado la operación).

En una etapa se pueden realizar una o varias transformaciones sobre el código fuente, aunque generalmente sólo se realiza una, por lo que los términos etapa y transformación se suelen usar de manera indistinta.

En una implementación un *recorrido de inicio a fin* (*pass*) es un módulo que agrupa una o más etapas.

Una recorrido de inicio a fin hace una lectura completa de una representación del código fuente, realiza las transformaciones que dictan cada una de sus etapas y escribe como salida otra representación (que reflejan el resultado de las transformaciones hechas).

El número de etapas (y de representaciones) de un compilador dependerá del lenguaje fuente, del lenguaje objeto, de si se realizan optimizaciones o no y del diseño del compilador.

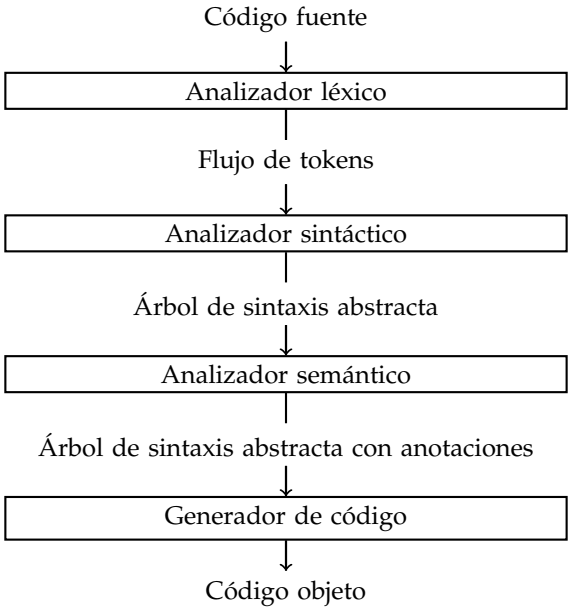


Figura 2.2: Compilador con las etapas clásicas.

No obstante, tradicionalmente un compilador cuenta con al menos las siguientes etapas (que se ilustran en la figura 2.2):

1. Análisis léxico.

2. Análisis sintáctico.

3. Análisis semántico.

4. Generación de código.

A continuación presentamos una breve descripción de cada una de las etapas y para ilustrar su funcionamiento y las transformaciones que realizan utilizaremos un ejemplo simple pero ilustrativo de un compilador tradicional.

Supongamos que queremos compilar programas similares al siguiente:

```
int x;
x = 2;
int y =    7;
5+x*y;
```

Ejemplo 2.1: Programa con asignaciones y expresiones aritméticas.

es decir, el lenguaje fuente debe contemplar: definiciones, declaraciones, asignaciones, expresiones aritméticas y secuencia; además únicamente se soportan los tipos int y float. Solamente se permiten operaciones y asignaciones entre entidades del mismo tipo, por ejemplo, 5 + 3.2 es incorrecto, lo mismo que float x = 3 mientras que 5 + 3 y float x = 3.0 son correctos.

2.1.1. Análisis léxico

Un *token* es un par (nombre,atributo) donde, nombre es un símbolo terminal de la gramática que define una sintaxis del lenguaje y atributo es un atributo opcional que algunos tokens tienen, es decir, cuando un token no sólo sirve para verificar que se cumpla la sintaxis, sino que además es relevante el atributo asociado a él para ser usado en las siguientes fases del proceso de compilación.

Un token representa un conjunto de cadenas válidas correspondientes a un símbolo terminal de una gramática, por ejemplo, **name** es un símbolo terminal de una gramática y tmp es una cadena válida asociada al token (**name**,□).[1] Como el conjunto representado por un token es regular se puede expresar mediante una expresión regular y se puede reconocer con un autómata finito.

Un *lexema* es una (sub)-cadena presente en el código fuente que corresponde a un token.

[1]Utilizaremos el símbolo "□" para denotar que en ese caso el token no tiene un atributo opcional o dicho de otra forma que el atributo es vacío.

El análisis léxico se encarga de leer el código fuente (que puede ser visto como un flujo de caracteres) los agrupa en lexemas y va revisando a qué token corresponde cada lexema (si un lexema no corresponde a ningún token envía un error), generando así como salida una secuencia de tokens. Adicionalmente, limpia el código de caracteres irrelevantes como espacios en blanco, saltos de línea y comentarios, es decir, reconoce estos lexemas y simplemente los descarta.

Para nuestro código de ejemplo 2.1, la salida del analizador léxico es:

(**INT**,□),(**NAME**,*x*),(**SCN**,□),(**NAME**,*x*),(**ASG**,□),(**INUM**,2),(**SCN**,□),(**INT**,□),(**NAME**,y),
(**ASG**,□),(**INUM**,*7*),(**SCN**,□),(**INUM**,5),(**PLUS**,□),(**NAME**,*x*),(**TIMES**,□),(**NAME**,y),(**SCN**,□)

Figura 2.3: Salida del analizador léxico

Nótese por ejemplo, que para 5 se regresa el token (**INUM**,5) mientras que para ";" (**SCN**,□), en el primer caso no sólo basta con que el analizador sintáctico encuentre el símbolo terminal **INUM** sino que, por ejemplo, el generador de código debe saber que se trata de un 5, mientras que para un ";" basta que el analizador sintáctico encuentre el símbolo terminal **SCN** (que es el que representa a ";" en la gramática del lenguaje).

La implementación de un analizador léxico, entonces, consiste en hacer un reconocedor que detecte todas las cadenas válidas (lexemas) asociadas a cada token y reporte cada presencia de un token al analizador sintáctico.

Pero como dijimos, para cada token esto se puede expresar mediante una expresión regular, de esta expresión se puede obtener un autómata finito, y por tanto, este proceso se puede automatizar [Aho+06]. De esta forma, para cada token, basta con dar una expresión regular y la acción a realizar cuando se encuentre en la entrada un lexema correspondiente a dicho token.

Lo anterior es justamente lo que hacen los generadores de analizadores léxicos, uno de los primeros y más influyentes de estos programas es *lex* [Les75; LS90], uno más reciente y quizás el más utilizado hoy en día es *flex* [PEM12].

A continuación presentamos en la figura 2.4 cada uno de los tokens junto con la correspondiente expresión regular que los denota para nuestro lenguaje de ejemplo.[2]

[2]Aquí utilizamos por un lado la notación de expresiones regulares extendidas y por otro la notación de *definiciones regulares* (*regular definitions*) ver [Aho+06]. En las definiciones regulares del lado izquierdo de "→" se escribe el nombre del token y del lado derecho la correspondiente expresión regular que lo denota.

plus	$\rightarrow$	+	Símbolo de adición
times	$\rightarrow$	*	Símbolo de producto
asg	$\rightarrow$	=	Símbolo de asignación
scn	$\rightarrow$	;	Punto y coma
intd	$\rightarrow$	int	
floatd	$\rightarrow$	$float$	
inum	$\rightarrow$	$[0-9]^+$	Enteros
fnum	$\rightarrow$	$(\sim?)(\textbf{\textit{inum}}(.\textbf{\textit{inum}})?([eE](\sim?)(\textbf{\textit{inum}}))$	Flotantes
		$\mid(\textbf{\textit{inum}}(.\textbf{\textit{inum}})([eE](\sim?)(\textbf{\textit{inum}}))?))$	
name	$\rightarrow$	$[A-Za-z][A-Za-z'\_0-9]^*$	Variables

Figura 2.4: Tokens de nuestro lenguaje de ejemplo.

La especificación del analizador léxico de nuestro lenguaje escrita en *flex*, es la siguiente:

```
CH     [A-Za-z'_0-9]
NAME [A-Za-z]{CH}*
INUM [0-9]+
EXPO [eE](~?){INUM}
FRAC "."{INUM}
FNUM  (~?)(({INUM}{FRAC}?{EXPO})|({INUM}{FRAC}{EXPO}?))
TIMES \*
PLUS \+
LP \(
RP \)
ASGS \=
INTD int
FLOATD float
SCN ;
WS [ \n\t]
%%

{PLUS}     {return PLUS;}
{TIMES}    {return TIMES;}
{LP}       {return LP;}
{RP}       {return RP;}
{ASGS}     {return ASGS;}
{SCN}      {return SCN;}
{INTD}     {return INTD;}
{FLOATD}   {return FLOATD;}
{INUM}     {yylval.inum = atoi(yytext); return INUM;}
{FNUM}     {yylval.fnum = atof(yytext); return FNUM;}
{NAME}     {yylval.name = new string(yytext); return NAME;}
{WS}       {}
```

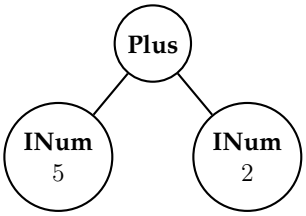
En la primera sección antes de `%%`, se le asigna un nombre a cada expresión regular, mientras que en la segunda (después de `%%`), en cada línea se escribe primero el nombre de la expresión regular y luego la acción que se debe realizar cuando se encuentra un lexema en la entrada que pertenezca al conjunto representado por dicha expresión.

Nótese cómo se descartan los espacios en blanco (expresión regular `WS`) y para `INUM` el atributo[3] se coloca en la variable `yylval` que es donde la espera el analizador sintáctico.

2.1.2. Análisis sintáctico

Hace falta verificar que nuestro programa sea un programa bien formado, es decir, que pertenezca a nuestro lenguaje fuente. Este último lo podemos especificar mediante una gramática libre del contexto. Así el conjunto de cadenas generado por la gramática será el conjunto de programas válidos, a este conjunto de cadenas se le denomina *sintaxis concreta*, es decir, a los programas válidos tal como los escribe el programador.

Pero no nos basta con esto, hasta el momento el código fuente no cuenta con una estructura, lo que es necesario para poder seguir trabajando con él. Entonces necesitamos una representación que refleje dicha estructura y tal representación resulta ser un árbol de sintaxis abstracta (Abstract Syntax Tree AST). Se llama sintaxis abstracta porque en comparación con la sintaxis concreta en ésta sólo se almacena la información esencial relevante de cada uno de los enunciados del lenguaje. Por ejemplo, el programa: `5 +2;`, es un programa válido y su AST correspondiente es:



(a) AST notación gráfica

(**Plus** (**INum** 5) (**INum** 2))

(b) AST en notación prefija

Figura 2.5: Árbol de sintaxis abstracta de "`5 +2;`"

Obsérvese cómo un AST puede representarse gráficamente explícitamente como un árbol (figura 2.5a) y por otro lado también se puede representar en notación prefija (figura 2.5b), pero ambas representaciones denotan el mismo AST.

Por otra parte, nótese cómo no se almacenan los espacios en blanco y el `;` pues son completamente irrelevantes, en un programa de un lenguaje con soporte de condicional se podría tener por ejemplo, `if 5 < 10 then 2 else 4` en este caso el AST no guardaría las palabras `then` ni `else` pues son completamente irrelevantes, su árbol en notación prefija sería (**IF** (**Eq** (**INum** 5) (**INum** 10)) (**INum** 2) (**INum** 4)).

[3]Tanto en *flex* como en *bison* al atributo opcional de un token se le llama *valor semántico*.

La función del analizador sintáctico es por tanto, ir leyendo los tokens que genera el analizador léxico, verificar (con base en la gramática del lenguaje) que cumplen con la sintaxis y al mismo tiempo ir construyendo el AST (descartando los tokens superfluos).

Existen diferentes algoritmos de análisis sintáctico (*parsing*), unos construyen el árbol comenzando por la raíz (descendentes (*top-down*)) y otros comienzan por las hojas (ascendentes (*bottom-up*)), por otra parte, unos leen la cadena de izquierda a derecha y encuentran la derivación más a la izquierda (*LL Left to right Left most derivation*), mientras que otros leen la cadena de izquierda a derecha y encuentran la derivación más a la derecha (*LR Left to right Right most derivaction*).

En el análisis sintáctico de compiladores, usualmente se trabaja con gramáticas libres del contexto. Una técnica muy sencilla y fácil de implementar que sigue una estrategia descendente es el descenso recursivo, que es ampliamente utilizada en la práctica debido a que su implementación generalmente lleva poco tiempo comparado con otras técnicas. La desventaja del descenso recursivo es que puede llegar a tomar tiempo exponencial.[4] Por otro lado dos de los algoritmos más conocidos para realizar esta tarea son el de Earley [Ear68] y el CYK [Coc69; You67; Kas66], ambos algoritmos toman tiempo $O(n^3)$ en el caso general, sin embargo, en el contexto de compiladores se busca un menor tiempo de compilación y esto se logra con algoritmos deterministas,[5] de éstos los que siguen una estrategia ascendente son los más expresivos pero trabajan con un subconjunto estricto de las gramáticas libres del contexto. En efecto, fue tomando como base el estudio de los algoritmos que siguen una estrategia ascendente que se definieron los lenguajes libres de contexto deterministas que resultan ser los $LR(k)$ y a las correspondientes gramáticas deterministas que los generan, las $LR(k)$, donde k se refiere al número de símbolos de adelanto (*lookahead*) que se utilizan para determinar qué producción elegir (de manera determinista, sin tener luego que hacer retroceso *backtrack*).

Afortunadamente, utilizando uno de estos últimos algoritmos, es posible construir a partir de una gramática determinista el autómata de pila determinista que reconoce el lenguaje generado por ésta y desde luego, este proceso se puede automatizar. A los programas que realizan esta función se les conoce como *generadores de analizadores sintácticos* (*parser generators*). Uno de los primeros generadores de analizadores sintácticos y más influyentes fue *yacc* [Joh75] y uno de los actuales y de más amplio uso es *bison* [DS15].[6]

La sintaxis de nuestro lenguaje de ejemplo es la siguiente:

- Un factor F es un identificador, un flotante, un entero o una expresión entre paréntesis.

- Un término T es un factor o un término multiplicado por un factor.

[4] Aunque existen variantes donde se acota su tiempo de ejecución a un complejidad menor.

[5] El descenso recursivo, el algoritmo de Earley y el de CYK son no deterministas.

[6] Originalmente *bison* trabajaba con gramáticas LALR(1), que son un subconjunto estricto de las LR(1), que surgió debido a que las tablas LR(1) ocupaban demasiada memoria en la época en la que la memoria era escasa, hoy en día bison cuenta ya con soporte de gramáticas LR(1).

- Una expresión E es un término o una expresión sumada a un término.
- Una asignación A es un identificador (objetivo de la asociación que le sigue) seguido de una asociación y una expresión.
- Una declaración D es un tipo (int o float) seguido de un identificador.
- Una definición N es un tipo (int o float) seguido de un identificador (objetivo de la asociación que le sigue) donde a su vez a este identificador le sigue una asociación y una expresión.
- Un enunciado (*construct*) C es una expresión, una asignación, una declaración (todos seguidos de ";") o cualquiera de éstos seguidos de un enunciado (una secuencia de enunciados).

Esto es, la gramática que define la sintaxis concreta de nuestro lenguaje de ejemplo es:

P	$\longrightarrow$	C	Programa
C	$\longrightarrow$	E ***scn***	Enunciados
	$\mid$	D ***scn***	Enunciados simples
	$\mid$	A ***scn***	
	$\mid$	N ***scn***	
	$\mid$	E ***scn*** C	Enunciados compuestos (secuencia)
	$\mid$	D ***scn*** C	
	$\mid$	A ***scn*** C	
	$\mid$	N ***scn*** C	
N	$\longrightarrow$	***intd name asg*** E	Definición de un entero
	$\mid$	***floatd name asg*** E	Definición de un flotante
D	$\longrightarrow$	***intd name***	Declaración de un entero
	$\mid$	***floatd name***	Declaración de un flotante
A	$\longrightarrow$	***name asg*** E	Asignación
E	$\longrightarrow$	E ***plus*** T	Adición
	$\mid$	T	
T	$\longrightarrow$	T ***times*** F	Multiplicación
	$\mid$	F	
F	$\longrightarrow$	(E)	
	$\mid$	***inum***	Enteros
	$\mid$	***fnum***	Flotantes
	$\mid$	***name***	Variables

Figura 2.6: Gramática del lenguaje que soporta expresiones aritméticas, declaraciones, definiciones, asignaciones y secuencias.

Donde E permite generar una expresión aritmética, A una asignación, N una definición, D una declaración y C (un enunciado) cualquiera de las anteriores o (recursivamente) una secuencia de C.

bison permite expresar cada una de las producciones de la gramática junto con las acciones a realizar[7] cuando una secuencia de tokens de la entrada coincide con el lado derecho de la producción, para cada producción. En nuestro analizador sintáctico, en la acciones debemos indicar cómo ir construyendo el AST para cada una de las producciones.[8]

La parte relevante de la especificación en *bison* de nuestro lenguaje de ejemplo se presenta a continuación.

```
prog: cs { $$ = $1;}
;

cs: exp SCN    { $$ = $1;}
| decl SCN     { $$ = $1;}
| asg SCN      { $$ = $1;}
| def SCN      {$$ = $1;}
| exp SCN cs   {$$ = makeSeqN($1,$3);}
| decl SCN cs  {$$ = makeSeqN($1,$3);}
| asg SCN cs   {$$ = makeSeqN($1,$3);}
| def SCN cs   {$$ = makeSeqN($1,$3);}
;

def: INTD NAME ASGS exp {$$= makeDefN(Type::INT,$2,$4);}
| FLOATD NAME ASGS exp  {$$ = makeDefN(Type::FLOAT,$2,$4);}
;

decl: INTD NAME {$$ = makeIDeclN($2);}
| FLOATD NAME   {$$ = makeFDeclN($2);}
;

asg: NAME ASGS exp {$$ = makeAsgN($1,$3);}
;

exp: exp PLUS term { $$ = makePlusN($1,$3);}
| term             { $$ =$1;}
;

term: term TIMES fact { $$ = makeTimesN($1,$3);}
| fact                { $$ = $1;}
;

fact: LP exp RP { $$ =$2;}
| INUM          { $$ = makeINumN($1);}
| FNUM          { $$ = makeFNumN($1);}
| NAME          { $$ = makeVarN($1);}
;

%%
```

[7] A estas acciones se les conoce como acciones semánticas.

[8] En el caso de un intérprete en las acciones se debe indicar cómo evaluar la parte del programa representada por la producción y en el caso de los compiladores dirigidos por la sintaxis de una única pasada ir indicando cómo generar código al vuelo.

en donde, del lado izquierdo se escribe una producción y del lado derecho la acción semántica asociada a ella, en las acciones la notación `$$` se refiere al valor del símbolo a la izquierda de la producción y la notación `$n` se refiere al valor del n-ésimo símbolo del lado derecho de la producción, las funciones `make` van construyendo las nodos del AST a partir de otros nodos (construidos previamente) o de los tokens generados por el analizador léxico.

A continuación mostramos la salida de nuestro analizador sintáctico para el código de ejemplo 2.1.

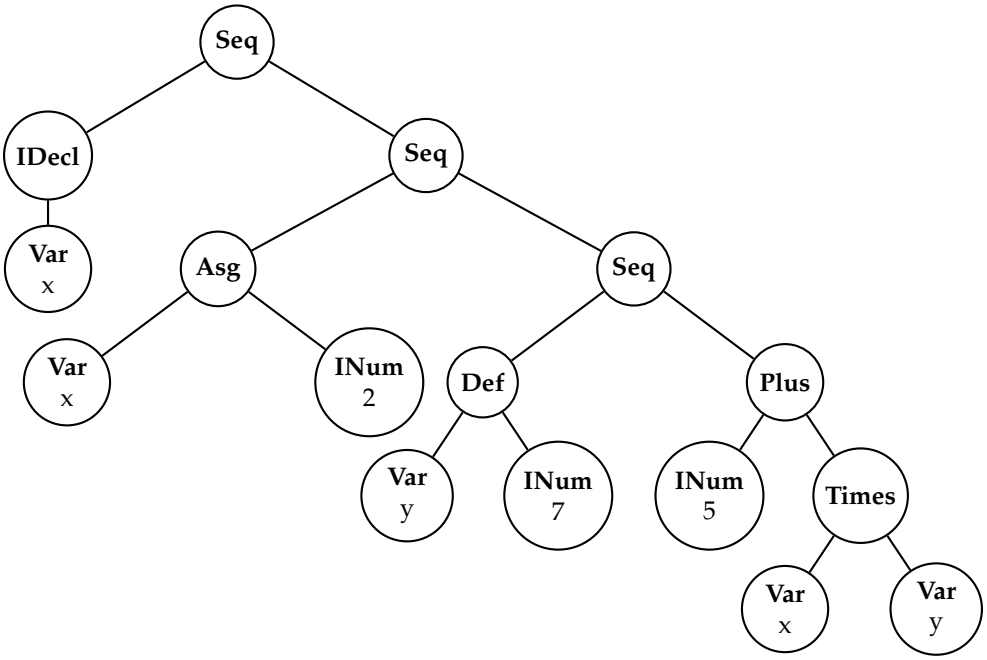


Figura 2.7: Árbol de sintaxis abstracta del ejemplo 2.1.

2.1.3. Análisis semántico

El analizador semántico se encarga de verificar que se cumplan algunas características dependientes del contexto como son (entre otras): que un identificador se haya declarado antes de usarse y que una función se llame con el número correcto de argumentos, pero su principal tarea es realizar verificación de tipos.

Por tanto la tarea del analizador semántico consiste en:

1. Llevar el seguimiento de las declaraciones de los símbolos para verificar que sus usos sean correctos.

2. Realizar verificación de tipos.

Para realizar la primera tarea se hace uso de una estructura de datos auxiliar conocida como la *tabla de símbolos* (*symbol table*).

Una tabla de símbolos es una tabla asociativa donde existe una entrada para cada símbolo[9] y en ella se almacena la información útil correspondiente a dicho símbolo.

Dicha tabla sirve como repositorio de información de los símbolos y generalmente es compartida entre las diferentes etapas a las que les es útil (para que agreguen o consulten información de los símbolos).

La información que es útil puede variar dependiendo del compilador, pero usualmente al menos se guarda el nombre del símbolo y su tipo. Adicionalmente, se guarda información útil para el generador de código, como el desplazamiento (*offset*) que sirve para calcular en qué posición se almacenará el valor de un símbolo en la pila de la máquina objetivo.[10]

En nuestro ejemplo debemos almacenar al menos el nombre, el tipo y el desplazamiento de los símbolos en la tabla de símbolos.

Entonces, el analizador semántico toma como entrada un AST y produce como salida un AST (decorado) con anotaciones de tipos.[11]

Así, el análisis semántico se puede realizar haciendo un recorrido en posorden, donde para cada nodo se va anotando su información de tipo y se va verificando la consistencia de los tipos, con ayuda de la tabla de símbolos.

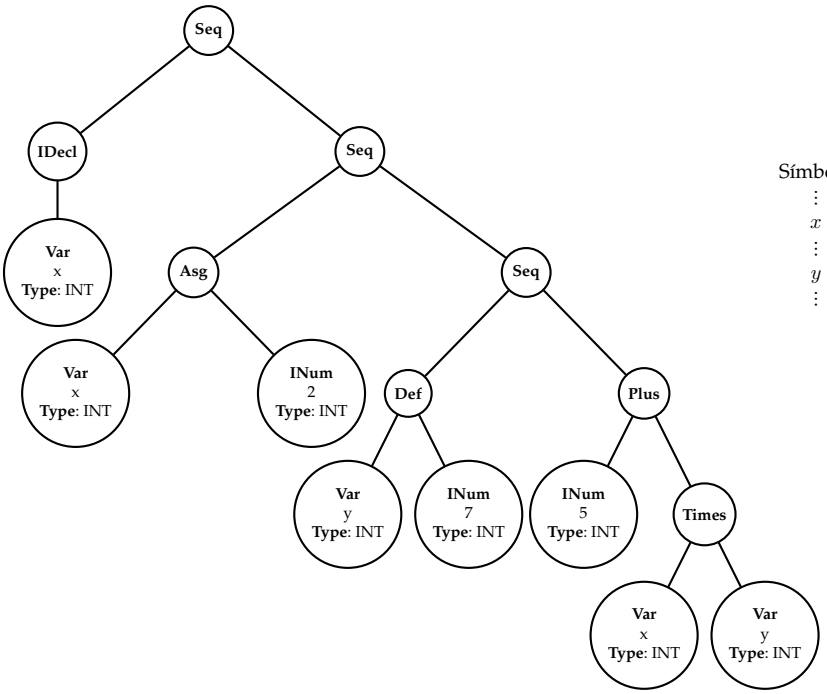
Es siguiendo la estrategia del párrafo anterior como se realizaría el análisis semántico para nuestro ejemplo. En la figura 2.8 se muestra cómo quedaría el AST con anotaciones y la tabla de símbolos al finalizar esta etapa.

Al terminar el análisis semántico se puede realizar la evaluación del programa de la siguiente manera: primero necesitamos un AST esta vez con anotaciones de valores (en lugar de anotaciones de tipos que se usan al realizar el análisis semántico, podemos nombrar a este árbol AST de evaluación) y una tabla de símbolos con un campo donde se almacenen los valores de éstos, luego se debe llevar a cabo un recorrido posorden donde, para cada (sub)-expresión anotemos el valor correspondiente a su evaluación (al realizar esta tarea la tabla de símbolos nos sirve como auxiliar para manejar los valores de los símbolos). Lo anterior se ilustra en la figura 2.9.

[9] En el caso de nuestro lenguaje de ejemplo los símbolos son únicamente las variables del lenguaje.

[10] Posteriormente otras etapas pueden enriquecer esta información, por ejemplo, un alojador de registros puede anotar el registro que se le asignó a una variable.

[11] Además desde luego agrega información a la tabla de símbolos.

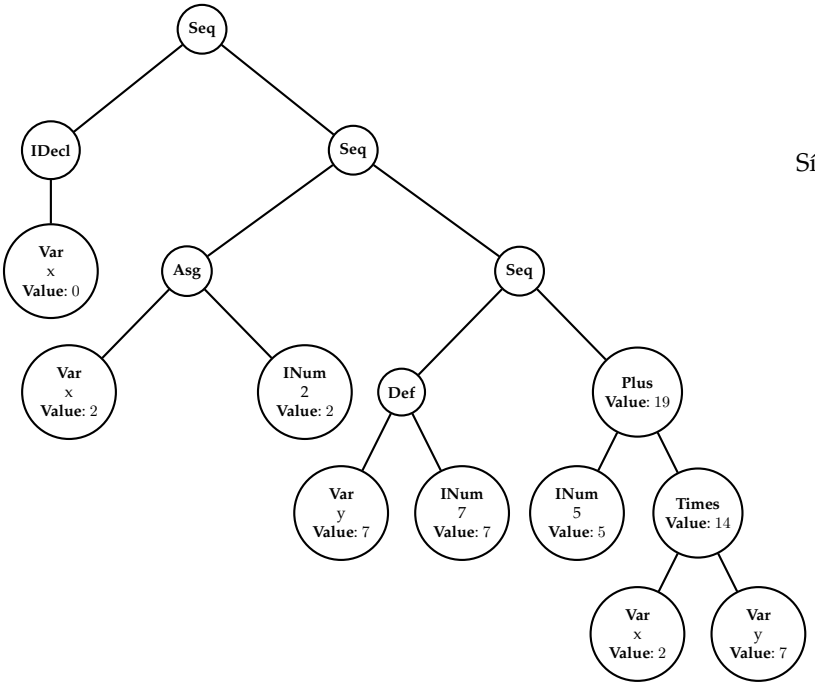


(a) AST con anotaciones de tipo

Símbolo		Nombre	Tipo	Desplazamiento
⋮				
x	←	x	INT	0
⋮				
y	←	y	INT	1
⋮				

(b) Tabla de símbolos

Figura 2.8: AST con anotaciones de tipo y tabla de símbolos del ejemplo 2.1 al finalizar el análisis semántico.



(a) AST de evaluación

Símbolo		Nombre	Tipo	Valor
⋮				
x	←	x	INT	2
⋮				
y	←	y	INT	7
⋮				

(b) Tabla de símbolos

Figura 2.9: AST de evaluación y tabla de símbolos del ejemplo 2.1 tras haber realizado la evaluación del programa.

De esta forma se obtiene un *intérprete* del lenguaje.[12]

Esto nos lleva a la siguiente pregunta: si ya hemos encontrado una forma de evaluar un programa ¿para qué compilarlo? y la respuesta es por **eficiencia**. Al ejecutar una versión compilada de un programa en el caso general se obtiene un menor tiempo de ejecución en comparación al que se obtiene al interpretarlo.

Debemos notar en este punto que no hemos definido de manera formal el análisis semántico y tampoco la evaluación de un programa, en su lugar lo que hemos hecho en ambos casos es dar una descripción informal de cómo se deben realizar estas dos etapas. Esto es desafortunado e indeseable porque una descripción informal puede dar lugar a vacíos y ambigüedades en una especificación de un lenguaje de programación. En el caso de este pequeño ejemplo no parece representar un problema crucial, sin embargo, en lenguajes de programación de amplio uso como C o C++ sí puede representar problemas más serios.

En particular es debido al uso de estas descripciones informales (dadas en lenguaje natural) que para un mismo programa escrito en un lenguaje de programación de amplio uso se pueden obtener resultados diferentes al evaluarlo, dependiendo de la implementación del lenguaje que se utilice. A continuación presentamos un ejemplo que ilustra este escenario para el caso particular del lenguaje C++.

Tomemos como punto de partida el siguiente programa escrito en C++:

```
#include <iostream>

void imprime(std::string s, std::string p);

int main(void){
  std::string s = "hola";
  imprime(s, s="mundo");
}

void imprime(std::string s, std::string p){
  std::cout << s << " " << p << std::endl;
}
```

en él se define una función `imprime` la cual se encarga de imprimir las dos cadenas que se le pasan como argumento separadas por un espacio en blanco. Por otro lado se tiene una cadena `s` a la cual se le asigna inicialmente el valor `hola`, entonces cuando se hace la llamada `imprime(s, s="mundo")` a priori lo que se espera como obtener como resultado de la evaluación del programa es que se imprima la cadena `hola mundo`. Sin embargo

[12]Es decir, si se realiza una etapa de evaluación después de haber realizado el análisis semántico entonces lo que se obtiene es un intérprete del lenguaje, en tal caso la etapa de evaluación es la última etapa del intérprete. En cambio si no se realiza esta etapa de evaluación y se sigue el proceso usual de compilación en el que típicamente se lleva a cabo la generación de código después de haber realizado el análisis semántico entonces se obtiene un compilador del lenguaje.

cuando compilamos y ejecutamos este programa con el compilador de C++ `g++` de *GCC* (*GNU Compiler Collection*) se obtiene lo siguiente:[13]

```
user@host ~ $ g++ ejEvD.cpp -o ejgcc
user@host ~ $ ./ejgcc
mundo mundo
```

es decir, se obtiene la cadena `mundo mundo`. En cambio si lo compilamos con `clang++` del compilador *LLVM* se obtiene lo siguiente:[14]

```
user@host ~ $ clang++ ejEvD.cpp -o ejllvm
user@host ~ $ ./ejllvm
hola mundo
```

es decir, en este caso sí se obtiene lo que se esperaba o sea la cadena `hola mundo` esto es debido a que en C++ no se sabe cuál será el orden de evaluación de los argumentos al realizar una llamada a una función, pues esto no se especifica formalmente de manera explícita en su especificación. Con esto tenemos un ejemplo concreto de cómo para un mismo programa escrito en un lenguaje de amplio uso se pueden obtener resultados diferentes al evaluarlo.

Lamentablemente la mayoría de las especificaciones de lenguajes en amplio uso en la vida real (como C o C++) cuentan con descripciones informales para las etapas de análisis semántico y evaluación. Nosotros estudiaremos cómo remediar esta situación, utilizando mecanismos para definir formalmente la semántica de un lenguaje en la sección 2.3 y así poder evitar estos problemas.

2.1.4. Generación de código

Para generar código (una vez que se ha realizado el análisis semántico) lo primero que se debe definir es la máquina objetivo, ésta puede ser una máquina virtual o una máquina real (discutimos ya en la introducción en el capítulo 1, brevemente algunas de las ventajas y desventajas de éstas).

Cabe mencionar que si se elige una máquina virtual entonces luego de haber generado código para ésta se puede traducir este código al código una (o varias) máquina(s) real(es). Esto es, una vez que se ha generado código para la máquina virtual, cada instrucción de la máquina virtual se expande en una secuencia de instrucciones de la máquina real, a esta tarea se le conoce como *expansión de código* [Ler90].

[13] Se utilizó la versión `4.9.0` de *GCC* sobre el sistema operativo Linux en un arquitectura `x64`.
[14] Se utilizó la versión `3.4.2` de *LLVM* sobre el sistema operativo Linux en un arquitectura `x64`.

Una vez elegida la máquina objetivo, para generar código se puede realizar un recorrido en posorden (con ayuda de la tabla de símbolos) sobre el AST decorado que generó el analizador semántico, donde, para cada nodo se va emitiendo el conjunto de instrucciones de máquina que implementan la operación representada por el nodo en la máquina objetivo.[15]

Para realizar expansión de código (si es que se generó previamente código para una máquina virtual), se va haciendo un recorrido lineal sobre el código de la máquina virtual, donde, para cada instrucción de la máquina virtual se emite la secuencia de instrucciones de la máquina real que realizan dicha operación (como por lo general las instrucciones de una máquina virtual tienen un mayor nivel de abstracción en comparación con las de una máquina real es necesario implementarlas como una o varias instrucciones de una máquina real).

Por otra parte como mencionamos nuestro lenguaje a compilar será un lenguaje funcional basado en el cálculo lambda, pero ¿cómo es que a partir del cálculo lambda podemos llegar a un lenguaje de programación? Ahora, también dijimos que el sistema formal de nuestro asistente de pruebas Coq es el cálculo de construcciones inductivas que es un cálculo lambda con tipos. Por tanto para poder comprender el funcionamiento de nuestro lenguaje fuente debemos estudiar primero qué es el cálculo lambda y algunas propiedades importantes con las que cuenta. Además por otro lado para comprender por qué Coq tiene ciertas características específicas como que una función recursiva se tiene que definir utilizando recursión estructural basada en la sintaxis, debemos estudiar primero el cálculo lambda con tipos (del cual el sistema formal de Coq es una extensión). Por estas razones estudiaremos el cálculo lambda (con y sin tipos) en la siguiente sección.

2.2. Cálculo lambda

El cálculo lambda es un sistema formal cuyo objetivo original era ser usado como fundamento de las matemáticas.

Nosotros lo estudiaremos por las siguientes razones:

1. Porque es nuestro deseo que nuestro lenguaje fuente sea un lenguaje funcional basado en el cálculo lambda, por tanto para poder llegar a comprender el funcionamiento y las propiedades que éste tendrá es necesario partir del cálculo lambda. Posteriormente en el capítulo 4 analizaremos lo necesario para poder utilizar el cálculo lambda como lenguaje de programación.[16]

[15] Esta es una generación de código ingenua, por lo general una operación representada por un nodo del AST se puede implementar por diferentes secuencias de instrucciones de código de la máquina objetivo por lo que en los compiladores en producción se suelen utilizar algoritmos que elijan la secuencia de instrucciones óptima con base en un criterio específico, por ejemplo: eficiencia en tiempo de ejecución o menor consumo de energía del equipo.

[16] Ya Church en [Chu32] vislumbraba el hecho que se le podían dar otros usos tal como se refleja cuando dice: "*There may, indeed, be other applications of the system than its use as logic*".

2. Por otra parte como deseamos poder realizar la demostración de la corrección de nuestro compilador, pensamos que nos proporcionará claridad saber qué es un sistema de demostración y ver que el cálculo lambda puede ser utilizado como tal. En efecto el sistema de demostración que utiliza Coq es un cálculo lambda tipificado con diversas extensiones al que se le conoce como cálculo de construcciones inductivas.

El cálculo lambda original es un cálculo cuya definición como veremos más adelante es muy simple y al que se le conoce como cálculo lambda puro. A lo largo del tiempo se le fueron haciendo diversas extensiones con diferentes fines una de las más destacadas fue el agregarle tipos (reminiscencia de la idea original de Russell de utilizar tipos para evitar paradojas en el fundamento de las matemáticas), por eso cuando se habla del cálculo lambda en general no se hace alusión a un cálculo en específico sino a una familia de cálculos, podemos decir que esta familia se divide en dos grandes grupos el cálculo lambda con diferentes extensiones sin tipos y el cálculo lambda con diferentes extensiones con tipos o tipificado.

Primero estudiaremos el cálculo lambda puro (sin tipos) porque será el que tomaremos como base para llegar a nuestro lenguaje fuente y después presentaremos el cálculo lambda simplemente tipificado y veremos cómo se puede formular como sistema de demostración en ese mismo sentido se pueden utilizar diferentes extensiones de este cálculo como sistema de demostración en particular el cálculo de construcciones inductivas. En efecto el cálculo de construcciones inductivas es una extensión del cálculo lambda simplemente tipificado.

El cálculo lambda simplemente tipificado también se puede utilizar como lenguaje de programación, un teorema que se cumple en el cálculo lambda simplemente tipificado y que no se cumple en el cálculo lambda puro es el teorema de normalización fuerte el cual intuitivamente nos indica que todo programa que se exprese en este cálculo termina. El cálculo de construcciones inductivas que como mencionamos es una extensión del cálculo lambda simplemente tipificado también se puede usar como lenguaje de programación y cumple con el teorema de normalización fuerte, por eso todo programa que se escribe en Coq necesariamente debe terminar. Por lo anterior también enunciaremos el teorema de normalización fuerte al estudiar el cálculo lambda simplemente tipificado.

Al cálculo de construcciones inductivas se la dan pues ambos usos, esto es, se usa como sistema de demostración y como lenguaje de programación.

Comenzaremos ahora nuestro estudio del cálculo lambda, cabe mencionar que nuestro tratamiento está basado en [HS08].

2.2.1. Cálculo lambda sin tipos

Primero presentamos la sintaxis.

Definición 2.2.1 (λ−términos)**.** Sea $V = \{v_0, \ldots, v_i, \ldots\}$ un conjunto (infinito) de variables y

sea $C = \{c_0, \ldots, c_i, \ldots\}$ un conjunto (infinito) de constantes atómicas[17] entonces el conjunto de λ–términos se define inductivamente como:

1. Para toda $x,\ x \in V$ y para toda $c,\ c \in C$, x y c son λ–términos a los que se les llama *átomos*.

2. Si M y N son λ–términos, entonces (MN) es un λ–término al que se llama *aplicación*.

3. Si M es un término y $x \in V$, entonces $(\lambda x.M)$ es un λ–término al que se le denomina *abstracción*.

Si en 1. únicamente consideramos variables, entonces nuestro sistema será el cálculo lambda puro. Si consideramos variables y constantes entonces a este sistema se llama cálculo lambda aplicado.

T	$\longrightarrow$	c	Constantes
	$\mid$	x	Variables
	$\mid$	$(\lambda x.T)$	Abstracción
	$\mid$	$T\ T$	Aplicación

Figura 2.10: Alternativamente se puede considerar que esta gramática define la sintaxis del cálculo lambda.

Cuando no haya lugar a ambigüedad diremos simplemente la palabra término para referirnos a un λ–término.

Ejemplo 2.2.1 (λ–términos)**.** Los siguientes son λ–términos:

1. x

2. (xy)

3. $(\lambda x.x)$

4. $(\lambda x.(yz))$

5. $((\lambda y.y)(\lambda x.(xy)))$

6. $(x(\lambda x.(\lambda x.x)))$

[17] Distinto del conjunto de variables, es decir, $V \cap C = \varnothing$.

El símbolo "≡" (sin comillas) denota equivalencia sintáctica.
Además por convención se tienen las siguientes reglas de precedencia y asociatividad:

- $MN_1 \dots N_n \equiv (\dots(MN_1)\dots N_n)$ (asociatividad izquierda de la aplicación)
- $\lambda x.MN \equiv (\lambda x.(MN))$ (precedencia de la aplicación sobre la abstracción)
- $\lambda x_1 \dots x_n.M \equiv (\lambda x_1.(\dots(\lambda x_n.M)\dots))$ (asociatividad a la derecha de la abstracción)

Ejemplo 2.2.2 (Precedencia y asociatividad)**.** Los siguientes términos son equivalentes (sintácticamente):

1. $wxyz \equiv (((wx)y)z)$
2. $\lambda yx.xyz \equiv (\lambda y.(\lambda x.((xy)z)))$

La operación fundamental que representa la esencia de un cálculo en el cálculo lambda es la de contracción β que presentaremos más adelante, pero antes en vías de poder dar la definición de esta operación es necesario dar la definición varios conceptos básicos que iremos presentado a continuación.

Damos primero el concepto de ocurrencia o subtérmino que nos será de ayuda para definir otros conceptos más adelante.

Definición 2.2.2 (Ocurre)**.** Sean P y Q λ–términos. Entonces definimos la relación P *ocurre* en Q (o P es un *subtérmino* de Q o Q *contiene* a P) inductivamente de la siguiente manera:

1. P ocurre en P
2. si P ocurre en M o en N, entonces P ocurre en (MN)
3. si P ocurre en M o $P \equiv x$, entonces P ocurre en $(\lambda x.M)$

Ejemplo 2.2.3 (Ocurre)**.** Los siguientes son algunos ejemplos de las relación ocurre:

1. $\lambda x.x$ ocurren en $\lambda x.x$
2. x ocurre en $((xy)z)$
3. x ocurre en $\lambda w.((xy)z)$
4. w ocurre en $\lambda w.((xy)z)$

Intuitivamente es claro que un término Q puede tener diferentes ocurrencias en un término P, por ejemplo, x tiene dos ocurrencias en el término $\lambda x.xy$ mientras y sólo una; por tanto se da por concedido la noción de "una ocurrencia de P en Q".[18]

A continuación damos el concepto de alcance que nos servirá entre otras cosas para poder definir cuándo una ocurrencia de una variable está libre o ligada.

[18]Podemos dar una definición de ocurrencia que formalice nuestra idea intuitiva, no obstante eso significaría entrar en detalles técnicos que no son relevantes para nuestra discusión. Para el lector riguroso hemos dado una definición formal en el apéndice .

Definición 2.2.3 (Alcance)**.** Sea P un λ–término y supongamos que existe una ocurrencia de $\lambda x.M$ en P. Entonces se dice que la ocurrencia de M es el alcance de la ocurrencia de λx (que está a su izquierda).

Ejemplo 2.2.4 (Alcance)**.** Lo siguiente ejemplifica el concepto de alcance:

1. En $\lambda x.(xy)$, (xy) es el alcance de λx.
2. En $\lambda y.(\lambda x.(xy))$, $\lambda x.(xy)$ es el alcance de λy.

Ahora daremos los conceptos de variables libres y ligadas que nos servirán entre otras cosas para poder definir de manera correcta la operación de sustitución más adelante.

Definición 2.2.4 (Variables libres y ligadas)**.** Una ocurrencia de una variable x en un término P se llama:

- *ligada* (*bound*) si está en el alcance de una λx en P
- *ligada* (*bound*) y *ligadora* (*binding*) si y sólo si es la x de una λx
- *libre* (*free*) en otro caso

Si x tiene al menos una ocurrencia ligadora (*binding ocurrence*) en P, se dice que x es una variable ligada de P. Si x tiene al menos una ocurrencia libre en P, se dice que x es una variable libre de P.

Ejemplo 2.2.5 (Variables libres y ligadas)**.** En lo siguiente se ejemplifican los conceptos de variables libres y ligadas:

1. En $\lambda x.(xy)$ la ocurrencia de x externa izquierda es una ocurrencia ligadora (y ligada), mientras que la x en (xy) es una ocurrencia ligada.
2. En $(y(\lambda y.xy))$ la única ocurrencia de x es libre, mientras que la y de la extrema izquierda es libre, la y de λy es ligadora (y ligada) y finalmente la y de (xy) es ligada. Nótese cómo una variable puede ser libre y ligada a la vez en un término, ya que ocurrencias diferentes de la misma variable pueden ser unas ligadoras y otras libres.

Damos en seguida la definición de conjunto de variables libres que nos servirá en la definición de la operación de sustitución y también para poder definir qué es un término cerrado.

Definición 2.2.5. El conjunto de variables libres de un λ–término M denotado como $FV(M)$ se define inductivamente como:

1. $FV(x) = \{x\}$
2. $FV(\lambda x.M) = FV(M) - \{x\}$

3. $FV(MN) = FV(M) \cup FV(N)$

Ejemplo 2.2.6 (Variables libres de un término)**.** Los siguientes son ejemplos del conjunto de variables libres de un término:

1. Si $M \equiv \lambda z.(x(\lambda y.yz))$, el conjunto de variables libres de M es $FV(M) = \{x\}$.
2. Si $M \equiv (\lambda z.(x(\lambda y.yz)))z$, el conjunto de variables libres de M es $FV(M) = \{xz\}$.
3. Si $M \equiv x(\lambda y.((\lambda z.z)y))$, el conjunto de variables libres de M es $FV(M) = \{x\}$.

El concepto de término cerrado nos servirá más adelante entre otras cosas para ver que en ciertos escenarios sólo tiene sentido considerar este tipo de términos.

Definición 2.2.6. Se dice que un término es *cerrado* si no tiene variables libres.

Ejemplo 2.2.7 (Término cerrado)**.** A continuación se presenta ejemplos de términos cerrados:

1. Si $M \equiv (\lambda x.x)(\lambda y.y)$, el conjunto de variables libres de M es $FV(M) = \varnothing$ por lo que M es cerrado.
2. Si $M \equiv \lambda x.(\lambda y.(xy))$, el conjunto de variables libres de M es $FV(M) = \varnothing$ por lo que M es cerrado.
3. Si $M \equiv \lambda y.((\lambda x.x)y)$, el conjunto de variables libres de M es $FV(M) = \varnothing$ por lo que M es cerrado.

Una operación que es de gran importancia es la de sustitución que a continuación presentamos. La operación de sustitución es fundamental para poder definir la de contracción β, esta última representa la operación fundamental de cálculo en el cálculo lambda. Cabe mencionar que si no se es cuidadoso al definir o realizar esta operación es fácil que se puedan cometer errores. En particular un problema muy común relacionado con esta operación es el de captura de variables que estudiaremos en la sección 4.2.

Definición 2.2.7 (Sustitución)**.** Sean M y N λ–términos y x una variable. Entonces se define la sustitución de x por N en M denotada como $[N/x]M$ (y se dice que toda ocurrencia libre de x en M se sustituye por N) de la siguiente manera:

1. $[N/x]x \equiv N$
2. $[N/x]a \equiv a$ para todo átomo $a \not\equiv x$
3. $[N/x](PQ) \equiv ([N/x]P[N/x]Q)$
4. $[N/x](\lambda x.P) \equiv \lambda x.P$
5. $[N/x](\lambda y.P) \equiv \lambda y.P$ si $x \notin FV(P)$

6. $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$ si $x \in FV(P)$ y $y \notin FV(N)$

7. $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$ si $x \in FV(P)$ y $y \in FV(N)$.

(En los incisos 5., 6. y 7. se tiene que $y \neq x$. Por otra parte en el inciso 7., z es una variable tal que $z \notin FV(NP)$ y generalmente se utiliza en el lugar de z la siguiente variable disponible de acuerdo al orden lexicográfico).

Veamos brevemente cómo se pueden cometer errores si no realizamos con cuidado la operación de sustitución como la acabamos de definir. Fijémonos en un ejemplo muy simple, consideremos el término $\lambda u.v$ en él la única ocurrencia de la variable u está ligada, ahora supongamos que deseamos realizar la sustitución $[u/v](\lambda u.v)$, entonces si no somos cuidadosos podemos aplicar el caso 6. de nuestra definición y obtendremos como resultado $\lambda u.u$ lo cual es un error, en esta situación la ocurrencia de una variable que era libre ahora está ligada. Lo correcto es aplicar el caso 7. con lo cual primero debemos renombrar, es decir, $\lambda u.v \equiv \lambda w.v$ y después realizar la sustitución propiamente dicha con lo cual se obtiene $\lambda w.u$. Realizar el paso de renombrado según se establece en el caso 7. es equivalente a pedir que se realice una conversión α la cual definiremos más adelante. Estudiaremos con más detalle el problema de captura de variables en la sección 4.2.

Ejemplo 2.2.8 (Sustitución)**.** Los siguientes son ejemplos de sustituciones:

1. $[x/y](xy) \equiv (xx)$

2. $[y/x](\lambda x.y) \equiv \lambda x.y$

3. $[x/y](\lambda y.\lambda x.xy) \equiv \lambda y.\lambda z.zx$

4. $[z/y](\lambda x.y) \equiv \lambda x.z$

5. $[u/v](\lambda u.v) \equiv \lambda w.u$

La conversión α de manera sencilla simplemente corresponde a cambiarle el nombre a una variable, ponerle un nombre diferente a una variable a priori es irrelevante. La importancia de esta operación en el cálculo lambda es que si no se realiza con cuidado una ocurrencia de una variable libre puede quedar ligada y viceversa. También por otro lado que ésta se puede utilizar como operación auxiliar al realizar sustitución que como mencionamos es una operación fundamental en el cálculo lambda.

Definición 2.2.8 (Conversión α)**.** Sea P un término que contiene una ocurrencia de $\lambda x.M$ y sea $y \in FV(M)$. Al acto de reemplazar esta $\lambda x.M$ por $\lambda y.[y/x]M$ se le denomina conversión α en P. Si (y sólo si) P se puede convertir a Q mediante una sucesión finita (posiblemente vacía) de conversiones α, entonces decimos que P se convierte a Q mediante α y se denota como $P \equiv_\alpha Q$.

Ejemplo 2.2.9 (Conversión α)**.** Lo siguiente ejemplifica la conversión α.

Si $P \equiv \lambda x.(\lambda y.x(xy))$ y $Q = \lambda u.(\lambda v.u(uv))$ entonces $P \equiv_\alpha Q$ ya que $\lambda x.(\lambda y.x(xy)) \equiv_\alpha \lambda x.(\lambda v.x(xv)) \equiv_\alpha \lambda u.(\lambda v.u(uv))$.

La contracción β por su parte representa la esencia de realizar un cálculo en el cálculo lambda y es por ello que generalmente se considera la de mayor importancia dentro de éste. Por otra parte la contracción β utiliza como operación auxiliar la de sustitución de ahí la relevancia de esta última.

Definición 2.2.9 (Contracción β)**.** Sea Q un λ–término si Q es de la forma $(\lambda x.M)N$ entonces a Q se le denomina *redex* β. Por otra parte si consideramos el término $[N/x]M$ a éste se llama *reducto* de Q. Sea P un λ–término cualquiera si (y sólo si) P contiene cualquier ocurrencia O de Q y O se reemplaza por $[N/x]M$ dando como resultado un término P' entonces decimos que se contrajo la ocurrencia O del redex Q de P . También se dice que P se contrae a P' mediante β y se denota como $P \to_\beta P'$.

Cuando no haya lugar a ambigüedad escribiremos simplemente $P \to Q$ para indicar $P \to_\beta Q$.

Ejemplo 2.2.10 (Contracción β)**.** Los siguientes son ejemplos donde realizan contracciones β:

1. $(\lambda x.x(xy))N \to N(Ny)$.
2. $(\lambda x.y)N \to y$.
3. $(\lambda x.(\lambda y.yx)z)v \to [v/x]((\lambda y.yx)z) \equiv (\lambda y.yv)z \to [z/y](yv) \equiv zv$.
4. $(\lambda x.xxy)(\lambda x.xxy) \to (\lambda x.xxy)(\lambda x.xxy)y \to (\lambda x.xxy)(\lambda x.xxy)yy \ldots$.

Ahora que ya vimos que intuitivamente una contracción β representa realizar un cálculo quisiéramos poder realizar una secuencia de éstos. Vale notar que dicha secuencia podría ser finita o infinita. Para este fin definimos la reducción β.

Definición 2.2.10 (Reducción β)**.** Una reducción β digamos ρ es una secuencia (finita o infinita) de contracciones de la forma:

$$X_1 \to_\beta Y_1 \equiv_\alpha X_2 \to_\beta Y_2 \equiv_\alpha X_3 \to_\beta \ldots$$

a X_1 se le denomina el comienzo de ρ.
La longitud $|\rho|$ de ρ es:

1. ∞ si ρ es una secuencia infinita
2. n en otro caso, pues entonces se trata de una secuencia finita y ρ es de la forma:

$$X_1 \to_\beta Y_1 =_\alpha X_2 \ldots X_i \to_\beta Y_i =_\alpha X_{i+1} \ldots Y_n =_\alpha X_{n+1}$$

es decir, n es el número de contracciones sin tomar en cuenta los pasos necesarios de las conversiones α. Por otro lado a X_{n+1} se le llama fin.

Sea ρ una reducción β con comienzo P y fin Q entonces ρ se denota como $P \twoheadrightarrow_\beta Q$.
Si $P \twoheadrightarrow Q$ entonces decimos que P se reduce a Q mediante β.

Cuando no haya lugar a ambigüedad escribiremos simplemente $P \twoheadrightarrow Q$ por decir $P \twoheadrightarrow_\beta Q$.

Alternativamente podemos presentar la reducción β "$\twoheadrightarrow_\beta$" como una relación definida inductivamente de la siguiente manera.

Definición 2.2.11 (Reducción β)**.** Definimos la relación reducción β denotada como $\twoheadrightarrow$ de la siguiente manera:

(α) $\lambda x.M \twoheadrightarrow \lambda y.[y/x]M$ si $y \notin FV(M)$

(β) $(\lambda x.M)N \twoheadrightarrow [N/x]M$

(ρ) $M \twoheadrightarrow M$ (reflexividad)

(μ) $\dfrac{M \twoheadrightarrow M'}{NM \twoheadrightarrow NM'}$ (monotonía derecha)

(ν) $\dfrac{M \twoheadrightarrow M'}{MN \twoheadrightarrow M'N}$ (monotonía izquierda)

(ξ) $\dfrac{M \twoheadrightarrow M'}{\lambda x.M \twoheadrightarrow \lambda x.M'}$

(τ) $\dfrac{M \twoheadrightarrow N \qquad N \twoheadrightarrow P}{M \twoheadrightarrow P}$ (transitividad)

Ejemplo 2.2.11 (Reducción β)**.** Los siguientes son ejemplos de reducciones β.

1. $(\lambda x.x(xy))N \twoheadrightarrow N(Ny)$ ya que $(\lambda x.x(xy))N \to N(Ny)$.
2. $(\lambda y.((\lambda x.(xy))z))z \twoheadrightarrow zz$ ya que $(\lambda y.((\lambda x.(xy))z))z \to (\lambda x.(xz))z \to zz$.
3. $(\lambda x.(xx))(\lambda x.(xx)) \twoheadrightarrow (\lambda x.(xx))(\lambda x.(xx)) \twoheadrightarrow \ldots$ ya que:
   $(\lambda x.(xx))(\lambda x.(xx)) \to (\lambda x.(xx))(\lambda x.(xx)) \to (\lambda x.(xx))(\lambda x.(xx)) \to \ldots$

Por otra parte dado un término cualquiera quisiéramos saber cuando no hay nada más por calcular. Con este fin introducimos el concepto de forma normal β.

Definición 2.2.12 (Forma normal β)**.** Dado un término Q cualquiera si Q no contiene *redexes* β entonces Q se llama forma normal β (o se dice que Q está en forma normal β).

Sea P un término cualquiera y sea Q un término en forma normal β, si P se reduce a Q mediante β (o sea $P \twoheadrightarrow Q$) entonces se dice que Q es una forma normal β de P.

Cuando no haya lugar a ambigüedad diremos simplemente forma normal por decir forma normal β.

Ejemplo 2.2.12 (Forma normal β)**.** Los siguientes son ejemplos de términos en forma normal β:

1. xy es una forma normal β porque no tiene redexes β.
2. zz es una forma normal β de $(\lambda y.((\lambda x.(xy))z))z$ ya que $(\lambda y.((\lambda x.(xy))z))z \twoheadrightarrow zz$
3. Veamos cómo el término $P \equiv ((\lambda u.v)((\lambda x.xx)(\lambda x.xx)))$ se puede reducir al menos de las dos siguientes maneras:

   a) $((\lambda u.v)((\lambda x.xx)(\lambda x.xx))) \to v$ por lo que v es una forma normal β de P.

   b) $((\lambda u.v)((\lambda x.xx)(\lambda x.xx))) \to ((\lambda u.v)((\lambda x.xx)(\lambda x.xx))) \to$
   $((\lambda u.v)((\lambda x.xx)(\lambda x.xx)))\ldots$

   Con lo que podemos observar que si un término P tiene una forma normal β no necesariamente todas sus reducciones son finitas. Intuitivamente nos dice que si un término se evalúa a un valor final eligiendo de cierto modo el siguiente paso a calcular (es decir, el siguiente redex β) no necesariamente terminará su evaluación si se elige el siguiente paso a calcular de distinta manera.
4. Podemos notar que el término $(\lambda x.xx)(\lambda x.xx)$ no tiene forma normal β.

Podemos preguntarnos entonces de manera sencilla si en el caso de que se llegue a un valor final siguiendo un posible camino de evaluación se puede llegar a un valor final distinto siguiendo otro, lo cual sería sin duda indeseable si se desea utilizar el cálculo lambda como lenguaje de programación. Es por ello que nosotros presentamos aquí uno de los teoremas más importantes del cálculo lambda conocido como Church-Rosser el cual nos dice intuitivamente que si se siguen dos caminos diferentes de evaluación entonces no importando cuánto se alejen el uno del otro en algún punto se encontrarán. Y luego como corolario de este teorema se tiene que si se llega a un valor final por uno de estos caminos se llegará al mismo valor final por el otro que es justo lo que se quería en el contexto de utilizar el cálculo lambda como lenguaje de programación.

Teorema 2.2.1 (Church-Rosser)**.** Si $P \twoheadrightarrow M$ y $P \twoheadrightarrow N$, entonces existe T tal que $M \twoheadrightarrow T$ y $N \twoheadrightarrow T$

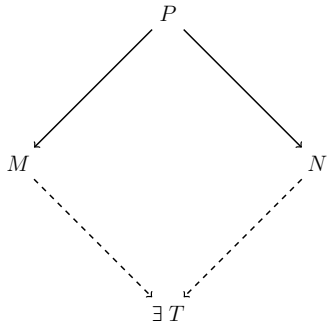


Figura 2.11

Para ver una demostración de este teorema favor de consultar [CFC58; Bar12].

La propiedad del teorema Church-Rosser que establece que si un término se puede reducir a dos términos diferentes entonces ambos tales términos se pueden reducir a un mismo término se denomina *confluencia*.

Como podemos notar este teorema establece que la reducción β es confluente.

Ahora presentamos el corolario que nos dice intuitivamente que si un término tiene valor final entonces éste es único.

Corolario 2.2.1 (Unicidad de la forma normal)**.** Sea P un término cualquiera, si P tiene una forma normal β entonces dicha forma normal es única modulo conversión α. Esto es si P tiene la formas normales M y N entonces se tiene que $M \equiv_\alpha N$.

Hasta este punto hemos presentado las bases del cálculo lambda y uno de sus teoremas más importantes en vías de poder utilizarlo como lenguaje de programación. En la sección 4.1 continuaremos el estudio de éste como lenguaje de programación con el objetivo de que sirva como base de nuestro lenguaje fuente.

Es nuestra intención en este espacio dar una mirada al cálculo lambda en la forma en que Coq lo utiliza.

Una propiedad muy importante que posee el cálculo lambda simplemente tipificado (y que no posee el cálculo lambda puro) es la de terminación, la cual de forma sencilla nos dice que si este cálculo se utiliza como lenguaje de programación entonces toda evaluación posible de cualquier término finaliza. Esta es sin duda una propiedad muy deseable aunque también de cierto modo restrictiva. El cálculo lambda en el que Coq está basado es una extensión del cálculo lambda simplemente tipificado el cual también cuenta con la propiedad de terminación. Es por lo anterior que presentaremos brevemente el cálculo lambda simplemente tipificado y enunciaremos el teorema de normalización fuerte que establece la propiedad de terminación.

Por otra parte daremos una mirada a qué es un sistema de demostración e ilustraremos este concepto presentando el sistema de deducción natural para la lógica proposicional intuicionista. Luego vislumbraremos cómo es que el cálculo lambda simplemente tipificado se puede formular como sistema de demostración y veremos que existe una correspondencia entre (el fragmento implicacional de) la lógica proposicional intuicionista y el cálculo lambda simplemente tipificado vía el isomorfismo de Curry-Howard. Vía este isomorfismo la proposiciones (de la lógica) corresponden a los tipos (del cálculo lambda) y las demostraciones corresponden a los términos. Esta presentación es con la única finalidad de ganar intuición de la teoría en la que descansa Coq que se ve reflejada en la práctica en forma de diferentes características con las que cuenta este asistente de pruebas.

2.2.2. Cálculo lambda simplemente tipificado

Comenzaremos presentado el cálculo lambda simplemente tipificado. Nuestra presentación será breve y paralela a la del calculo lambda puro.[19]

[19]Para ver más detalles favor de consultar [HS08].

Por su puesto el nuevo ingrediente principal son los tipos por lo que a continuación los presentamos.

Definición 2.2.13 (Tipos simples)**.** Sea T un conjunto (finito o infinito) de símbolos a los que llamaremos tipos atómicos entonces definimos un tipo como sigue:

1. Si $t \in T$ entonces t es un tipo.
2. Si σ y τ son tipos, entonces la expresión $(\sigma \to \tau)$ es un tipo, al que se le denomina tipo función.

Ahora veamos cómo se le añaden éstos a la variables.

Definición 2.2.14 (Variables y constantes tipificadas)**.** Sea $V = \{v_0, \ldots, \}$ un conjunto (infinito) de variables a las que llamaremos variables sin tipo. Sea x una variable sin tipo y τ un tipo, entonces podemos formar variables con tipo x^τ de la siguiente manera:

1. Condición de consistencia: A ninguna variable (sin tipo) se le asigna más de un tipo, es decir, no está permitido tener x^τ y x^σ con $\tau \nexists \sigma$
2. Todo tipo τ es asignado a un conjunto infinito de variables.

Si se tiene x^τ se dice que x tiene tipo τ. El caso de las constantes es análogo pero trabajando con un conjunto C finito (posiblemente vacío) o infinito de constantes.

Ahora presentamos la sintaxis, siguiendo la misma estructura que para el cálculo lambda puro.

Definición 2.2.15 (λ–términos simplemente tipificados)**.** El conjunto de λ–términos tipificados se define como:

1. Para toda variable tipificada x^τ y para toda constante tipificada c^τ, x^τ y c^τ son λ–términos tipificados de tipo τ
2. Si $M^{\sigma\to\tau}$ y N^σ son λ–términos tipificados de tipos $\sigma \to \tau$ y σ respectivamente, entonces el siguiente es un λ–término tipificado de tipo τ: $(M^{\sigma\to\tau}N^\sigma)^\tau$
3. Si x^σ es una variable de tipo σ y M^τ es un λ–término tipificado de tipo τ, entonces el siguiente es un λ–término de tipo $\sigma \to \tau$: $(\lambda x^\sigma.M^\tau)^{\sigma\to\tau}$

La operación de sustitución se define de la misma manera que para el caso sin tipos pero tomando en cuenta que una variable se puede sustituir por otra únicamente si ambas tiene el mismo tipo.

Definición 2.2.16 (Sustitución)**.** $[N^\sigma/x^\sigma](M^\tau)$ se define de la misma manera que para el caso sin tipos.

Nótese que no está definido $[N^\rho/x^\sigma](M^\tau)$ cuando $\rho \nexists \sigma$.

Todas las otras definiciones que dimos para el cálculo lambda sin tipos se pueden formular de la misma forma para el caso con tipos.

Ahora definimos la reducción β para el caso con tipos.

Definición 2.2.17 (Reducción β)**.** Definimos la relación reducción β denotada como $\twoheadrightarrow$ de la siguiente manera:

(α) $\lambda x^{\sigma}.M^{\tau} \twoheadrightarrow \lambda y^{\sigma}.[y^{\sigma}/x^{\sigma}]M^{\tau}$ si $y^{\tau} \notin FV(M^{\tau})$

(β) $((\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau}N^{\sigma})^{\tau} \twoheadrightarrow [N^{\sigma}/x^{\sigma}]M^{\tau}$

(ρ) $M^{\sigma} \twoheadrightarrow M^{\sigma}$ (reflexividad)

(μ) $\dfrac{M^{\sigma} \twoheadrightarrow N^{\sigma}}{P^{\sigma\to\tau}M^{\sigma} \twoheadrightarrow P^{\sigma\to\tau}N^{\sigma}}$ (monotonía derecha)

(ν) $\dfrac{M^{\sigma\to\tau} \twoheadrightarrow N^{\sigma\to\tau}}{M^{\sigma\to\tau}P^{\sigma} \twoheadrightarrow N^{\sigma\to\tau}P^{\sigma}}$ (monotonía izquierda)

(ξ) $\dfrac{M^{\tau} \twoheadrightarrow N^{\tau}}{\lambda x^{\sigma}.M^{\tau} \twoheadrightarrow \lambda x^{\sigma}.N^{\tau}}$

(τ) $\dfrac{M^{\sigma} \twoheadrightarrow N^{\sigma} \qquad N^{\sigma} \twoheadrightarrow P^{\sigma}}{M^{\sigma} \twoheadrightarrow P^{\sigma}}$ (transitividad)

Cabe señalar que el cálculo lambda simplemente tipificado cumple con el teorema 2.2.1 de Church-Rosser.[20]

Ahora con el propósito de establecer la propiedad de terminación definimos el concepto de normalización.

Definición 2.2.18 (Normalización)**.** Un término M se llama normalizable si tiene forma normal. Se llama fuertemente normalizable (*strong normalizable*) si todas las reducciones que comienzan en M tienen longitud finita.

Ahora enunciamos el teorema de normalización fuerte, que intuitivamente nos dice que toda evaluación de cualquier término finaliza. Este teorema no se cumple en el caso del cálculo lambda sin tipos que presentamos anteriormente pero sí en el cálculo lambda simplemente tipificado.

Teorema 2.2.2 (Normalización fuerte)**.** En el cálculo lambda simplemente tipificado, todo término es fuertemente normalizable, es decir, no existen reducciones β infinitas.[21]

Ejemplo 2.2.13. Ilustremos cómo en el cálculo lambda sin tipos puede haber reducciones infinitas, el ejemplo más simple es $\lambda x.(xx)\lambda x.(xx)$, entonces se tendría

$$\lambda x.(xx)\lambda x.(xx) \to \lambda x.(xx)\lambda x.(xx) \to \lambda x.(xx)\lambda x.(xx) \to \dots$$

en cambio es imposible expresar una versión de este término en el cálculo lambda simplemente tipificado, debido a que no es posible encontrar tipos que permitan construirlo según las reglas para construir términos de la definición 2.2.15.

[20] Por supuesto formulado de forma análoga para el caso con tipos.

[21] Para una demostración de este teorema consúltese [CFC58].

El cálculo de construcciones inductivas que es una extensión del cálculo lambda simplemente tipificado también cumple con el teorema de normalización fuerte [Wer94]. Por ello cuando Coq se utiliza como lenguaje de programación la evaluación de todo programa expresado en él termina.

En lo siguiente daremos un vistazo a cómo utilizar el cálculo lambda como sistema de demostración. En el camino presentaremos qué es un sistema de demostración y lo ilustraremos con el sistema de deducción natural para lo lógica proposicional intuicionista. Luego veremos cómo formular el cálculo lambda simplemente tipificado como sistema de demostración y observaremos la correspondencia entre la lógica y el cálculo lambda vía el isomorfismo de Curry Howard. No es nuestra intención abordar este tema a detalle pues este objetivo queda fuera del alcance del presente trabajo sino más bien proporcionar una mirada intuitiva de cómo el cálculo lambda se puede utilizar como sistema de demostración que nos será útil al utilizar Coq. Es por ello que omitiremos varios detalles y daremos por concedido que se cuentan con conocimientos previos de lógica clásica y sistemas de demostración. Para un tratamiento exhaustivo de este tema favor de consultar [SU06].

Comencemos viendo qué es un sistema de demostración.

Definición 2.2.19 (Sistema formal)**.** Un sistema formal (o sistema de demostración) tiene los siguientes componentes:

- Un alfabeto
- Una gramática
- Un conjunto de axiomas
- Un conjunto de reglas de inferencia

Un sistema de demostración que es muy conocido y ampliamente utilizado es el sistema de deducción natural.

A continuación presentamos el sistema de deducción natural para la lógica proposicional intuicionista.[22]

$$\frac{}{\Gamma, \varphi \vdash \varphi}$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \to\text{I} \qquad \frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \to\text{E}$$

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \wedge\text{I} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \wedge\text{E} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \vee\text{I} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} \qquad \frac{\Gamma \vdash \varphi \vee \psi \qquad \Gamma, \varphi \vdash \rho \qquad \Gamma, \psi \vdash \rho}{\Gamma \vdash \rho} \vee\text{E}$$

[22] Utilizaremos la lógica intuicionista porque Coq sigue el enfoque intuicionista.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \bot\text{E}$$

Definición 2.2.20. Un juicio en deducción natural es un par escrito como $\Gamma \vdash \varphi$ que se lee "Γ demuestra φ" donde Γ es un conjunto finito de fórmulas y φ es una fórmula.

En lógica clásica toda proposición es verdadera o falsa independientemente de cualquier otra cosa, esto es se cuenta con el principio del tercero excluido también conocido como *tertium non datur* que establece que $p \vee \neg p$ se cumple sin importar el valor de verdad de p. En cambio en la lógica intuicionista para afirmar la veracidad de una proposición es necesario dar una evidencia directa o "construcción" de que dicha proposición se cumple. Desde luego bajo esta visión constructiva no se acepta el principio del tercero excluido. Por otra parte tampoco se tiene que $\neg\neg p \to p$ que sí se cumple en lógica clásica.

En el initicionismo se considera la interpretación de Brouwer-Heyting-Kolmogorov (abreviada BHK) la cual establece lo siguiente:

- Una construcción de $\varphi_1 \wedge \varphi_2$ consiste de una construcción de φ_1 y una construcción de φ_2.
- Una construcción de $\varphi_1 \vee \varphi_2$ consiste de un indicador $i \in 1, 2$ y una construcción de φ_i.
- Una construcción de $\varphi_1 \to \varphi_2$ es un método (función) que transforma toda construcción de φ_1 en una construcción de φ_2.
- No existe construcción de $\bot$.

Por otro lado la negación $\neg\varphi$ se considera una equivalencia de $\varphi \to \bot$ y se puede explicar de la siguiente manera:

- Una construcción de $\neg\varphi$ es un método que transforma toda construcción de φ en un objeto inexistente.

La noción de "construcción" es informal y se puede entender de diferentes maneras. Obsérvese que las reglas de deducción natural pueden ser vistas como la formalización de la interpretación BHK donde "construcción" se puede leer como "demostración".

Entonces utilizando el sistema de deducción natural, en la lógica intuicionista no se tienen las siguientes reglas que sí forman parte de la lógica clásica:

$$\frac{\Gamma, \varphi \vdash \bot}{\neg\varphi} \qquad \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \neg\varphi}{\Gamma \vdash \bot} \qquad \frac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi}$$

Cuando se formula el cálculo lambda simplemente tipificado como sistema de demostración es más conveniente dar una presentación distinta a la que dimos anteriormente, no obstante ambas presentaciones son equivalentes como se ilustra en [Bar92].

Formularemos ahora el cálculo lambda como sistema de demostración dando una presentación alternativa más conveniente para este fin. Está presentación está basada en [Bar92].

Definición 2.2.21. Sea T un conjunto de tipos. El conjunto de λ-términos T-anotados (también llamados *pseudotérminos*), denotado como Λ_T se define mediante la siguiente gramática:

$$\begin{array}{rcll} \Lambda_T & \longrightarrow & x & \text{Variables} \\ & | & \Lambda_T \Lambda_T & \text{Aplicación} \\ & | & \lambda x : T.\Lambda_T & \text{Abstracción} \end{array}$$

Definición 2.2.22. El cálculo lambda simplemente tipificado se define como sigue:

- El conjunto de tipos T se define mediante la siguiente gramática:

$$\begin{array}{rcll} T & \longrightarrow & \sigma & \text{Variables de tipo} \\ & | & T \to T. & \text{Tipo función} \end{array}$$

- Un enunciado es de la forma $M : \sigma$ con un término $M \in \Lambda_T$ y un tipo $\sigma \in T$. El tipo σ es el predicado y el término M es el sujeto del enunciado.
- Una base Γ es un conjunto de enunciados donde únicamente se permiten variables distintas como sujetos.

Definición 2.2.23. Un enunciado $M : \sigma$ es derivable a partir de la base Γ, en notación $\Gamma \vdash M : \sigma$, si $M : \sigma$ se puede producir utilizando la siguientes reglas.

$$\frac{}{\Gamma \vdash x : \sigma}\ \text{Si } (x : \sigma \in \Gamma)$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.M) : (\sigma \to \tau)} \to\text{I} \qquad \frac{\Gamma \vdash M : (\sigma \to \tau) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \to\text{E}$$

Visto como sistema de demostración la sintaxis corresponde a la definición 2.2.21 donde se da la gramática que genera el conjunto de λ-términos T-anotados y los axiomas y reglas de inferencia a lo que se acaba de presentar.

Definición 2.2.24. El conjunto (legal) de λ-términos denotado como $\Lambda(\lambda_\to)$ se define como:

$$\Lambda(\lambda_\to) = \{M \in \Lambda_T | \exists \Gamma, \sigma \ \ \Gamma \vdash M : \sigma\}.$$

Ahora si consideramos el fragmento de la lógica proposicional intuicionista que contempla únicamente las reglas de la implicación y reescribimos las reglas que acabamos de presentar utilizando como nombre de variables φ y ψ en lugar de σ y τ como se muestra a continuación:

$$\frac{}{\Gamma, \varphi \vdash \varphi}$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \to\text{I} \qquad \frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \to\text{E}$$

Figura 2.12: Fragmento implicacional de la lógica proposicional intuicionista.

$$\frac{}{\Gamma, x : \varphi \vdash x : \varphi}$$

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash (\lambda x : \varphi . M) : \varphi \to \psi} \to\text{I} \qquad \frac{\Gamma \vdash M : \varphi \to \psi \qquad \Gamma \vdash N : \varphi}{\Gamma \vdash (MN) : \psi} \to\text{E}$$

Figura 2.13: Cálculo lambda simplemente tipificado.

podemos notar claramente cómo existe una correspondencia entre las proposiciones de la lógica y los tipos del cálculo lambda simplemente tipificado. Esto es lo que se conoce como el isomorfismo de Curry-Howard.

Debido también a esta correspondencia, realizar una demostración de una proposición φ en la lógica correspondería a construir (utilizando las reglas del cálculo lambda) un término de tipo φ en el cálculo lambda. Es decir, dada una proposición φ en la lógica ver si ésta se cumple equivaldría a encontrar un término t que habite el tipo φ o escrito de otra manera que se tenga $t : \varphi$.

Nótese cómo bajo esta visión es mucho más claro que se puede utilizar el cálculo lambda como sistema de demostración o más enfáticamente como una lógica (que era el objetivo original de su uso según lo estableció Church).

Por otra parte hemos considerado únicamente el fragmento de la lógica proposicional intuicionista para la implicación, para establecer la correspondencia de Curry-Howard. Si ahora deseamos considerar toda la lógica proposicional intuicionista que presentamos y queremos mantener esta correspondencia, es necesario extender el cálculo lambda simplemente tipificado con reglas correspondientes a la conjunción $\wedge$ y la disyunción $\vee$, que en términos de tipos significa agregar los tipos producto y suma respectivamente. Lo cual realizamos a continuación.

$$\frac{}{\Gamma, x : \varphi \vdash x : \varphi}$$

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash (\lambda x : \varphi . M) : \varphi \to \psi} \qquad \frac{\Gamma \vdash M : \varphi \to \psi \qquad \Gamma \vdash N : \varphi}{\Gamma \vdash (MN) : \psi}$$

$$\frac{\Gamma \vdash M : \varphi \qquad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N \rangle : \varphi \wedge \psi} \qquad \frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \pi_1(M) : \varphi} \qquad \frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \pi_2(M) : \psi}$$

$$\frac{\Gamma \vdash M : \varphi}{\text{in}_1^{\varphi \vee \psi}(M) : \varphi \vee \psi} \qquad \frac{\Gamma \vdash M : \psi}{\text{in}_2^{\varphi \vee \psi}(M) : \varphi \vee \psi} \qquad \frac{\Gamma \vdash L : \varphi \vee \psi \qquad \Gamma, x : \varphi \vdash M : \rho \qquad \Gamma, y : \psi \vdash N : \rho}{\Gamma \vdash (\text{case } L \text{ of } [x]M \text{ or } [y]N) : \rho}$$

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \varepsilon_\varphi(M) : \varphi}$$

Figura 2.14: Cálculo lambda simplemente tipificado, extendido con tipos producto y suma.

Obsérvese que en las reglas para la conjunción hemos escrito el tipo producto como $\varphi \wedge \psi$ para resaltar la correspondencia con las reglas de la conjunción de la lógica, no

obstante en teoría de tipos el tipo producto se escribe como $\varphi \times \psi$. Lo mismo vale para el caso de la disyunción donde escribimos el tipo suma como $\varphi \vee \psi$ y en teoría de tipos se escribe como $\varphi + \psi$.

Hemos visto ya que los tipos producto corresponden a la conjunción y los tipos suma corresponden a la disyunción ahora podemos ser más ambiciosos y buscar una forma de extender la correspondencia de Curry-Howard a la lógica de predicados. Con este objetivo Howard y De Bruijn introdujeron tipos dependientes de la forma $A(x)$ que corresponden a predicados $P(x)$. Luego establecieron productos indexados $\Pi x : X.A(x)$ que corresponden a la cuantificación universal $\forall x \in X.A(x)$ y sumas indexadas $\Sigma x : X.A(x)$ que corresponden a la cuantificación existencial $\exists x \in X.A(x)$. De esta manera llegamos a los tipos dependientes Π y Σ. Bajo la interpretación BHK una construcción de $\forall x \in X.A(x)$ es un método que transforma todo elemento $a \in X$ en una construcción de $A(a)$ y una construcción de $\exists x \in X.A(x)$ es un par $\langle a, p\rangle$ tal que $a \in X$ y p es una construcción de $A(a)$ (para más detalles acerca de esto se recomienda consultar [BG01; AGN09]).

Supongamos que por alguna razón deseamos considerar los naturales mayores que 0. Entonces podemos hacer uso de un tipo dependiente para expresarlos, de la siguiente manera:

$$\Sigma x{:}\mathsf{nat}.x \geq 0$$

de esta forma si consideramos su construcción tendríamos un par $\langle a, p\rangle$ donde $a : \mathsf{nat}$ y $p : a \geq 0$, es decir, a es un natural y p es una demostración de que a es mayor que 0. Notemos que en lugar de considerar una demostración p podemos considerar una distinta q o sea una demostración diferente de que a es mayor que 0 en realidad nos es irrelevante la demostración que se ofrezca por lo que bajo este enfoque toda demostración se considera equivalente, esto es lo que se conoce como el *principio de irrelevancia de la prueba*. Por otra parte notemos cómo a es por un lado un objeto de tipo nat pero también por otro es un término que representa un número natural y por tanto cuando estudiemos Coq veremos que podemos decir que tiene contenido computacional, por su parte p es un término que representa una demostración de una proposición por lo que no tiene contenido computacional.[23] Por otro lado notemos cómo el objeto depende de la demostración por eso es que se llaman tipos dependientes.

Entonces podemos decir que haciendo uso de un tipo dependiente es posible expresar que existe un objeto de un tipo cualquiera que cumple con una proposición (que podemos ver como una especificación) y hacer una construcción de este tipo implica dar una construcción del objeto y una demostración de que cumple la proposición. Con fines prácticos de la construcción de este tipo se puede obtener la construcción del objeto que tiene contenido computacional y descartar la demostración de la proposición que no tiene contenido computacional porque resulta irrelevante (ya que basta con que se haya demostrado ya que el objeto cumple con la propiedad).

Hasta este momento hemos mostrado ya el cálculo lambda simplemente tipificado como lenguaje de programación que cuenta con la propiedad de terminación. Por otro lado también vimos cómo utilizar este cálculo como sistema de demostración. Adicionalmente

[23] Obsérvese que en Coq el usuario sería el encargado de ofrecer tanto a como p.

mostramos cómo es que existe una correspondencia entre el fragmento implicacional de la lógica intuicionista y el cálculo lambda simplemente tipificado vía el isomorfismo de Curry-Howard. También vimos cómo el cálculo lambda simplemente tipificado se puede extender con los tipos producto y suma para que de esta manera la correspondencia también contemple la conjunción y la disyunción presentes en la lógica. En ese mismo sentido el cálculo de construcciones inductivas se puede considerar como el cálculo simplemente tipificado con diferentes extensiones, una de las más importantes es el uso de tipos dependientes, es por eso que se dice que Coq es una asistente de pruebas basado en la correspondencia de Curry-Howard. Cabe mencionar que el cálculo de construcciones inductivas también cuenta con al propiedad de terminación. Con lo anterior contamos ya con una noción de los fundamentos teóricos sobres los que Coq descansa y que nos serán de gran ayuda para comprender algunas de las características con las que éste cuenta. Estudiaremos propiamente Coq más adelante en el capítulo 3.

Con esto damos por concluido nuestro estudio preliminar del cálculo lambda y veremos ahora cómo se puede especificar formalmente la semántica de un lenguaje de programación.

2.3. Semántica formal

Originalmente las computadoras se programaban en lenguaje de máquina y posteriormente en lenguaje ensamblador, en ambos casos, esto resultaba en un estilo precario de programación con diversos inconvenientes, algunos de los más destacados eran: tener que escribir secuencias de instrucciones básicas que muchas veces resultaban repetitivas y tediosas de escribir, proceso en el cual no era difícil cometer errores; un programa servía únicamente para una arquitectura específica, si se requería que el programa se ejecutara en otra computadora entonces se tenía que reescribir con código específico de la otra arquitectura; entre muchos otros.

Por otra parte, la forma natural de dar un algoritmo[24] es como una secuencia de pasos con un nivel de abstracción alto.[25] Tomando esto en cuenta y como respuesta a todos los problemas que representaba programar en bajo nivel, surgieron los llamados lenguajes de alto nivel haciendo referencia a un alto nivel de abstracción.

Con la llegada de los lenguajes de alto nivel surgió la necesidad de traducir un lenguaje (de alto nivel) a código de máquina. En efecto, al inicio esto se veía como la compilación de diferentes rutinas escritas en lenguaje ensamblador y de ahí se acuñó el nombre *compilador* al programa que realizaba esta tarea.

Un lenguaje de programación cuenta con una sintaxis que permite al programador escribir un programa, esta sintaxis la conforman los diferentes enunciados soportados por el lenguaje por ejemplo, enunciados if y secuencias. Pero a un programador no le basta con poder escribir un programa sino que además necesita conocer el efecto que tendrá

[24] O un procedimiento en general.

[25] En comparación con el nivel de las instrucciones de máquina las cuales claramente son operaciones básicas con un nivel de abstracción bajo.

el ejecutar ese programa, es decir, con un enfoque operacional el comportamiento que tendrá el programa. Entonces si el programador conoce el comportamiento de cada uno de los enunciados que soporta el lenguaje tendrá el conocimiento necesario para poder escribir el programa que desea. En el caso de un escritor de compiladores el conocer el comportamiento de cada uno de los enunciados del programa le permite (entre otras cosas) saber qué código generar para cada enunciado específico. Por otra parte si tomamos un enfoque más abstracto entonces nos gustaría conocer el significado o interpretación de un programa ya que esto nos permitiría entre otras cosas establecer propiedades y en su caso también demostrarlas, por ejemplo la equivalencia entre dos programas. Ambos enfoques forman parte de la semántica de un lenguaje de programación.

Por su puesto que una descripción informal de la semántica no resulta satisfactoria.

Con base en las necesidades anteriores se han desarrollado tres principales estilos de semántica formal:

- Semántica operacional: En ésta se utiliza una máquina abstracta, la cual se define dando para cada una de sus instrucciones el estado actual de la máquina y el estado siguiente que refleja los cambios resultantes tras haberse efectuado la instrucción correspondiente. La idea es que la máquina sea tan simple que no haya posibilidad de confusión o ambigüedad al ejecutar su código. Así, para especificar la semántica de un lenguaje de programación, se da la compilación de cada uno de los enunciados del lenguaje a la máquina. Luego para determinar de forma precisa el efecto que tiene un programa, basta con estudiar cómo se ejecuta el código de máquina correspondiente a este programa en la máquina.

- Semántica denotativa: En ésta, para cada una de las categorías sintácticas del lenguaje se da una función de evaluación, la cual asigna valores abstractos (como pueden ser números, valores de verdad o funciones) que la correspondiente categoría denota. También se le conoce como semántica matemática.

- Semántica axiomática: En ésta, para cada uno de los enunciados soportados por el lenguaje se da un regla que enuncia las propiedades que se cumplen en el caso en que dicho enunciado se ejecute, tales propiedades se dan en términos de lo que se puede afirmar antes de que se realice la ejecución del enunciado correspondiente.

Por supuesto, cada tipo de semántica sirve mejor para ciertos propósitos y así, con base en las necesidades específicas de un usuario, es que se suele elegir una de éstas.[26] De esta forma estos tres tipos diferentes de semántica son complementarios.

La semántica denotativa surgió con base en la semántica en el sentido en que se utiliza en lógica y es claro que sigue más cercanamente la motivación de tener una interpretación del lenguaje. Por su parte, la semántica operacional resulta ser más intuitiva y sigue más de cerca la motivación de expresar el comportamiento de un lenguaje y ésta desde luego resulta útil para guiar implementaciones. Realizar una implementación es lo que nos ocupa aquí y por eso nosotros utilizaremos la semántica operacional.

[26] O desarrollar una nueva tomando como base alguna de éstas o una completamente diferente.

Cabe mencionar que la semántica de un lenguaje de programación está conformada por:

- Semántica estática en la cual se establecen las propiedades con las que cuentan los enunciados de un lenguaje antes de la ejecución de los mismos.
- Semántica dinámica en la que se describe el comportamiento en tiempo de ejecución de cada uno de los enunciados que conforman el lenguaje.

Usualmente la semántica estática principalmente se refiere a especificar el sistema de tipos del lenguaje y las propiedades que éste cumple, por su parte la semántica dinámica se refiere a especificar las reglas de evaluación del lenguaje.

2.3.1. Semántica operacional estructural

La idea original de la semántica operacional era poder especificar el comportamiento de un lenguaje de programación. Para lo cual la estrategia a seguir era para cada uno de los enunciados de un lenguaje dar su compilación a instrucciones de una máquina abstracta y luego observar su comportamiento en términos de los estados de la máquina al ejecutar dichas instrucciones. La idea es que la máquina fuese lo más simple posible para que toda persona pudiese comprenderla. Así se tendría en términos de la máquina el comportamiento de cada uno de los enunciados del lenguaje por ejemplo un if o una secuencia.

Luego Plotkin en [Plo81] vislumbra una forma de especificar la semántica de un lenguaje, en la cual se especifica el comportamiento de cada uno de los enunciados del lenguaje directamente sobre el enunciado mismo sin tener que hacer una compilación ni utilizar una máquina abstracta que es el método que explicaremos en esta sección.

Plotkin crítica el uso de una máquina abstracta en la semántica operacional argumentando que hacer uso de ella no se trata de una formalización de las ideas intuitivas operacionales sino que más bien utilizar una máquina abstracta es correcto dadas esas ideas intuitivas. Por eso desarrolla un nuevo método dando respuesta a esa crítica en el que no se utiliza una máquina abstracta.

A continuación presentaremos la idea general del método de Plotkin.

Para ilustrar dicho método utilizaremos un ejemplo simple e intuitivo.

Consideremos un lenguaje de expresiones aritméticas sobre enteros y consideremos únicamente por simplicidad el caso para la adición.[27]

Así (suponiendo que ya se ha realizado el análisis léxico y sintáctico) nuestra sintaxis abstracta es:[28]

e	::=	Const n	$n \in \mathbb{N}$	Constantes naturales
	|	Plus $e\ e$		Adición

[27] El caso para los otros operadores es análogo.

[28] Nótese que esta gramática define la sintaxis abstracta del lenguaje y no se debe confundir con la que define la sintaxis concreta de un lenguaje como la que presentamos en la figura 1.1.

Ahora debemos definir la máquina abstracta que utilizaremos como objetivo.

Para definir una máquina básicamente lo que necesitamos es dar su conjunto de instrucciones y el comportamiento de cada una de éstas, es decir, el efecto que tendrán éstas sobre los componentes de la máquina cuando se ejecuten.

En este caso nuestra máquina será muy simple y sólo contará con las siguientes instrucciones:

inst	::=	IConst n	$n \in \mathbb{N}$	instrucción de carga de una constante en la pila
	|	IAdd		instrucción de adición

Entonces el código de máquina es una secuencia de instrucciones. Para representar esta secuencia en abstracto utilizaremos la notación usual de secuencia, por ejemplo, $[\text{IConst } 5, \text{IConst } 2, \text{IAdd}]$ es una secuencia de instrucciones (correspondiente al código de máquina que calcula la adición de 5 y 2). Para denotar la concatenación de secuencias utilizaremos el símbolo "$\cdot$", así el ejemplo anterior sería equivalente a escribir: $[\text{IConst } 5] \cdot [\text{IConst } 2, \text{IAdd}]$. Por otra parte (en algunas ocasiones) cuando una secuencia esté conformada por un único elemento abusaremos de la notación y escribiremos por ejemplo $\text{IConst } 5 \cdot [\text{IConst } 2, \text{IAdd}]$ por decir $[\text{IConst } 5] \cdot [\text{IConst } 2, \text{IAdd}]$.

Entonces nuestra máquina estará conformada por una sección para el código y una pila de evaluación, es decir, $M = \langle C, S \rangle$.

Ahora para poder dar de manera formal el comportamiento de nuestra máquina primero presentaremos la definición de sistema de transición.

Definición 2.3.1. Un sistema de transición es un par $\langle \Gamma, \to \rangle$ donde Γ es un conjunto de elementos γ llamados configuraciones y $\to \subseteq \Gamma \times \Gamma$ es una relación binaria llamada relación de transición.

De esta manera $\gamma \to \gamma'$ denota una transición de la configuración γ a la configuración γ'.

Por su parte $\to^+$ denota la cerradura transitiva de la relación de transición, mientras que $\to^*$ la cerradura transitiva y reflexiva de la misma.

Entonces aplicado a nuestra máquina, las configuraciones serían de la forma $\gamma = \langle c, s \rangle$ donde $c \in C$ y $s \in S$. Nótese cómo una configuración representa un estado de la máquina.

Ahora toca el turno de definir la relación de transición "$\to$", lo cual hacemos de la siguiente manera:

- $([\text{IConst } n] \cdot c, s) \to (c, [n] \cdot s)$
- $([\text{IAdd}] \cdot c, [n_2, n_1] \cdot s) \to (c, [n_1 + n_2] \cdot s)$

donde $c \in C$ es un código cualquiera y $s \in S$ una pila cualquiera.

En realidad cuando se define una máquina, en lugar de presentar las transiciones como lo acabamos de hacer se suele hacer gráficamente mediante una tabla donde del lado izquierdo se pone el estado actual de la máquina y del lado derecho el estado siguiente

(al que transita) como se muestra a continuación (pero formalmente significa definir las transiciones de la forma en que lo hicimos).

Configuración actual		Configuración siguiente	
Código	Pila	Código	Pila
[IConst n] $\cdot\, c$	s	c	$[n] \cdot s$
[IAdd] $\cdot\, c$	$[n_2, n_1] \cdot s$	c	$[n_1 + n_2] \cdot s$

Habiendo definido ya nuestra objetivo podemos presentar la compilación lo cual hacemos a continuación.[29]

$$Compile(e) = \begin{cases} [\text{IConst } n] & \text{si } e = \text{Const n} \\ Compile\ e_1 \cdot Compile\ e_2 \cdot [\text{IAdd}] & \text{si } e = \text{Plus } e_1\ e_2 \end{cases}$$

Tomemos ahora un momento para analizar cómo se evalúa una expresión. Por ejemplo fijémonos en $5 + 2 + 1$, lo primero que debemos decir es que sabemos por convención que la adición es asociativa a la izquierda o sea la expresión anterior en realidad se trata de $(5 + 2) + 1$, luego para evaluar la expresión primero se debe evaluar la adición principal (la de la derecha correspondiente al segundo +) a su vez para evaluar ésta primero se debe evaluar su operando izquierdo que se trata de la adición $5 + 2$ para evaluar esta última evaluamos primero su operando izquierdo que es 5 y se evalúa a 5 y luego el derecho que es 2 y se evalúa a 2 después se realiza la adición y se obtiene 7 con esto ya tenemos la evaluación del operando izquierdo de la suma principal, luego continuamos evaluado el operando derecho de dicha suma que es 1 y se evalúa a 1 y finalmente se realiza la adición y se obtiene como resultado final 8. Generalizando, esta estrategia de evaluación corresponde a, para una operación dada evaluar recursivamente su operando izquierdo, luego el derecho y finalmente realización la operación. La estrategia anterior correspondería a realizar un recorrido en posorden sobre el AST de la expresión tal como lo vimos en la sección 2.1.3.

Para realizar la compilación se sigue la misma estrategia, solo que en lugar de evaluar se compila, es decir, se emite la instrucción correspondiente para que en un futuro la máquina se encargue de evaluarlo. Justo esta estrategia de compilación es la que está reflejada en nuestra función $Compile$. Así en nuestro ejemplo la expresión $5{+}2{+}1$ en la sintaxis abstracta de nuestro lenguaje se escribiría Plus (Plus (Const 5)(Const 2))(Const 1) y para compilarla primero compilaríamos el operando izquierdo de la adición principal que es una adición, a su vez para compilar esta última compilaríamos su operando izquierdo que es Const 5 para lo cual se emitiría la instrucción IConst 5, luego el derecho y tendríamos [IConst 5, IConst 2] luego la adición y quedaría [IConst 5, IConst 2, IAdd] posteriormente se compilaría el operando derecho de la adición principal y obtendríamos [IConst 5, IConst 2, IAdd, IConst 1] y finalmente la adición principal con lo que al final se

[29] En realidad Plotkin en [Plo81] utiliza una máquina abstracta que trabaja directamente sobre los enunciados del lenguaje por lo que no da una compilación ya que en ese caso no es necesaria.

tendría [IConst 5, IConst 2, IAdd, IConst 1, IAdd]. Nótese cómo en el código de máquina que se obtuvo el orden de evaluación está explícito (que se podría considerar como instrucciones de máquina correspondientes a la expresión en notación posfija). Así que cuando la máquina evalúe el código que generó el compilador correspondiente a una expresión simplemente tiene que evaluar las instrucciones linealmente en el orden en que se encuentran. Las transiciones de la máquina correspondientes a la evaluación del código de nuestro ejemplo son las siguientes:

1) ([IConst 5, IConst 2, IAdd, IConst 1, IAdd], s) $\rightarrow$
2) ([IConst 2, IAdd, IConst 1, IAdd], $[5] \cdot s$) $\rightarrow$
3) ([IAdd, IConst 1, IAdd], $[2, 5] \cdot s$) $\rightarrow$
4) ([IConst 1, IAdd], $[7] \cdot s$) $\rightarrow$
5) ([IAdd], $[1, 7] \cdot s$) $\rightarrow$
6) ([], $[8] \cdot s$) $\rightarrow$

Obsérvese cómo al final de la evaluación no hay más instrucciones por evaluar y el resultado de la evaluación queda en el tope de la pila.

Podemos decir que el método que acabamos de ilustrar era en lo que originalmente consistía la semántica operacional. Esto es para especificar formalmente la semántica de un lenguaje utilizando la semántica operacional, se definía la sintaxis del lenguaje, luego se definía la máquina (incluyendo desde luego la definición del comportamiento de cada una de sus instrucciones en términos de transiciones de la máquina) y se daba la compilación de cada uno de los enunciados del lenguaje a código de la máquina. De esta forma para estudiar el comportamiento de un enunciado del lenguaje por ejemplo un if se veía a qué código se compilaba y luego se estudiaba el comportamiento de este enunciado observando los estados por los que transitaba la máquina al evaluar el código.

Luego Plotkin observó que no era necesario utilizar una máquina para poder estudiar el comportamiento de un enunciado del lenguaje, el comportamiento de éste se podía especificar directamente sobre la sintaxis abstracta correspondiente al enunciado. Para llegar a este resultado primero estudió que al evaluar el código de máquina correspondiente a la compilación de un enunciado del lenguaje muchos de los pasos (transiciones) que realizaba la máquina eran completamente irrelevantes para capturar la esencia del comportamiento de dicho enunciado y se fijo exclusivamente en aquellos pasos que sí resultaban esenciales, luego se dio cuenta que esos pasos los podía expresar directamente sobre la sintaxis abstracta del lenguaje y además que podía especificarlos haciendo uso de la misma notación que se utiliza para denotar reglas en lógica. A continuación mostraremos el razonamiento que siguió Plotkin.

Tomemos como punto de partida la expresión $5+(2+7)+(3+9)$ cuya sintaxis abstracta es:

Plus (Plus (Const 5) (Plus (Const 2) (Const 7))) (Plus (Const 3) (Const 9)).

Por su parte el código correspondiente a su compilación es:

$$[\text{IConst } 5, \text{IConst } 2, \text{IConst } 7, \text{IAdd}, \text{IAdd}, \text{IConst } 3, \text{IConst } 9, \text{IAdd}, \text{IAdd}]$$

Luego al evaluar este código en la máquina se realizan las siguientes transiciones:

	1)	([IConst 5, IConst 2, IConst 7,IAdd, IAdd, IConst 3, IConst 9, IAdd, IAdd], s)	$\rightarrow$
	2)	([IConst 2, IConst 7,IAdd, IAdd, IConst 3, IConst 9, IAdd, IAdd], $[5] \cdot s$)	$\rightarrow$
	3)	([IConst 7,IAdd, IAdd, IConst 3, IConst 9, IAdd, IAdd], $[2, 5] \cdot s$)	$\rightarrow$
	4)	([IAdd, IAdd, IConst 3, IConst 9, IAdd, IAdd], $[7, 2, 5] \cdot s$)	$\rightarrow$
$\star$	5)	([IAdd, IConst 3, IConst 9, IAdd, IAdd], $[9, 5] \cdot s$)	$\rightarrow$
$\star$	6)	([IConst 3, IConst 9, IAdd, IAdd], $[14] \cdot s$)	$\rightarrow$
	7)	([IConst 9, IAdd, IAdd], $[3, 14] \cdot s$)	$\rightarrow$
	8)	([IAdd, IAdd], $[9, 3, 14] \cdot s$)	$\rightarrow$
$\star$	9)	([IAdd], $[12, 14] \cdot s$)	$\rightarrow$
$\star$	10)	([], $[26] \cdot s$)	$\rightarrow$

De aquí, podemos notar claramente que los pasos realmente relevantes son aquellos que tienen una $\star$, en los cuales si hacemos un símil con la noción de redex y reducto del cálculo lambda se realiza una contracción. Es en estos pasos que se captura la esencia del comportamiento del programa porque en éstos es en donde se realiza propiamente un cálculo. Ahora veamos que los podemos expresar directamente sobre la sintaxis abstracta del programa (es decir, sin la necesidad de compilar y de utilizar una máquina) de la siguiente manera:

1)	Plus (Plus (Const 5) (Plus (Const 2) (Const 7))) (Plus (Const 3) (Const 9))	$\rightarrow$
2)	Plus (Plus (Const 5) (Const 9)) (Plus (Const 3) (Const 9))	$\rightarrow$
3)	Plus (Const 14) (Plus (Const 3) (Const 9))	$\rightarrow$
4)	Plus (Const 14) (Const 12)	$\rightarrow$
5)	Plus (Const 26)	$\rightarrow$

A estas secuencias de transiciones se les llama secuencias de reducción (*reduction sequences*).

Entonces ahora podemos delinear informalmente esta estrategia de evaluación directamente sobre la sintaxis abstracta del lenguaje de la siguiente manera:

- Const n se considera evaluado, consigo mismo como valor.
- Plus e_1 e_2
  1. Evaluar e_1 obteniendo un Const n_1 como resultado
  2. Evaluar e_2 obteniendo un Const n_2 como resultado
  3. Sumar n_1 y n_2 obteniendo un n_3 como resultado de esta suma y entonces el resultado final de la evaluación será Const n_3

Así, para evaluar una expresión e tenemos secuencias de transiciones de la forma:
$e = e_1 \to \ldots e_i \to \ldots e_n = n$ donde n es el resultado.
Ahora, si únicamente nos fijamos en el primer paso, entonces tenemos que:

1. Si e_1 no es un número (un Const n) el primer paso de la evaluación de Plus e_1 e_2 es el primer paso de la evaluación de e_1

2. Si e_1 es un número, pero e_2 no lo es, entonces el primer paso de la evaluación de Plus e_1 e_2 es el primer paso de la evaluación de e_2

3. Si e_1 y e_2 son números entonces el primero (y último) paso de la evaluación de Plus e_1 e_2 es la suma de e_1 con e_2.

Esto da como resultado reglas para establecer relaciones binarias de la forma $e \to e'$ donde e' es el resultado del primer paso de la evaluación de e.

Así, finalmente podemos formular para la adición la siguientes reglas:[30]

1. $$\frac{e_1 \to e_1'}{\text{Plus } e_1\ e_2 \to \text{Plus } e_1'\ e_2}$$

2. $$\frac{e_2 \to e_2'}{\text{Plus (Const } n_1)\ e_2 \to \text{Plus (Const } n_1)\ e_2'}$$

3. $$\frac{}{\text{Plus (Const } n_1) \text{ (Const } n_2) \to \text{ Const } n_1 + n_2}$$

Nótese que, en lugar de dar reglas para un único paso, podemos dar reglas directamente sobre las secuencias de reducción, es decir, directamente sobre la evaluación.[31] Sin embargo, Plotkin menciona que, no obstante, axiomatizar un paso es intuitivamente más simple, pero deja abierta la posibilidad de considerar el "tamaño" de paso adecuado según sea conveniente,[32] por ejemplo, presenta una regla para el If de la siguiente manera:

1. $$\frac{b \to^* true}{\text{If } b\ c_1\ c_2 \to c_1}$$

2. $$\frac{b \to^* false}{\text{If } b\ c_1\ c_2 \to c_2}$$

Como podemos observar de este modo ya no es necesario utilizar una máquina abstracta ni dar la compilación del lenguaje al código de ésta. Bajo este esquema basta con dar la sintaxis abstracta del lenguaje y luego dar las reglas correspondientes a cada uno

[30] Utilizando la notación que se utiliza al expresar reglas en lógica.

[31] Más adelante veremos que este es el enfoque que se sigue en la semántica natural también llamada de paso grande.

[32] Sin embargo, cuando se suele hacer alusión a esta semántica se da por hecho que se trabaja con un paso simple.

de los enunciados que conforman el lenguaje. Estas reglas especifican formalmente el comportamiento de cada uno de los enunciados del lenguaje.

Utilizando este método también se pueden dar las reglas del sistema de tipos del lenguaje, tal como lo muestra Plotkin en [Plo81].

A esta semántica, se le conoce como semántica de reducción (*reduction semantics*), semántica de paso pequeño (*small step semantics*) o semántica operacional estructural (*structural operational semantics*).[33]

2.3.2. Semántica natural

Con base en el trabajo de Plotkin, Kahn [Kah87] busca un método unificador que permita expresar no sólo la semántica dinámica y la semántica estática de un lenguaje sino que además permita expresar su compilación.

Como se menciona en [Kah87] la idea general de este tipo de semántica es dar axiomas y reglas de inferencia que caractericen los diferentes predicados semánticos[34] por definir sobre una expresión e.

Así por ejemplo, en la semántica estática, si se quiere expresar que una expresión e tiene tipo τ en un entorno Γ, entonces se axiomatiza como:

$$\Gamma \vdash e : \tau$$

donde, el entorno Γ es una colección de suposiciones sobre los tipos de las variables de la expresión e. Para especificar una traducción de un lenguaje L_1 a un lenguaje L_2, se dan reglas de la forma:

$$\Gamma \vdash e_1 \twoheadrightarrow e_2$$

donde, Γ contiene las suposiciones sobre los identificadores presentes en e_1. Por otra lado, como se espera, la expresión e_1 está en lenguaje fuente L_1 y la expresión e_2 está en el lenguaje objetivo L_2. Por su parte, en la semántica dinámica puede haber variaciones en el estilo, dependiendo de las propiedades del lenguaje a describir, en los lenguajes más simples es suficiente expresar que la evaluación de una expresión e en un estado (o configuración) s_1 lleva a un estado nuevo s_2. Este predicado se escribe como:

$$s_1 \vdash e \Rightarrow s_2$$

donde s_1 (entre otras cosas) contiene los valores de las identificadores que aparecen en e.

Una definición semántica, es una lista de axiomas y reglas de inferencia que define uno de los predicados anteriores. En otras palabras, una definición semántica se identifica con una lógica y razonar sobre el lenguaje es probar teoremas dentro de esta lógica. Por ejemplo, dado un estado s_1 y un programa e, ¿existe un estado s_2 tal que $s_1 \vdash e \Rightarrow s_2$ se cumple?

[33] Para un tratamiento más detallado de esta semántica consúltese [Plo81].

[34] Estos predicados semánticos están pensados que expresen la semántica estática, la semántica dinámica y la compilación de un lenguaje.

Aquí vale la pena resaltar un punto:[35] dado que en esta semántica la presentación es inherentemente relacional en lugar de funcional, en el caso general es no determinista.

Como se puede notar, las reglas son muy parecidas a las reglas de deducción natural, de aquí que se acuñó el nombre *semántica natural* (*natural semantics*).

Ilustramos ahora el funcionamiento de la semántica natural utilizando nuestro ejemplo de un lenguaje de expresiones aritméticas, considerando únicamente el caso para la adición.

La sintaxis abstracta es:

$$\begin{array}{rcl} e & ::= & \text{Const } n \\ & | & \text{Plus } e\ e \end{array}$$

Ahora como en esta semántica intuitivamente para un enunciado del lenguaje se especifica su evaluación a un valor final, entonces podemos tener explícitamente una categoría que exprese todos los posibles valores finales de una evaluación. De esta manera en las reglas de la semántica se establecerá una relación que va de los enunciados del lenguaje a valores, donde estos últimos representan los valores finales de su evaluación.

Entonces los valores para nuestro lenguaje de ejemplo son:

$$\begin{array}{rcl} v & ::= & \text{Num } n \end{array}$$

es decir, para el caso particular de nuestro lenguaje de ejemplo hay una única forma que pueden tomar lo valores finales que es Num n. Presentamos ahora la semántica (dinámica) del lenguaje:

$$\frac{}{\text{Const } n \Rightarrow \text{Num } n} \qquad \frac{e_1 \Rightarrow \text{Num } n_1 \qquad e_2 \Rightarrow \text{Num } n_2}{\text{Plus } e_1\ e_2 \Rightarrow \text{Num } n_1 + n_2}$$

Nótese cómo en este caso el resultado siempre nos devuelve un valor final en comparación con la semántica de paso pequeño, la cual nos puede devolver una expresión (con un paso menos por realizar) susceptible de seguirse evaluando, es decir, en esta semántica siempre[36] se devuelve el valor final de una expresión en un paso (que corresponde a muchos pasos de la semántica de paso pequeño), por eso también se le conoce como semántica de paso grande (*big step semantics*).[37]

Ahora debemos definir la máquina abstracta que utilizaremos y en efecto, también podemos usar la semántica natural con este propósito, es decir, para especificar la semántica de la máquina, así, la definición de la máquina es la siguiente:

$$\frac{[n] \cdot s \vdash c \Rightarrow s_f}{s \vdash [\text{IConst } n] \cdot c \Rightarrow s_f} \qquad \frac{[n_1 + n_2] \cdot s \vdash c \Rightarrow s_f}{[n_2, n_1] \cdot s \vdash [\text{IAdd}] \cdot c \Rightarrow s_f}$$

[35] Que será fundamental para nuestros propósitos más adelante, cuando formalicemos nuestro trabajo en Coq.

[36] Aquí nos referimos a siempre que el programa termine que es el único caso que consideraremos en este trabajo.

[37] Que como vimos, Plotkin en [Plo81] ya había contemplado esta posibilidad pero prefirió axiomatizar un paso simple.

Podemos decir en este caso que los estados de la máquina (configuraciones) son de la forma $\langle c, s\rangle$ donde c es un código y s una pila y entonces la semántica establece una relación que va de estados a estados finales, donde del lado izquierdo de "$\vdash$" se escribe la pila actual (el tope de la pila está a la izquierda) y del lado derecho el código actual por evaluar, s_f representa un estado final, donde un estado final es aquel en el que no hay más código por evaluar.

Nótese que hasta el momento, hemos considerado que el código de una máquina abstracta es una secuencia de instrucciones, por ejemplo: [IConst 5, IConst 2, IAdd]. Otra posibilidad es considerar explícitamente en la máquina una instrucción "ISeq" que denote una operación de secuencia en la máquina, así el ejemplo anterior se escribiría: ISeq (ISeq (IConst 5) (IConst 2)) IAdd.

Siguiendo este enfoque la definición de la máquina es:

$$\frac{}{s \vdash \text{IConst } n \Rightarrow [n] \cdot s} \qquad \frac{}{[n_2, n_1] \cdot s \vdash \text{IAdd} \Rightarrow [n_1 + n_2] \cdot s}$$

$$\frac{s \vdash c_1 \Rightarrow s_1 \qquad s_1 \vdash c_2 \Rightarrow s_2}{s \vdash \text{ISeq } c_1\ c_2 \Rightarrow s_2}$$

En este caso la relación que establece la semántica va de estados (configuraciones $\langle c, s\rangle$ donde c es una instrucción y s una pila) a pilas. Vale la pena resaltar que aquí un programa de la máquina no es un código entendido como una secuencia de instrucciones como lo habíamos considerado sino que bajo este enfoque un programa de la máquina es una única instrucción (compuesta por sub-instrucciones).[38]

En efecto, este es el enfoque que sigue Khan en [Kah87].

Nosotros hemos presentado la definición de la máquina donde el código es una secuencia de instrucciones utilizando semántica natural y hasta donde el conocimiento del autor abarca esta presentación es original.

Toca el turno de presentar la compilación. Como ahora contamos con dos definiciones posibles de la máquina presentaremos la compilación para cada uno de estos casos.

Primero damos la compilación hacia una máquina donde el código es una secuencia de instrucciones:

$$\frac{}{\text{Const } n \rightarrowtail [\text{IConst } n]} \qquad \frac{e_1 \rightarrowtail c_1 \qquad e_2 \rightarrowtail c_2}{\text{Plus } e_1\ e_2 \rightarrowtail c_1 \cdot c_2 \cdot [\text{IAdd}]}$$

Ahora damos la compilación hacia la máquina que cuenta con una instrucción "ISeq" explícita:

$$\frac{e_1 \rightarrowtail c_1 \qquad e_2 \rightarrowtail c_2}{\text{Plus } e_1\ e_2 \rightarrowtail \text{ISeq (ISeq } c_1\ c_2) \text{ IAdd}}$$

Solo damos la regla para el enunciado Plus ya que la regla para el enunciado Const es la misma que para la máquina anterior.

[38] Sin embargo, este no es el enfoque que se sigue en las máquinas reales donde, como en nuestro esquema original, un programa es una secuencia de instrucciones.

Hemos mostrado con este ejemplo simple cómo siguiendo el enfoque unificador de Khan podemos utilizar la semántica natural para especificar la semántica del lenguaje,[39] pero no sólo eso, sino que también sirve para especificar la semántica de la máquina y la compilación.[40]

Por último, señalar que debido a que nuestro lenguaje de ejemplo es simple no se hace uso de entornos, pero bien se puede hacer uso de ellos según se requiera con base en el lenguaje fuente.[41]

[39] Aquí por simplicidad sólo mostramos la semántica dinámica, pero Khan en [Kah87] muestra también las reglas para la semántica estática.

[40] Cabe observar que esta visión unificadora resulta natural para nuestros propósitos.

[41] En efecto, para el lenguaje que consideraremos en nuestro compilador principal se hace uso de ellos.

CAPÍTULO 3

Coq

3.1. Introducción

Coq es un asistente de pruebas basado en la tradición de LCF. El sistema formal en el que está basado es el cálculo de construcciones inductivas. La lógica proposicional intuicionista[1] (el fragmento implicacional de ésta) tiene una correspondencia con el cálculo lambda simplemente tipificado vía el isomorfismo de Curry-Howard. Debido a este isomorfismo por una parte las proposiciones de la lógica corresponden a los tipos del cálculo lambda y por otra las demostraciones de ésta corresponden a los términos de éste. Si extendemos el cálculo lambda con tipos suma y producto entonces esta correspondencia abarca la lógica proposicional intuicionista completa (es decir, con disyunción y conjunción). En ese mismo sentido si se extiende el cálculo lambda simplemente tipificado con tipos dependientes y otras características (incluidas definiciones inductivas) se obtiene el cálculo de construcciones inductivas. Con base en lo anterior es claro que este cálculo se puede utilizar como sistema de demostración. Por otra parte el cálculo[2] simplemente tipificado cumple con la propiedad de terminación, es decir, con el teorema de normalización fuerte 2.2.2, lo que intuitivamente significa que si este cálculo se utiliza como lenguaje de programación entonces toda evaluación posible de cualquier término de éste finaliza. El cálculo de construcciones inductivas que es una extensión del cálculo lambda simplemente tipificado también cumple con la propiedad de terminación, es decir, también cumple con el teorema de normalización fuerte.

Tomando en cuenta lo anterior a continuación daremos una presentación de Coq de la siguiente manera:

[1] Como vimos en la sección 2.2.2.
[2] Como estudiamos en la sección 2.2.2.

1. Primero mostraremos el uso de Coq como lenguaje de programación y estudiaremos cómo funciona el mecanismo de extracción de contenido computacional en este caso.

2. Luego presentaremos el uso de Coq como sistema de demostración y el funcionamiento del mecanismo de extracción de contenido computacional para este caso.

3. Posteriormente estudiaremos definiciones inductivas, el uso que tienen cuando Coq se utiliza como lenguaje de programación y el uso que tienen cuando Coq se utiliza como sistema de demostración.

4. Después ilustraremos con un pequeño ejemplo por un lado en qué consiste la verificación de un programa y cómo funciona el mecanismo de extracción en este caso. Y luego, por otro, mostraremos con el mismo ejemplo en qué consiste la certificación de un programa, veremos que en este caso es necesario hacer uso de un tipo dependiente, posteriormente presentaremos el funcionamiento del mecanismo de extracción en este contexto.

5. Una vez presentado lo anterior mostraremos cómo realizar un compilador correcto verificado de un pequeño lenguaje de expresiones aritméticas y en el camino nos encontraremos con el mecanismo de recursión estructural basada en sintaxis con el que Coq cuenta.

   Este punto en particular lo desarrollaremos para contar con todos los conocimientos necesarios para realizar un compilador correcto verificado en Coq. Daremos por sentados estos conocimientos cuando realicemos la verificación de la corrección de nuestro compilador principal en el capítulo 5.

De forma simplificada cuando utilizamos Coq lo que está sucediendo es que estamos escribiendo términos del cálculo de construcciones inductivas. En el cálculo de construcciones inductivas todo término tiene un tipo. En este cálculo los tipos también son términos por tanto un tipo también cuenta con un tipo.

Por otro lado el contenido computacional se refiere a que si se tiene un término t, intuitivamente t represente un término en el que se puedan realizar cálculos, de ser así se dice que t tiene contenido computacional, en otro caso se dice que t no tiene contenido computacional. El mecanismo de extracción de contenido computacional se refiere a que se si tiene un término t se pueda obtener el contenido computacional con el que cuenta éste, nótese que no necesariamente todo t debe tener contenido computacional, puede ser que únicamente un subtérmino t_c de t lo tenga (mientras que el subtérmino restante que conforma t no lo tenga), en tal caso el mecanismo de extracción obtendrá solamente aquel subtérmino t_c de t que tiene contenido computacional (mientras que el restante simplemente lo descartará). En la práctica en Coq el contenido computacional se refiere a que dado un desarrollo (un término) sea susceptible a partir de éste obtener un programa escrito en un lenguaje de programación funcional convencional de propósito general como

OCaml. Por tanto bajo esta perspectiva al mecanismo de extracción de contenido computacional también se le puede llamar mecanismo de extracción de programas. Entonces el mecanismo de extracción de programas se encarga de obtener programas a partir de los desarrollos en Coq que sean susceptibles de extracción, es decir, de aquellos que cuenten con contenido computacional. Coq cuenta con soporte de extracción de programas hacia los lenguajes OCaml, Haskell y Scheme, por omisión el mecanismo de extracción genera programas en OCaml.

La idea general es que si se tiene un término t_c de tipo φ o sea $t_c : \varphi$ y t_c tiene contenido computacional entonces el tipo de φ debe ser Set. Por otra parte si se tiene un término t_n de tipo ψ o sea $t_n : \psi$ y t_n no tiene contenido computacional entonces el tipo de ψ debe ser $Prop$. Podemos decir entonces que la distinción entre Set y $Prop$ obedece al mecanismo de extracción pues sirve para distinguir aquellos términos que tienen contenido computacional de los que no lo tienen.

Entonces se tienen los siguientes escenarios posibles dependiendo de cómo se utiliza Coq:

1. Como lenguaje de programación. En este caso si se tiene un término t_c de tipo φ o sea $t_c : \varphi$ entonces el tipo de φ será Set. Aquí podemos pensar en el tipo φ como un tipo de los que se utilizan usualmente en los lenguajes de programación.

2. Como sistema de demostración. Para este caso si se tiene un término t_n de tipo ψ o sea $t_n : \psi$ entonces el tipo de ψ será $Prop$. Es de utilidad aquí recordar el isomorfismo de Curry-Howard como lo estudiamos en la sección 2.2.2. Dado este isomorfismo las proposiciones de la lógica corresponden a los tipos, por tanto en este caso ψ representaría una proposición. Por otro lado las demostraciones corresponden a los términos, por lo que en este caso t_n sería un término que representa una demostración (de la proposición representada por ψ). Por lo que es claro que t_n no tiene contenido computacional (una demostración estrictamente de una proposición no contiene cálculos por realizar) y de esta manera toma sentido que el tipo de ψ sea $Prop$.

3. Como sistema de demostración e implícitamente como lenguaje de programación. En este caso es necesario hacer uso de un tipo dependiente Σ. Tomando como base el isomorfismo de Curry-Howard como lo estudiamos en la sección 2.2.2 a una proposición de la forma $\exists x \in X.P(x)$ le corresponde el tipo dependiente $\Sigma x : X.P(x)$. Entonces si se tiene un término t de tipo $\Sigma x : X.P(x)$ o sea $t : \Sigma x : X.P(x)$, intuitivamente un subtérmino $t_x : X$ de t usualmente representa un algoritmo para calcular un objeto x de tipo X y el subtérmino $t_p : P$ restante que conforma t representa una demostración de que el objeto x cumple con la proposición P. Por tanto el tipo de X es Set mientras que el tipo de P es $Prop$. Nótese como t sí tiene contenido computacional que es aquel correspondiente a su subtérmino t_x por tanto el tipo de $\Sigma x : X.P(x)$ es Set. Cuando se utilice el mecanismo de extracción de programas sobre t se extraerá el término t_x, es decir, el programa que calcula el objeto x, en cuanto a t_p este término es irrelevante y simplemente se descartará ya que no tiene

contenido computacional pues corresponde a la demostración de que x cumple con la proposición P. Podemos decir que t_x es un programa que calcula el objeto x y que t_p es un certificado de que el objeto x cumple con la proposición P por tanto se dice que el programa t_x que se obtiene al utilizar el mecanismo de extracción es un programa certificado.

Por otro lado dijimos que en el cálculo de construcciones inductivas un tipo también cuenta con un tipo. Por lo que nos podemos preguntar por el tipo de Set y $Prop$ respectivamente. El tipo de Set es $Type$ a su vez nos podemos preguntar por el tipo de $Type$. Para poder responder esta pregunta primero debemos decir que Coq cuenta con una jerarquía de universos infinita denotada por tipos de la forma $Type_i$ donde i se refiere al nivel del universo, así se tiene que el tipo de $Type_0$ es el tipo $Type_1$, el tipo de $Type_1$ es $Type_2$ y así sucesivamente, en general el tipo de $Type_i$ es el tipo $Type_{i+1}$. Aunque esta jerarquía existe, Coq la maneja internamente y no está disponible para su manipulación por el usuario, así que cualquier tipo $Type_i$ Coq simplemente lo presentará como $Type$. Vale la pena aclarar que en Coq Set es un sinónimo de $Type_0$ así en realidad el tipo de Set es $Type_1$. Por su parte el tipo de $Prop$ también es $Type_1$.

Comenzaremos ahora nuestra presentación de Coq propiamente dicha mostrando el funcionamiento de éste como lenguaje de programación.

3.2. Coq como lenguaje de programación

Primero hemos de decir que Coq visto como aplicación se inicia con el programa `coqtop`. A lo largo de este capítulo iremos mostrando las diferentes características con las que Coq cuenta a través de pequeños ejemplos y al mismo tiempo iremos mostrando la salida que Coq nos ofrezca como respuesta a éstos. Así cuando ejecutamos `coqtop` se muestra lo siguiente:

```
Welcome to Coq 8.4pl5 (February 2015)

Coq <
```

Cuando utilizamos Coq como lenguaje de programación podemos pensar en este programa como si se tratase de un intérprete interactivo de un lenguaje funcional. Tomando esto en cuenta, de forma simple en general lo que podemos hacer es definir funciones y realizar cálculos sobre éstas.

Comencemos viendo cómo escribir una función (no recursiva). Para este fin se utiliza la sintaxis `Definition` como se ilustra a continuación:

```
Coq < Definition double (n:nat) := n+n.
double is defined

Coq <
```

Aquí hemos escrito la función `double` que calcula el doble de un número natural.

Por otro lado para poder realizar el cálculo de una función se cuenta con el comando `Compute`. A continuación se ilustra su funcionamiento.

```
Coq < Compute (double 5).
     = 10
     : nat

Coq <
```

Aquí se ha calculado la función `double` con el argumento `5` y podemos ver que `Compute` devuelve el resultado esperado `10` junto con su tipo que es `nat`.

Ahora veamos cómo escribir una función recursiva. Para tal objetivo se cuenta con la sintaxis `Fixpoint` que se utiliza de la siguiente manera:

```
Coq < Fixpoint fact (n:nat) :=
Coq < match n with
Coq < | O => 1
Coq < | S k => (S k) * fact k
Coq < end.
fact is recursively defined (decreasing on 1st argument)

Coq <
```

Aquí hemos definido la función `fact` que calcula el factorial de un número natural. Nótese que `match` permite realizar *pattern matching* sobre los parámetros de una función y de esta manera poder definir la función por casos. Ahora utilizamos `Compute` para calcular el factorial de `5`

```
Coq < Compute (fact 5).
     = 120
     : nat

Coq <
```

y como se esperaba obtenemos `120` como resultado.

Por otra parte es de utilidad conocer el tipo de un término en Coq, el cual se puede conocer a través del comando `Check` como ejemplificamos a continuación.

```
Coq < Check 5.
5
     : nat

Coq < Check double.
double
     : nat -> nat

Coq < Check fact.
fact
     : nat -> nat

Coq <
```

Con estos ejemplos simples hemos mostrado a grandes rasgos cómo Coq se puede utilizar como un intérprete interactivo de un lenguaje funcional.

Veamos ahora cómo funciona el mecanismo de extracción para este caso.

3.2.1. Mecanismo de extracción

Tomemos como punto de partida la función `double`. Como vimos la función `double` tiene tipo `nat -> nat`, con base en nuestro razonamiento previo de la sección 3.1 nos preguntamos por el tipo de `nat -> nat`.

```
Check (nat -> nat).
nat -> nat
     : Set
```

y vemos que como esperábamos es de tipo `Set`. Entonces podemos hacer uso del mecanismo de extracción para obtener el contenido computacional de la función `double`. Coq cuenta con el comando `Extraction` para utilizar su mecanismo de extracción, el cual como dijimos si no se indica explícitamente otro lenguaje, por omisión Coq generará un programa en el lenguaje OCaml. A continuación ilustramos el uso de `Extraction`.

```
Coq < Extraction double.
(** val double : nat -> nat **)
```

```
let double n =
  plus n n

Coq <
```

y aquí podemos observar claramente cómo hemos obtenido un programa en OCaml correspondiente al contenido computacional extraído de la función `double` que escribimos en Coq. Podemos notar que Coq también escribe como comentario el tipo del término del que extrajo el contenido computacional, en este caso el tipo de la función `double` que es `nat -> nat`.

En el caso de la función `fact` ya vimos que cuenta también con el tipo `nat -> nat` y tenemos el mismo razonamiento que para la función `double` y por tanto podemos extraer su contenido computacional de la siguiente manera:

```
Coq < Extraction fact.
(** val fact : nat -> nat **)

let rec fact = function
| O -> S O
| S k -> mult (S k) (fact k)
```

y obtenemos como esperábamos una versión de la función `fact` escrita en OCaml.

Con esto damos por terminada nuestra sección de uso de Coq como lenguaje de programación y damos paso ahora a estudiar Coq como sistema de demostración.

3.3. Coq como sistema de demostración

Estudiaremos aquí cómo utilizar Coq como sistema de demostración. Siendo más precisos como vimos en la sección 2.2.2 el cálculo de construcciones inductivas se puede utilizar como sistema de demostración. Coq desde el punto de vista de aplicación cuando el objetivo es utilizar el cálculo de construcciones inductivas como sistema de demostración sirve como un asistente de pruebas.[3] En este sentido Coq es un programa que permite interactivamente al usuario expresar una proposición (lema, teorema, corolario) y luego demanda del usuario la prueba de esta proposición. Esta prueba se hace de manera interactiva, donde, el usuario va indicando cada uno de los pasos a seguir hasta (en su caso) llegar a la conclusión de la misma. Los pasos a seguir se indican mediante las tácticas que ofrece Coq, existen tácticas primitivas y tácticas compuestas (a partir de las primitivas u

[3] Y dado que este es el uso principal que se le da, usualmente se dice que Coq es un asistente de pruebas.

otras compuestas) a las que también se les llama *tacticals*. Para cada una de las reglas del cálculo de construcciones inductivas debe haber al menos una táctica que la implemente, esto es, al final lo que hace cada táctica es aplicar una secuencia de reglas del cálculo de construcciones inductivas; esto en la práctica es de gran ayuda, porque sería muy precario y tedioso aplicar en un paso una única regla del cálculo de construcciones inductivas y así las tácticas ofrecen al usuario un mayor nivel de abstracción. Las tácticas originalmente se escribían en OCaml, después se desarrolló un lenguaje específico para escribir tácticas de Coq, este lenguaje se llama $\mathcal{L}_{tac}$ [Del01].

Coq es un asistente de pruebas dirigido por objetivos (*goal directed*). Como vimos podemos iniciar Coq ejecutando el programa `coqtop` al cual al iniciar nos muestra la siguiente salida:

```
 Welcome to Coq 8.4pl5 (February 2015)

Coq <
```

en este momento podemos enunciar nuestro objetivo (proposición), lo cual se realiza de la siguiente manera:

```
Welcome to Coq 8.4pl5 (February 2015)

Coq < Goal forall p q:Prop, (p -> q) -> p -> q.
1 subgoal

  ============================
   forall p q : Prop, (p -> q) -> p -> q

Unnamed_thm <
```

Obsérvese cómo hemos enunciado el teorema (objetivo) $\forall p\ q, (p \to q) \to p \to q$ y luego Coq nos indica que tenemos un (sub)-objetivo por demostrar. En Coq $\forall$ se escribe: `forall` y $\exists$ se escribe: `exists`; ahora nos encontramos dentro del contexto de una prueba, damos nuestros primeros pasos de la siguiente manera:

```
Unnamed_thm < intro.
1 subgoal

  p : Prop
  ============================
   forall q : Prop, (p -> q) -> p -> q
```

```
Unnamed_thm < intro.
1 subgoal

  p : Prop
  q : Prop
  ============================
   (p -> q) -> p -> q

Unnamed_thm < intro H1.
1 subgoal

  p : Prop
  q : Prop
  H1 : p -> q
  ============================
   p -> q

Unnamed_thm < intro H2.
1 subgoal

  p : Prop
  q : Prop
  H1 : p -> q
  H2 : p
  ============================
   q

Unnamed_thm <
```

Aquí hemos utilizado la táctica `intro`[4] cuatro veces, esta táctica corresponde a la regla de introducción de la implicación "$\rightarrow$" en un sistema de deducción natural (en la tabla de la figura 3.1 se muestran las tácticas correspondientes a las reglas de deducción natural de los conectivos y cuantificadores lógicos usuales).

Arriba de la línea `==========` se muestran las hipótesis del objetivo actual. En Coq el razonamiento es hacia atrás (*backwards*), esto es, por ejemplo si queremos probar q y tenemos la hipótesis $p \rightarrow q$ para probar q nos basta con dar una prueba de p, así tenemos lo siguiente:

```
1 subgoal

  p : Prop
```

[4]La variante `intro name` le asigna el nombre `name` a la hipótesis, si se utiliza únicamente `intro` entonces el sistema le asigna de forma automática un nombre a la hipótesis.

	Introducción	Eliminación
$\bot$		`exfalso`
$\neg$	`intro` H	`destruct` H
$\rightarrow$	`intro` H	`apply` H
$\forall$	`intro` x	`apply` H
$\wedge$	`split`	`destruct` H `as` (x, y)
$\vee$	`left`, `right`	`destruct` H `as` $[x\|y]$
$\exists$	`exists` t	`destruct` H `as` (x, Hx)

Figura 3.1: Tácticas correspondientes a las reglas de deducción natural en Coq.

```
  q : Prop
  H1 : p -> q
  H2 : p
  ============================
   q

Unnamed_thm < apply H1.
1 subgoal

  p : Prop
  q : Prop
  H1 : p -> q
  H2 : p
  ============================
   p

Unnamed_thm <
```

`apply` corresponde a la regla de eliminación de la implicación "$\rightarrow$". En este punto justo lo que queremos probar (es decir, `p`), lo tenemos como hipótesis (es decir, `H2 : p`), aquí tenemos al menos dos opciones: utilizar la táctica `exact H2` o utilizar la táctica `assumption`.

```
Unnamed_thm < exact H2.
No more subgoals.

Unnamed_thm < Restart.
1 subgoal

  ============================
   forall p q : Prop, (p -> q) -> p -> q

Unnamed_thm < intros.
```

```
1 subgoal

  p : Prop
  q : Prop
  H : p -> q
  H0 : p
  ============================
   q

Unnamed_thm < apply H.
1 subgoal

  p : Prop
  q : Prop
  H : p -> q
  H0 : p
  ============================
   p

Unnamed_thm < assumption.
No more subgoals.
```

como podemos notar con `Restart` se puede reiniciar la prueba (descartando la parte de la prueba que llevábamos).[5] En ambos casos terminamos la prueba y se indica que no hay más (sub)-objetivos por probar, con esto podemos notar que se puede utilizar más de una táctica con el mismo fin. Para finalizar la prueba debemos escribir `Save`.

```
Unnamed_thm < assumption.
No more subgoals.

Unnamed_thm < Save.
intros.
apply H.
assumption.

Unnamed_thm is defined

Coq <
```

Así nuestra prueba queda definida (en esta sesión) en Coq. Demostraremos ahora la conmutatividad de la conjunción "$\wedge$".

[5] Con `Abort` se descarta la prueba y el objetivo actual.

```
Coq < Lemma andconm: forall p q:Prop, p /\ q -> q /\ p.
1 subgoal

  ============================
   forall p q : Prop, p /\ q -> q /\ p

andconm < intros.
1 subgoal

  p : Prop
  q : Prop
  H : p /\ q
  ============================
   q /\ p

andconm < split.
2 subgoals

  p : Prop
  q : Prop
  H : p /\ q
  ============================
   q

subgoal 2 is:
 p

andconm < destruct H.
2 subgoals

  p : Prop
  q : Prop
  H : p
  H0 : q
  ============================
   q

subgoal 2 is:
 p

andconm < exact H0.
1 subgoal

  p : Prop
  q : Prop
  H : p /\ q
  ============================
   p
```

```
andconm < destruct H.
1 subgoal

  p : Prop
  q : Prop
  H : p
  H0 : q
  ============================
   p

andconm < trivial.
No more subgoals.

andconm < Qed.
intros.
split.
 destruct H.
 exact H0.

 destruct H.
 trivial.

andconm is defined

Coq <
```

Como podemos notar, aquí utilizamos `Lemma andconm:` en lugar de `Goal`, de esta forma podemos darle un nombre a nuestra proposición para poder utilizarla en otra prueba, por ejemplo utilizando `apply andconm`, también podemos en lugar de usar `Save` utilizar `Qed` para finalizar una prueba (que es lo que se prefiere actualmente) y siempre al inicio de una prueba utilizar `Proof`[6] sólo por convención y claridad ya que este comando no tiene ningún efecto sobre la prueba.

Por otro lado, también notamos cómo durante una prueba pueden surgir diferentes (sub)-objetivos, tal como pudimos darnos cuenta después de haber utilizado la táctica `split` (que corresponde a la introducción de la conjunción "$\wedge$" en deducción natural, como lo indica nuestra tabla de la figura 3.1).

Por otra parte, siempre se puede pedir a Coq mostrar un objetivo particular utilizando el comando: `Show n` donde, `n` es el número de objetivo. También se puede dejar de hacer la prueba del objetivo actual y enfocarse a hacer la prueba del objetivo que se desea en ese momento utilizando el comando `Focus n` donde, `n` es el número de objetivo del cual se desea realizar la prueba.

El lenguaje de especificación de Coq (del que hemos estado haciendo uso) se llama gallina.

[6]Sobre todo cuando se usa un archivo para guardar la prueba.

También existen tácticas destinadas a automatizar la prueba (o al menos parte de ella), por ejemplo, para nuestra primera proposición podemos utilizar lo siguiente:

```
Coq < Lemma mp: forall p q:Prop, (p -> q)-> p -> q.
1 subgoal

  ============================
   forall p q : Prop, (p -> q) -> p -> q

mp < tauto.
No more subgoals.

mp < Qed.
tauto.

mp is defined
```

Aquí la táctica `tauto` implementa un procedimiento de decisión para la lógica proposicional intuicionista.

Otros ejemplos (entre varios) de tácticas de automatización son: `ring` que resuelve ecuaciones de estructuras de anillo y `omega` que implementa un procedimiento de decisión para la aritmética de Presburger. Hay una táctica general de automatización `auto` que toma en cuenta una lista de proposiciones (*hint list*) predefinida (a la que el usuario puede agregar las proposiciones que desee), en esta táctica se implementa un procedimiento de resolución al estilo de Prolog.

3.3.1. Mecanismo de extracción

Para comprender el mecanismo de extracción cuando Coq se utiliza como asistente para realizar pruebas observemos primero que,[7] cuando enunciamos un lema, teorema o corolario, lo que estamos haciendo es enunciar un tipo que representa una proposición φ. Inmediatamente después Coq nos demanda realizar una demostración de la proposición φ, es decir, construir un término t que tenga tipo φ. Coq nos ayuda a construir este término t a través de tácticas (que corresponden a utilizar una regla del cálculo de construcciones inductivas o una secuencia de ellas). Nótese que es lo mismo que sucede en lógica al realizar una demostración, se van utilizando las reglas de inferencia y axiomas con las que cuenta el sistema de demostración que se esté utilizando al ir desarrollando la demostración de una proposición. Entonces en el caso en que concluyamos la demostración, t por supuesto tendrá tipo φ. Ilustremos este razonamiento con un ejemplo muy simple, con la proposición: $\forall p \to p$.

[7] Aquí estamos dando por concedido que sabemos lo que estudiamos en las secciones 2.2.2 y 3.1.

```
Coq < Lemma pimpp: forall p:Prop, p -> p.
1 subgoal

  ============================
   forall p : Prop, p -> p

pimpp < intro.
1 subgoal

  p : Prop
  ============================
   p -> p

pimpp < intro.
1 subgoal

  p : Prop
  H : p
  ============================
   p

pimpp < apply H.
No more subgoals.

pimpp < Qed.
intro.
intro.
apply H.

pimpp is defined
```

En este ejemplo el tipo φ sería $\forall p \to p$ como se puede observar en Coq.

```
Coq < Check pimpp.
pimpp
     : forall p : Prop, p -> p
```

Por otro lado, el término t sería aquel que construimos al realizar la demostración con ayuda de las tácticas `intro`, `intro` y `apply` correspondientes a utilizar respectivamente las reglas de introducción de $\forall$, introducción de $\to$ y eliminación de $\to$. Podemos observar este término de la siguiente manera:

```
Coq < Print pimpp.
pimpp = fun (p : Prop) (H : p) => H
     : forall p : Prop, p -> p
```

es decir, el nombre del lema, es el nombre que Coq le da al término t correspondiente a la demostración. Aquí también podemos ver claramente cómo el término t tiene el tipo φ.

Ahora nos preguntamos por el tipo de `forall p : Prop, p -> p` y tenemos:

```
Coq < Check (forall p : Prop, p -> p).
forall p : Prop, p -> p
     : Prop
```

y vemos que tiene tipo `Prop` como esperábamos, por lo tanto sabemos que t no cuenta con contenido computacional, lo cual podemos verificar utilizando el mecanismo de extracción de Coq sobre este término:

```
Coq < Extraction pimpp.
(** val pimpp : __ **)

let pimpp =
  __

Coq <
```

Aquí __ denota que no se cuenta con contenido computacional o de manera equivalente que el contenido computacional que se extrajo es vacío. Este es el caso para todas las demostraciones de proposiciones (términos t de tipo φ, donde φ tiene tipo $Prop$) en Coq. Con esto concluimos esta sección y nos disponemos ahora a presentar el uso de definiciones inductivas en Coq.

3.4. Definiciones inductivas

Originalmente Coq estaba basado en el cálculo de construcciones, posteriormente se vislumbró que sería mucho más claro y cómodo realizar desarrollos en Coq si éste contara con definiciones inductivas soportadas primitivamente. Esto motivó el desarrollo del cálculo de construcciones inductivas.

Una definición inductiva sirve para diferentes propósitos. Nosotros podemos pensarlas en que tienen un uso con base en como se utilice Coq, de la siguiente manera:

1. Como lenguaje de programación. En ese sentido podemos utilizar una definición inductiva para definir un tipo de datos recursivo como lo que se utilizan usualmente en un lenguaje de programación funcional. En tal caso la definición inductiva debe tener el tipo Set. Aquí si se usa el mecanismo de extracción sobre la definición inductiva se obtendrá un tipo recursivo escrito en un lenguaje de programación funcional convencional (hemos dicho ya que en Coq por omisión este lenguaje es OCaml).

2. Como sistema de demostración. En este sentido podemos utilizar una definición recursiva para definir proposiciones inductivas. En este caso el tipo de la definición inductiva debe ser $Prop$.[8] En cuanto al mecanismo de extracción, en este caso la definición inductiva como se espera no cuenta con contenido computacional.

3.4.1. Como lenguaje de programación

Veamos un ejemplo del primero de estos usos definiendo el tipo de datos recursivo correspondiente a los números naturales. En Coq este tipo de datos se define de la siguiente manera:

```
Inductive nat : Set :=
| O: nat
| S : nat -> nat.

nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

Al utilizar `Inductive` se da un nombre a la definición, en este caso `nat`; un tipo, en este caso `Set` y uno o varios constructores, en este caso `O` (que no recibe argumentos) y `S` que dado un `nat` construye otro (su sucesor), en la pruebas se puede hacer referencia a cada uno de los constructores utilizando la táctica `constructor n` donde, `n` indica el número de constructor, en este caso `constructor 1` haría referencia al constructor `O` y `cosntructor 2` al constructor `S`. En ciertos casos Coq puede determinar de manera automática a qué constructor se está haciendo referencia en tal caso se utiliza la táctica `constructor` simplemente.

Podemos verificar el tipo de `O` como se muestra a continuación.

[8] Siendo más precisos debe tener un tipo que exprese que cuando se construya un término utilizando los constructores de esa definición dicho término tenga el tipo `Prop`.

```
Coq < Check O.
O
     : nat
```

Y como se esperaba vemos que es `nat`.

Ahora verifiquemos el de `S O`

```
Coq < Check (S O).
S O
     : nat
```

y también es `nat` como esperábamos.

Nótese que también como resultado de utilizar la definición `Inductive` anterior se generó el teorema `nat_ind`.

```
Print nat_ind.
nat_ind =
fun P : nat -> Prop => nat_rect P
     : forall P : nat -> Prop,
       P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

El teorema `nat_ind` que Coq generó automáticamente corresponde al principio de inducción correspondiente a la definición inductiva que vimos, en este caso como definimos los números naturales, tal principio es el principio de inducción de los naturales.

El principio de inducción que Coq genera con base en una definición inductiva resulta muy importante y de gran utilidad ya que con base en él entre otras cosas se pueden realizar demostraciones por inducción sobre la entidad que se está definiendo.

Al realizar una prueba por inducción en lugar de utilizar tácticas básicas, existe una táctica `induction` que precisamente utiliza el principio de inducción que Coq genera de manera automática.

Como ejemplo probaremos por inducción que un natural es distinto de su sucesor.

```
Lemma difsuc: forall n:nat, n <> (S n).
1 subgoals, subgoal 1 (ID 15)

  ============================
   forall n : nat, n <> S n
Proof.
```

```
1 subgoals, subgoal 1 (ID 15)

  ============================
   forall n : nat, n <> S n
intros.
1 subgoals, subgoal 1 (ID 16)

  n : nat
  ============================
   n <> S n
induction n.
2 subgoals, subgoal 1 (ID 19)

  ============================
   O <> S O

subgoal 2 (ID 22) is:
 S n <> S (S n)
discriminate.

1 subgoals, subgoal 1 (ID 22)

  n : nat
  IHn : n <> S n
  ============================
   S n <> S (S n)

intuition.

1 subgoals, subgoal 1 (ID 32)

  n : nat
  IHn : n = S n -> False
  H : S n = S (S n)
  ============================
   False
apply IHn.

1 subgoals, subgoal 1 (ID 52)

  n : nat
  IHn : n = S n -> False
  H : S n = S (S n)
  ============================
   n = S n
inversion H.
1 subgoals, subgoal 1 (ID 69)

  n : nat
```

```
  IHn : n = S n -> False
  H : S n = S (S n)
  H1 : n = S n
  ============================
   S n = S (S n)
trivial.

No more subgoals.

(dependent evars:)
Qed.

difsuc is defined
```

Este ejemplo muestra (uno de los posibles) funcionamientos de las tácticas `induction`, `inversion` y `discriminate` que son tácticas que trabajan sobre definiciones inductivas.

Mecanismo de extracción

Veamos ahora lo que se obtiene al utilizar el mecanismo de extracción sobre la definición `nat`

```
Coq < Extraction nat.
type nat =
| O
| S of nat

Coq <
```

que es justamente lo que esperábamos obtener, el tipo `nat` recursivo en OCaml.

Por último vale la pena aclarar que el tipo `nat` está disponible en Coq y aquí lo hemos definido sólo para utilizarlo como ejemplo.

Veamos ahora el uso de una definición inductiva cuando Coq se utiliza como sistema de demostración.

3.4.2. Como sistema de demostración

Por otra parte, las definiciones inductivas sirven también para expresar proposiciones inductivas (que también pueden ser vistas como relaciones). Por ejemplo, podemos utilizar una de ellas para expresar la proposición ser par sobre números naturales.

```
Inductive even : nat -> Prop :=
| EvenO : even O
| EvenSS : forall n, even n -> even (S (S n)).
```

Nótese cómo en este caso el tipo de la definición es `nat -> Prop` ya que un constructor de esta definición toma un número natural y regresa una proposición que este natural cumple.

Verifiquemos el tipo de `even O` y de `even (S O)`

```
Coq < Check (even O).
even O
     : Prop

Coq < Check (even (S O)).
even (S O)
     : Prop

Coq <
```

También podemos utilizar una de estas definiciones para expresar la relación menor o igual sobre números naturales de la siguiente manera:

```
Inductive le : nat -> nat -> Prop :=
| le0: forall n:nat, le O n
| leS: forall n m:nat, le n m -> le (S n) (S m).
```

obsérvese que podemos expresar esta relación de manera abstracta de la siguiente manera:[9]

$$\frac{}{0 \leq n} \qquad \frac{n \leq m}{(Sn) \leq (Sm)}$$

Mecanismo de extracción

Veamos cómo funciona el mecanismo de extracción en este caso utilizándolo sobre la definición `even`

[9]También podemos escribir de forma similar el predicado ser par.

```
Coq < Extraction even.
(* even : logical inductive *)
(* with constructors : EvenO
EvenSS *)
```

Como podemos observar lo que se obtiene únicamente es un comentario en el que Coq indica que se trata de una proposición lógica inductiva y el nombre de los constructores que tiene, pero no se obtiene ningún programa, es decir, como esperábamos la definición no tiene contenido computacional.

Observemos ahora el caso de la definición `le`

```
Coq < Extraction le.
(* le : logical inductive *)
(* with constructors : le_n
le_S *)
```

nuevamente como esperábamos no se cuenta con contenido computacional.

Estudiemos ahora cómo realizar la verificación de un programa.

3.5. Verificación de un programa

Podemos decir que en general nuestro objetivo es contar con un programa que cumpla cierta especificación.

Un camino posible para realizarlo es a través de verificación.

La idea general de la verificación es utilizar Coq como lenguaje de programación para escribir un programa y luego utilizar Coq como sistema de demostración para probar que dicho programa cumple una propiedad.

Hemos visto que en Coq al decir programa nos estamos refiriendo a una función y al decir propiedad nos estamos refiriendo a una proposición. Por otra parte en Coq al decir especificación también nos referimos a una proposición.

Ilustraremos la verificación de un programa mediante un ejemplo muy sencillo.

Supongamos que deseamos contar con un programa que cumpla con la especificación $f(n) = 2 * n$.

Entonces lo primero que debemos hacer es escribir un programa que sea nuestro candidato a cumplir con la especificación. Con tal motivo escribimos la función `double` de la siguiente manera:

```
Coq < Definition double (n:nat) := n+n.
double is defined

Coq <
```

y ahora ya que hemos escrito nuestro programa, debemos mostrar que en efecto cumple con la especificación, lo cual hacemos en Coq mediante un teorema de la siguiente forma:

```
Coq < Theorem doublec: forall n:nat, double n = 2 * n.
1 subgoal

  ============================
   forall n : nat, double n = 2 * n

doublec < Proof.
1 subgoal

  ============================
   forall n : nat, double n = 2 * n

doublec < intros.
1 subgoal

  n : nat
  ============================
   double n = 2 * n

doublec < unfold double.
1 subgoal

  n : nat
  ============================
   n + n = 2 * n

doublec < omega.
No more subgoals.

doublec < Qed.
intros.
unfold double.
omega.

doublec is defined
```

Con esto contamos en este momento ya con la certeza que nuestra función `double` cumple con la especificación $f(n) = 2 * n$ (la cual Coq nos exige que seamos más precisos

y la escribamos como $\forall n, f(n) = 2 * n$). Nótese que aquí podemos calcular la función `double` para un argumento dentro de Coq mediante `Compute`.

```
Coq < Compute (double 5).
     = 10
     : nat

Coq <
```

Una vez que hemos verificado que nuestro programa cumple con la especificación deseada, por supuesto podemos utilizar el mecanismo de extracción sobre nuestro programa, en este caso la función `double` y obtener así un programa verificado.

```
Coq < Extraction double.
(** val double : nat -> nat **)

let double n =
  plus n n

Coq <
```

Como podemos observar hemos obtenido la función `double` escrita en OCaml lista para ser utilizada en la vida real con la certeza que cumple con la especificación que establecimos.

Al proceso que acabamos de ilustrar se le conoce como verificación de programas.

Nótese cómo en este caso no se tiene nada que extraer del teorema `doublec`. Este teorema como sabemos es una proposición y tiene tipo $Prop$, por tanto no tiene contenido computacional, como podemos constatar a continuación.

```
Coq < Extraction doublec.
(** val doublec : __ **)

let doublec =
  __

Coq <
```

Nosotros utilizaremos verificación para demostrar la corrección de nuestro compilador principal.

Veamos ahora en qué consiste la certificación de un programa.

3.6. Certificación de un programa

En esta ocasión partimos del mismo objetivo de la sección anterior, contar con un programa que cumpla con cierta especificación.

El otro camino posible para lograrlo es a través de la certificación.

Para poder realizar certificación en Coq es necesario hacer uso de un tipo dependiente Σ. Como lo discutimos en la sección 3.1 un tipo $\Sigma x : X.P(x)$ intuitivamente permite expresar un objeto x de tipo X que cumple una proposición P. Entonces podemos tomar el objeto x como la salida del programa y la proposición P como la especificación que debe cumplir dicha salida del programa.

En Coq el tipo $\Sigma x : X.P(x)$ se escribe como `{x:X | P(x)}`.

La idea general de la certificación consiste en utilizar Coq como sistema de demostración e implícitamente como lenguaje de programación.

Esto significa expresar en Coq mediante un tipo dependiente que la salida x de un programa cumple con cierta especificación P. Esto demandará una demostración en la que por una parte se debe construir implícitamente con ayuda de las tácticas de Coq el programa que calcula x y por otra realizar la demostración de que x cumple con la proposición P.

Ilustraremos la certificación de un programa con el mismo ejemplo de la sección anterior.

Entonces la especificación que debe cumplir la salida del programa k es $\forall n, f(n) = 2*n$ y esto lo expresamos en Coq mediante el tipo dependiente Σ de la siguiente manera:

```
Coq < Definition doublecc: forall n:nat, {k:nat | k = 2 * n}.
1 subgoal

  ============================
   forall n : nat, {k : nat | k = 2 * n}
```

Como podemos ver Coq demanda inmediatamente realizar una demostración, la cual podemos iniciar de la siguiente manera:

```
doublecc < Proof.
1 subgoal

  ============================
   forall n : nat, {k : nat | k = 2 * n}

doublecc < intros.
1 subgoal

  n : nat
```

```
  ============================
   {k : nat | k = 2 * n}
```

justo aquí debemos mostrar cómo construir la salida que cumple con la especificación, lo cuál lo hacemos de la siguiente forma:

```
doublecc < exists (n+n).
1 subgoal

  n : nat
  ============================
   n + n = 2 * n
```

ahora comienza la parte de la prueba, dedicada propiamente dicho a la demostración de que salida del programa cumple con la especificación, la cual realizamos de la siguiente manera:

```
doublecc < omega.
No more subgoals.

doublecc < Defined.
intros.
exists (n + n).
omega.

doublecc is defined

Coq <
```

Con esto concluimos nuestra demostración.

Veamos ahora qué pasa si intentamos utilizar `doublecc` para realizar un cálculo dentro de Coq.

```
Compute (doublecc 5).

  = exist (fun k : nat => k = 2 * 5) 10
         (Decidable.dec_not_not (10 = 10) (or_introl eq_refl)
            (fun x : 10 <> 10 =>
             match
               match Nat2Z.inj_add 5 5 in (_ = x0) return (x0 = 10%Z) with
```

```
              | eq_refl => eq_refl
              end in (_ = x0) return ((x0 = 10%Z -> False) -> False)
            with
            | eq_refl =>
                match
                  match
                    Nat2Z.inj_mul 2 5 in (_ = x0) return (x0 = 10%Z)
                  with
                  | eq_refl => eq_refl
                  end in (_ = x0) return ((10%Z = x0 -> False) -> False)
                with
                | eq_refl =>
...
```

y lo que sucede es que si bien sí calcula el resultado de nuestra función que en este caso es `10` también construye un término correspondiente a la demostración de que la salida de esta función cumple con la especificación. En comparación como vimos en la verificación únicamente se obtiene `10` (al calcular la función `double`) que para nuestros fines es lo que realmente nos interesa.

Por otra parte retomando el punto en que concluimos nuestra demostración notemos que en la primera parte de ésta hemos construido implícitamente el programa que calcula la salida mientras que en la segunda desarrollamos la demostración de que la salida cumple la especificación. A esta última demostración también se le conoce como certificado ya que certifica que la salida cumple con la especificación. Ahora si utilizamos el mecanismo de extracción con base en lo que estudiamos en la sección 3.1 esperamos obtener el programa en OCaml correspondiente al programa que construimos implícitamente en la primera parte.

```
Coq < Extraction doublecc.
(** val doublecc : nat -> nat **)

let doublecc n =
  plus n n

Coq <
```

y es justamente lo que se obtiene, entonces se dice que este programa es un programa certificado.

Verifiquemos el tipo de `forall n:nat, {k:nat | k = 2 * n}`

```
Coq < Check (forall n:nat, {k:nat | k = 2 * n}).
forall n : nat, {k : nat | k = 2 * n}
     : Set
```

y vemos que es Set como esperábamos, lo que indica que un término de este tipo (como el que construimos al realizar la demostración) sí tiene contenido computacional.

Ahora inspeccionemos el tipo Σ de Coq.

```
Coq < Print sig.
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> {x | P x}
```

Aquí podemos observar explícitamente cómo este tipo dependiente Σ está conformado por un objeto x de tipo A, donde A tiene tipo Set,[10] es decir, tiene contenido computacional y por una proposición $P(x)$ de tipo $Prop$, es decir, que no tiene contenido computacional. Por otro lado el tipo Σ tiene tipo Set lo que indica que sí tiene contenido computacional y sabemos que es aquel correspondiente al objeto x.

Con esto damos por concluido la certificación de nuestro programa.

Como pudimos observar a través de estos dos ejemplos contamos con dos vías diferentes de obtener un programa con la certeza que cumple cierta especificación a saber la verificación y la certificación.

Las principales diferencias entre éstas y el por qué hemos elegido utilizar verificación para realizar la corrección de nuestro compilador principal las hemos enunciado ya en la sección 1.3.

Estudiemos ahora cómo realizar un compilador correcto verificado para un lenguaje muy simple de ejemplo correspondiente a expresiones aritméticas.

3.7. Compilador correcto de expresiones aritméticas

En esta sección ilustraremos cómo realizar la verificación de la corrección de un compilador, a través de un ejemplo simple desarrollando un compilador correcto verificado para un lenguaje de expresiones aritméticas. Esto con la finalidad de estudiar y conocer todos los pasos necesarios para realizar la verificación de la corrección de un compilador en Coq. Conocimientos que daremos por sentados en el capítulo 5 cuando realicemos la verificación de la corrección de nuestro compilador principal.

[10] Recordemos que (como vimos en la sección 3.1) $Type_0$ es un sinónimo del tipo Set.

Lo primero es decir que consideraremos que ya se han realizado las etapas de análisis léxico, análisis sintáctico y análisis semántico;[11] por otro lado, consideraremos únicamente expresiones aritméticas sobre naturales y nos enfocaremos en el caso de la adición.[12]

Entonces, lo que queremos es un compilador que respete la semántica (dinámica) del lenguaje, esto es, si se tiene un programa p podemos conocer su comportamiento con base en la semántica del lenguaje, por otra parte, si lo compilamos y obtenemos un código c podemos conocer el comportamiento de c con base en la semántica de la máquina virtual, entonces, queremos que estos dos comportamientos sean en algún sentido equivalentes.

Utilizaremos la semántica natural, entonces con base en lo que estudiamos en la sección 2.3.2 nuestra primera tarea es presentar por una parte la sintaxis abstracta del lenguaje y por otra los valores finales.

Nuestra sintaxis abstracta es:

$$\begin{array}{rcl} e & ::= & \text{Const } n \\ & | & \text{Plus } e\ e \end{array}$$

y los valores son:

$$\begin{array}{rcl} v & ::= & \text{Num } n \end{array}$$

Ahora debemos expresarla en Coq, y nos fijamos que puede ser como un tipo recursivo, o sea, una definición inductiva.

```
Inductive exp: Set :=
| Const: nat -> exp
| Plus: exp -> exp -> exp.

Inductive val: Set :=
| Num: nat -> val.
```

Ahora podemos escribir la expresión $5 + 2$ en sintaxis abstracta de nuestro lenguaje y verificar que tiene el tipo correcto, como se muestra a continuación:

```
Check(Plus (Const 2) (Const 5)).
Plus (Const 2) (Const 5)
     : exp
```

[11] Más adelante en las conclusiones (capítulo 6) mencionaremos brevemente cuál es el estado actual de la verificación de la corrección de esas etapas dentro de un compilador.

[12] El desarrollo para los otros operadores aritméticos es análogo.

Nótese cómo nuestro término `Plus (Const 2) (Const 5)` de tipo `exp` corresponde exactamente a un nodo `Plus` de un AST escrito en notación prefija, por esto en lugar del nombre `exp` también pudimos haberle puesto `ast` porque tanto en Coq como en los lenguajes funcionales, un AST se puede expresar naturalmente como un tipo recursivo.

Debemos ahora presentar la semántica del lenguaje, lo que hacemos a continuación.

$$\frac{}{\text{Const } n \Rightarrow \text{Num } n} \qquad \frac{e_1 \Rightarrow \text{Num } n_1 \qquad e_2 \Rightarrow \text{Num } n_2}{\text{Plus } e_1\ e_2 \Rightarrow \text{Num } n_1 + n_2}$$

veamos ahora cómo podemos expresar la semántica natural en Coq, aquí nos acordamos que las reglas y axiomas de la semántica natural se expresan como una relación y vimos que podemos escribir una relación como una definición inductiva en Coq, así obtenemos lo siguiente:

```
Inductive expBSS : exp -> val -> Prop :=
| BSConst: forall n:nat, expBSS (Const n) (Num n)
| BSPlus: forall e1 e2:exp, forall n1 n2:nat,
            expBSS e1 (Num n1) ->
            expBSS e2 (Num n2) ->
            expBSS (Plus e1 e2) (Num (n1 + n2)).
```

aquí tenemos un constructor por cada regla: `BSConst` corresponde al axioma constante y `BSPlus` corresponde a la regla de la adición, por otra lado aquí se nota explícitamente cómo en la semántica natural siempre que se evalúa una expresión se devuelve un valor (final).[13]

Notemos aquí que podemos ver la semántica de lenguaje representada en Coq como una proposición inductiva como una especificación. Y entonces siguiendo el enfoque de la verificación podemos construir un programa y luego verificar que éste cumple con esta especificación y en este caso obtendríamos un intérprete. Desde luego realizar un intérprete no es nuestro objetivo principal y no es necesario realizarlo estrictamente hablando para nuestros propósitos, sin embargo aquí nos hemos encontrado con esa posibilidad y nos parece conveniente con fines de experimentación.

Damos pues primero nuestro programa, es decir, nuestro intérprete

```
Fixpoint eval (e:exp) :=
match e with
| Const n => Num n
| Plus e1 e2 => match eval e1 with
                      | Num n1 => match eval e2 with
                                          | Num n2 => Num (n1+n2)
```

[13]Siempre que existan reglas para la expresión y sub-expresiones que la componen.

```
                                        end
                    end
end.
```

y luego verificamos que cumple con la especificación, es decir, con la semántica del lenguaje.[14]

```
Lemma evalc:
forall e:exp, forall v:val,
expBSS e v ->
eval e = v.
Proof.
intros.
dependent induction e.
(*Caso Const*)
inversion H.
simpl.
trivial.

(*Caso Plus*)
inversion H.
simpl.
destruct(eval e1) eqn:?.
destruct(eval e2) eqn:?.
assert((Num n) = (Num n1)).
apply IHe1.
trivial.
assert((Num n0) = (Num n2)).
apply IHe2.
trivial.
congruence.
Qed.
```

Nótese que aquí podemos utilizar nuestro intérprete dentro de Coq de la siguiente manera:

```
Compute (eval (Plus (Const 5) (Const 2))).

     = Num 7
     : val
```

[14]Esta vez únicamente mostramos las tácticas utilizadas en la demostración y no la salida que producen cada una de ellas por cuestiones de espacio.

Ahora podemos utilizar el mecanismo de extracción

```
Extraction eval.

(** val eval : exp -> val0 **)

let rec eval = function
| Const n -> n
| Plus (e1, e2) -> plus (eval e1) (eval e2)
```

y obtenemos así un intérprete verificado (escrito en el lenguaje en OCaml).

También podemos utilizar certificación para obtener nuestro intérprete, lo cual hacemos de la siguiente manera.

Primero enunciamos el tipo dependiente correspondiente y realizamos su demostración

```
Definition expBSSC : forall e, {v:val | expBSS  e v}.
Proof.
intros.
induction e.
intros.
exists (Num n).
constructor.

inversion IHe1.
inversion IHe2.

destruct x.
destruct x0.

exists (Num (n+n0)).
constructor.
trivial.
trivial.
Defined.
```

ahora podemos tratar de utilizar `expBSSC` para realizar cálculos sobre nuestro intérprete

```
Compute (expBSSC (Plus (Const 5) (Const 2))).
     = exist (fun v : val => expBSS (Plus (Const 5) (Const 2)) v)
         (Num 7) (BSPlus (Const 5) (Const 2) 5 2 (BSConst 5) (BSConst 2))
     : {v : val | expBSS (Plus (Const 5) (Const 2)) v}
```

y vemos que aunque si calculó el resultado esperado `Num 7`, Coq también muestra el término correspondiente a la demostración de que se cumple la especificación.

Ahora podemos utilizar el mecanismo de extracción de programas

```
Extraction expBSSC.

(** val expBSSC : exp -> val0 **)

let rec expBSSC = function
| Const n -> n
| Plus (e0, e1) -> plus (expBSSC e0) (expBSSC e1)
```

y obtenemos de esta manera un intérprete certificado.

Retomando nuestro objetivo principal de desarrollar un compilador correcto verificado debemos ahora definir la máquina que utilizaremos. Para lo cual primero presentamos por un lado sus instrucciones y por otro los valores que puede almacenar su pila.

Las instrucciones son:

$$\begin{array}{lll} \mathit{inst} & ::= & \text{IConst } n \\ & | & \text{IAdd} \end{array}$$

mientras que los valores son:

$$v_p \quad ::= \quad \text{SVNum } n \qquad\qquad n \in \mathbb{N}$$

Notemos que hemos elegido representar el código de la máquina como una secuencia de instrucciones.[15]

El desarrollo correspondiente en Coq es:

```
Inductive instr: Set :=
| IConst: nat -> instr
| IAdd: instr.

Definition code := list instr.

Inductive sval: Set :=
| SVNum: nat -> sval.

Definition stack := list sval.
```

[15]Es decir, sin una instrucción ISeq explícita.

Como se puede observar el código de la máquina se implementa como una lista de instrucciones y la pila como una lista de valores de pila.

Toca el turno de especificar la semántica de la máquina, tradicionalmente para especificar la semántica de una máquina se utiliza la semántica de paso pequeño (o una versión informal de ésta), por otra parte, nos parece natural utilizar la semántica de paso grande con base en la visión unificadora de Khan, como ambos enfoques nos parecen lo suficientemente relevantes utilizaremos ambos y esto nos lleva a probar su equivalencia.[16]

Comencemos por la semántica de paso grande:

$$\frac{[\text{SVNum } n] \cdot s \vdash c \Rightarrow m_f}{s \vdash [\text{IConst } n] \cdot c \Rightarrow m_f} \qquad \frac{[\text{SVNum } (n_1 + n_2)] \cdot s \vdash c \Rightarrow m_f}{[\text{SVNum } n_2,\ \text{SVNum } n_1] \vdash \text{IAdd } \cdot c \Rightarrow m_f}$$

donde m es una configuración de la forma (c, s), aquí c es un código de la máquina y s una pila; m_f hace alusión a una configuración final (c_f, s_f).

El desarrollo en Coq correspondiente a la semántica natural de la máquina es el siguiente:

```
Definition mconf := (code * stack)%type.
Inductive BSMsem : mconf -> mconf -> Prop :=
| BSICons: forall n:nat, forall c:code, forall s:stack,
           forall mf:mconf,
             BSMsem (c, SVNum n::s) mf ->
             BSMsem (IConst n::c,s) mf

| BSIAdd: forall n1 n2:nat, forall c:code, forall s:stack,
          forall mf:mconf,
            BSMsem(c,SVNum (n1+n2)::s) mf ->
            BSMsem(IAdd::c,SVNum n2::SVNum n1::s) mf.
```

Ahora notemos cómo aquí (al igual que lo hicimos con la semántica del lenguaje anteriormente) podemos considerar la semántica natural de la máquina representada en Coq como la proposición inductiva `BSMsem` como una especificación. De esta manera podemos seguir el camino de la verificación, esto es, escribir un programa y luego demostrar que éste cumple con dicha especificación y obtener así un intérprete verificado de la máquina. Obsérvese que en este caso es de gran utilidad contar con un intérprete verificado de la máquina ya que en el supuesto que contemos con un compilador verificado (que estamos por desarrollar) esto no implicaría que tuviésemos un intérprete de la máquina donde se pudiera ejecutar el código de máquina de un programa generado por nuestro compilador. Así que es de gran ayuda que hayamos encontrado en el camino una forma de obtener este intérprete.

[16] Aquí podemos notar cómo contar con más de un formalismo nos ayuda en el sentido en que podemos compararlos, observar sus similitudes y diferencias, sus fortalezas y debilidades y qué es lo que cada uno nos permite observar acerca del comportamiento de un programa.

A continuación escribimos (el primer intento de) este intérprete en Coq.

```
Fixpoint mexecrec(m:mconf) : mconf :=
match m with
| (nil, s) => (nil,s)
| (IConst n::c,s) => mexecrec (c, SVNum n::s)
| (IAdd::c, SVNum n2::SVNum n1::s) =>
  mexecrec(c,SVNum (n1+n2)::s)
end.

Error: Non exhaustive pattern-matching: no clause found for pattern
(IAdd :: _, nil)
```

Sin embargo como podemos observar Coq nos muestra un mensaje indicando que ha sucedido un error. Analicemos la causa de este error. Recordando lo que estudiamos y mencionamos en la sección 2.2.2 el cálculo de construcciones inductivas cumple con el teorema de normalización fuerte, lo cual intuitivamente significa que toda evaluación de cualquier programa necesariamente debe terminar. En la práctica podemos pensar en Coq como una implementación de este cálculo. Entonces desde un enfoque pragmático Coq debe contar con algún mecanismo que garantice que todo programa (función) que se escriba en él necesariamente termine, para que de este modo Coq sea correcto respecto de esta propiedad del cálculo de construcciones inductivas.

Por otro lado, desde luego que para que todo cálculo de una función termine debe estar definida para todo valor de entrada posible (pues de otra forma para un valor de entrada no definido no se sabría cómo calcular la función y por supuesto su cálculo no terminaría), es decir, debe ser una función total. Entonces en nuestro ejemplo debemos definir nuestra función para todo caso de entrada posible.

Nos podemos preguntar entonces por qué no nos sucedió esto al definir la semántica y la respuesta: es porque cuando definimos la semántica utilizamos Coq como sistema de demostración, es decir, enunciamos la semántica como una proposición inductiva mediante una definición inductiva, esto es más evidente al notar la presencia del tipo `Prop` en nuestra definición. Por tanto, como sabemos esta definición no tiene contenido computacional y por ello no es susceptible de que se realicen cálculos sobre ella, por eso en particular no es necesario definirla sobre todos los casos posibles de entrada.

Volviendo a nuestra función, una instrucción `IAdd` sólo está definida para el caso en que haya dos naturales en el tope de la pila para ser sumados y esto debe ser siempre así si el código que se está ejecutando es el resultado del análisis léxico, sintáctico, semántico y generador de código de nuestro compilador, pero Coq no sabe que el único código que vamos a ejecutar es la salida de nuestro compilador.[17] Por tanto debemos dar un valor como resultado a este caso sin sentido (es decir, el caso en que no haya dos naturales en

[17]Podríamos especificárselo utilizando un tipo dependiente, pero como hemos dadas por concedidas las tres primeras etapas del compilador, se deja como experimento para trabajo futuro.

el tope de la pila cuando se va realizar una suma), en este tipo de escenarios es útil el tipo `option`:

```
Print option.
Inductive option (A : Type) : Type :=
Some : A -> option A
| None : option A
```

El tipo `option` tiene dos constructores: `Some` para regresar los valores que tienen sentido y `None` para los que no lo tienen, así nuestra función quedaría:

```
Fixpoint mexecrec(m:mconf) : option mconf :=
match m with
| (nil, s) => Some (nil,s)
| (IConst n::c,s) => mexecrec (c, SVNum n::s)
| (IAdd::c, SVNum n2::SVNum n1::s) =>
  mexecrec(c,SVNum (n1+n2)::s)
| _ => None
end.

Error:
Recursive definition of mexecrec is ill-formed.
Recursive call to mexecrec has principal argument equal to
"(c, SVNum n :: s)" instead of a subterm of "m".
```

Como podemos observar una vez más hemos recibido un mensaje indicando un error. La causa nuevamente es que todo programa en Coq debe terminar. Mencionamos arriba que en la práctica Coq debe contar con un mecanismo que permita asegurar esta propiedad y el mecanismo que se utiliza por omisión para garantizar que todo cálculo sobre una función (recursiva) termina, es recursión estructural basada en sintaxis (presentada por Giménez en [Gim96]). La idea general sobre la que descansa este mecanismo es que por un lado se cuente con un orden estructural sobre el tipo del argumento sobre el que se hace la recursión y por el otro que al calcular la función se realicen llamadas recursivas sobre elementos menores que el actual (con base en dicho orden) y de esta manera garantizar que la función siempre terminará. Para tener certeza que se hará (al evaluar la función) una llamada sobre un elemento menor, estáticamente (al definir la función) se verifica que la llamada se haga sobre una subestructura del argumento (que en Coq equivale a un subtérmino del término correspondiente al argumento de entrada), la forma de revisar que en efecto se trata de una subestructura es mediante la sintaxis.

Aunque en la práctica este mecanismo funciona bien para una gran parte de los casos, representa un problema para aquellas funciones que siempre terminan pero que no

utilizan recursión estructural basada en sintaxis. En estos últimos casos representa un reto expresar en Coq que la función en efecto siempre termina, para dichos casos se han desarrollado diferentes soluciones alternativas (en [Sai10] se hace una comparación de algunas de éstas) e incluso se han desarrollado tesis doctorales con este fin [Bal02; Sac11], sin embargo ninguna de estas soluciones ha sido completamente satisfactoria ni ha sido adoptada ampliamente.

En nuestro caso el parámetro de entrada es una configuración `m`, entonces Coq espera que las llamadas recursivas se hagan sobre un subtérmino de `m` y esto no es así.

En la práctica lo que suele hacer en estos casos es agregar un natural como parámetro que representa la profundidad de la recursión, es decir, la recursión va estar acotada (*bounded recursion*) y por tanto la función siempre terminará, con esto nuestra función queda de la siguiente manera:

```
Fixpoint mexecrec (depthR:nat) (m:mconf) : option mconf :=
match depthR with
| O => Some m
| S k => match m with
           | (nil, s) => Some (nil,s)
           | (IConst n::c,s) => mexecrec k (c, SVNum n::s)
           | (IAdd::c, SVNum n2::SVNum n1::s) =>
             mexecrec k (c,SVNum (n1+n2)::s)
           | _ => None
         end
end.

mexecrec is recursively defined (decreasing on 1st argument)
```

Esta vez hemos definido exitosamente nuestra función, es decir nuestro intérprete de la máquina.

Ahora mostramos algunos ejemplos de cálculos sobre el intérprete `mexecrec`

```
Compute(mexecrec 9 (IConst 5::IConst 2::IAdd::nil,nil)).
     = Some (nil, SVNum 7 :: nil)
     : option mconf

Compute(mexecrec 2 (IConst 5::IConst 2::IAdd::nil,nil)).
     = Some (IAdd :: nil, SVNum 2 :: SVNum 5 :: nil)
     : option mconf

Compute(mexecrec 8 (IConst 4::IAdd::nil,nil)).
     = None
     : option mconf
```

Nótese cómo basta que el primer argumento (el natural) sea un número suficientemente grande para que se calcule completamente la función, por eso Leroy en [JPL12] lo llama combustible (*fuel*).

Toca el turno de verificar que nuestro intérprete cumple con la especificación, es decir, con la semántica de paso grande de la máquina.

```
Lemma BSMsemRtoF:forall mi mf,
BSMsem mi mf ->
exists n, mexecrec n mi = Some mf.
Proof.
intros.

induction H.
exists 0.
simpl.
trivial.

(*Caso IConst*)
inversion IHBSMsem.
exists (S x).
simpl.
trivial.

(*Caso IAdd*)
inversion IHBSMsem.
exists (S x).
simpl.
trivial.
Qed.
```

Ahora podemos utilizar el mecanismo de extracción

```
Extraction mexecrec.

(** val mexecrec : nat -> mconf -> mconf option **)

let rec mexecrec depthR m =
  match depthR with
  | O -> Some m
  | S k ->
    let Pair (c0, s) = m in
    (match c0 with
     | Nil -> Some (Pair (Nil, s))
     | Cons (i, c) ->
       (match i with
```

```
        | IConst n -> mexecrec k (Pair (c, (Cons (n, s))))
        | IAdd ->
          (match s with
           | Nil -> None
           | Cons (s0, l) ->
             (match l with
              | Nil -> None
              | Cons (s1, s2) ->
                mexecrec k (Pair (c, (Cons ((plus s1 s0), s2))))))))
```

obteniendo así un intérprete verificado (que sigue la semántica de paso grande) de la máquina.

Presentamos ahora la semántica de paso pequeño:

Configuración actual		Configuración siguiente	
Código	Pila	Código	Pila
[IConst n] $\cdot\, c$	s	c	[SVNum n] $\cdot\, s$
[IAdd] $\cdot\, c$	[SVNum n_2, SVNum n_1] $\cdot\, s$	c	[SVNum $(n_1 + n_2)$] $\cdot\, s$

y su correspondiente desarrollo en Coq es:

```
Inductive SSMsem: mconf -> mconf -> Prop :=
| SSICons: forall n:nat, forall c:code, forall s:stack,
             SSMsem  (IConst n::c, s) (c, SVNum n::s)
| SSIAdd: forall n1 n2:nat, forall c:code, forall s:stack,
            SSMsem (IAdd::c, SVNum n2::SVNum n1::s)
                        (c, SVNum (n1+n2)::s).
```

Aquí también podemos realizar verificación y así lo haremos. Por tanto como primer paso escribimos el programa, es decir, el intérprete de la máquina. Nótese que este intérprete de la máquina estará basado en la semántica de paso pequeño de ésta, mientras que el intérprete que desarrollamos anteriormente estaba basado en la semántica de paso grande de la misma. De esta manera obtendremos dos intérpretes de la máquina, el que ya hemos desarrollado con base en la semántica de paso grande y el que nos encontramos desarrollando que está basado en la semántica de paso pequeño.

```
Definition mexec (m: mconf) : option mconf :=
match m with
| (nil,s) => Some (nil,s)
| (IConst n::c, s) => Some (c, SVNum n::s)
```

```
| (IAdd::c,SVNum n2::SVNum n1::s) =>
    Some (c,SVNum (n1+n2)::s)
| _ => None
end.
```

Un ejemplo de cálculo sobre este intérprete es el siguiente:

```
Compute(mexec (IConst 5::IConst 2::IAdd::nil,nil)).
     = Some (IConst 2 :: IAdd :: nil, SVNum 5 :: nil)
     : option mconf
```

Ahora realizamos la demostración de que en efecto cumple con la especificación, es decir, con la semántica de paso pequeño de la máquina.

```
Lemma SSMsemRtoF:forall mi mf,
SSMsem mi mf ->
mexec mi = Some mf.
Proof.
intros.
induction H.

(*Caso IConst*)
simpl.
trivial.

(*Caso IAdd*)
simpl.
trivial.
Qed.
```

Utilizamos el mecanismo de extracción

```
Extraction mexec.

(** val mexec : mconf -> mconf option **)

let mexec = function
| Pair (c0, s) ->
  (match c0 with
   | Nil -> Some (Pair (Nil, s))
```

```
  | Cons (i, c) ->
    (match i with
     | IConst n -> Some (Pair (c, (Cons (n, s))))
     | IAdd ->
       (match s with
        | Nil -> None
        | Cons (s0, l) ->
          (match l with
           | Nil -> None
           | Cons (s1, s2) -> Some (Pair (c, (Cons ((plus s1 s0), s2))))))))
```

y obtenemos un intérprete verificado de un único paso que cumple con la semántica de paso pequeño de la máquina.

Observemos que esta vez hemos escrito un intérprete que efectúa un único paso en la evaluación del código de máquina que se le alimenta como entrada. Esto debido a que éste intérprete está basado en la semántica de paso pequeño de la máquina. Para obtener una evaluación completa del código de máquina que se le alimenta, lo que se debe hacer es tomar como especificación la cerradura transitiva y reflexiva "$\rightarrow^*$" de la semántica de paso pequeño.

Notemos aquí que podemos escribir la cerradura transitiva y reflexiva $\rightarrow^*$ de una relación $\rightarrow$ como una definición inductiva utilizando la notación de reglas en lógica, de la siguiente manera:

$$\frac{}{m \rightarrow^* m} \qquad\qquad \frac{m_1 \rightarrow m_2 \qquad m_2 \rightarrow^* m_3}{m_1 \rightarrow^* m_3}$$

donde m_1, m_2 y m_3 en este caso son configuraciones de la máquina.

Con base en lo anterior, podemos expresar la cerradura transitiva y reflexiva de la semántica de paso pequeño de la siguiente manera:

```
Inductive TRCSSMsem: mconf -> mconf -> Prop :=
| TRCR: forall m:mconf, TRCSSMsem m m
| TRCT: forall m1 m2 m3:mconf,
          SSMsem m1 m2 -> TRCSSMsem m2 m3 -> TRCSSMsem m1 m3.
```

Ahora debemos realizar verificación tomando ésta como especificación.

Por lo que primero escribimos el intérprete que posteriormente veremos que cumple con ella.

```
Fixpoint mexecms (depthR:nat) (m:mconf): option mconf :=
match depthR with
| O => Some m
| S n =>
  match mexec m with
    | None => None
    | Some (nil,s) => Some (nil,s)
    | Some mi => mexecms n mi
  end
end.
```

Un ejemplo de cálculo sobre este intérprete es el siguiente.

```
Compute(mexecms 9 (IConst 5::IConst 2::IAdd::nil,nil)).

     = Some (nil, SVNum 7 :: nil)
     : option mconf
```

Como podemos observar éste intérprete lo que hace es simular la cerradura transitiva y reflexiva de la semántica de paso pequeño de la máquina realizando un ciclo sobre el intérprete de un único paso basado en la semántica de paso pequeño de la máquina.

Ahora verificamos que en efecto este intérprete cumple con la especificación, es decir, con la cerradura transitiva y reflexiva de la máquina.

```
Lemma TRCSSMsemRtoF:
forall mi mf,
TRCSSMsem mi mf ->
exists n,
mexecms n mi = Some mf.
Proof.
intros.
induction H.
exists 0.
simpl.
trivial.
apply SSMsemRtoF in H.
inversion IHTRCSSMsem.
exists (1 + x).
simpl.
destruct m2.
rewrite H.
destruct c eqn:?.
destruct x.
```

```
simpl in H1.
trivial.
simpl in H1.
trivial.
trivial.
Qed.
```

Ahora podemos utilizar el mecanismo de extracción

```
Extraction mexecms.

(** val mexecms : nat -> mconf -> mconf option **)

let rec mexecms depthR m =
  match depthR with
  | O -> Some m
  | S n ->
    (match mexec m with
     | Some mi ->
       let Pair (c, s) = mi in
       (match c with
        | Nil -> Some (Pair (Nil, s))
        | Cons (i, l) -> mexecms n mi)
     | None -> None)
```

obteniendo un intérprete verificado de la máquina (basado en la cerradura transitiva y reflexiva de la misma).

Por otro lado, podemos notar cómo el intérprete que sigue la semántica de paso grande dobla en su interior el intérprete que ejecuta muchos pasos.[18] En cambio, en la semántica de paso pequeño está desdoblado, es decir, se requiere explícitamente de un intérprete que realice muchos pasos en un ciclo iterativo sobre el intérprete de un único paso.

Esto parece un punto a favor de la semántica de paso grande ya que permite un nivel de abstracción mayor, lo que se refleja en menos tareas por realizar.

Veamos ahora la equivalencia entre los dos tipos de semántica de la máquina.

Estudiemos primero cómo de la semántica de paso pequeño se sigue la semántica de paso grande. Lo cual se enuncia formalmente en el siguiente teorema.

Teorema 3.7.1. Para todo código c, c_f, para toda pila s, s_f, si $(c, s) \to^* (c_f, s_f)$ entonces $s \vdash c \Rightarrow (c_f, s_f)$.[19]

[18] En un ciclo iterativo donde se ejecuta en cada iteración un único paso.

[19] Estrictamente hablando debemos considerar la cerradura reflexiva de "⇒", es decir, de la semántica de paso grande de la máquina. En lugar de definir un nuevo predicado inductivo con este fin en Coq, por simplicidad simplemente agregamos el constructor `| BSRC: forall k:mconf, BSMsem k k` a la semántica de paso grande de la máquina.

Para poder demostrar este teorema antes debemos demostrar dos lemas.
El primero de ellos es el siguiente.

Lema 3.7.1. Para todo código c_1, c_2, c_3, para toda pila s_1, s_2, s_3, si se tiene por una parte que $s_1 \vdash c_1 \Rightarrow (c_2, s_2)$ y por otra que $s_2 \vdash c_2 \Rightarrow (c_3, s_3)$ entonces $s_1 \vdash c_1 \Rightarrow (c_3, s_3)$.

Como podemos observar, se refiere a la transitividad de la semántica de paso grande.
Su correspondiente desarrollo en Coq es el siguiente:

```
Lemma BSMsemTRN:
forall m1 m2 m3,
BSMsem m1 m2 ->
BSMsem m2 m3 ->
BSMsem m1 m3.
Proof.
intros.
dependent induction H.
trivial.

constructor.
apply IHBSMsem.
trivial.

constructor.
apply IHBSMsem.
trivial.
Qed.
```

El segundo lema que debemos probar lo enunciamos a continuación.

Lema 3.7.2. Para todo código c_1, c_2, para toda pila s_1, s_2, si $(c_1, s_1) \rightarrow (c_2, s_2)$ entonces $s_1 \vdash c_1 \Rightarrow (c_2, s_2)$.

Lo que en general dice que si en la semántica de paso pequeño de una configuración se llega a otra (en un único paso) lo mismo se puede hacer utilizando la semántica de paso grande, en este caso particular se utilizaría la semántica de paso grande para dar el mismo único paso que se dio en la semántica de paso pequeño.
El correspondiente desarrollo en Coq de este teorema es el siguiente.

```
Lemma SSMsemTBSMsem:
forall m1 m2,
SSMsem m1 m2 ->
BSMsem m1 m2.
Proof.
```

```
intros.
destruct H.
constructor.
constructor.
constructor.
constructor.
Qed.
```

Ahora estamos listos para mostrar el desarrollo en Coq del teorema 3.7.1 (dicho desarrollo por supuesto incluye la demostración del teorema).

```
Lemma TRCSSMsemTBSMsem:
forall mi mf,
TRCSSMsem mi mf ->
BSMsem mi mf.
Proof.
intros.
induction H.
constructor.

eapply BSMsemTRN.
apply SSMsemTBSMsem in H.
exact H.
trivial.
Qed.
```

Ahora veamos cómo de la semántica de paso grande se sigue la de paso pequeño, lo cual se enuncia en el teorema siguiente.

Teorema 3.7.2. Para todo código c, c_f, para toda pila s, s_f, si $s \vdash c \Rightarrow (c_f, s_f)$ entonces $(c, s) \rightarrow^* (c_f, s_f)$.

Presentamos ahora el desarrollo en Coq correspondiente a este teorema.

```
Lemma BSMsemTTRCSSMSem:
forall mi mf,
BSMsem mi mf ->
TRCSSMsem mi mf.
Proof.
intros.
induction H.
constructor.
```

```
(*Caso IConst*)
econstructor.
econstructor.
trivial.

(*Caso IAdd*)
econstructor.
econstructor.
trivial.
Qed.
```

Con esto hemos demostrado la equivalencia entre la dos semánticas. Pasemos ahora a la compilación.

Analicemos primero con qué medio contamos para definir formalmente la compilación.

El primero que nos viene a la mente y es natural pensar en él es a través de una función, así la compilación de nuestro de lenguaje de ejemplo es:

$$Compile(e) = \begin{cases} [\text{IConst } n] & \text{si } e = \text{Const n} \\ Compile\ e_1 \cdot Compile\ e_2 \cdot [\text{IAdd}] & \text{si } e = \text{Plus } e_1\ e_2 \end{cases}$$

y su correspondiente desarrollo en Coq es:

```
Fixpoint compile (e:exp) : code :=
match e with
| Const n => IConst n::nil
| Plus e1 e2 => compile e1 ++ compile e2 ++ IAdd::nil
end.
```

Un ejemplo de cálculo sobre `compile` es el siguiente:

```
Compute (compile (Plus (Const 5) (Const 2))).

     = IConst 5 :: IConst 2 :: IAdd :: nil
     : code
```

Esta solución es simple e intuitiva. Mejor aún, como sabemos desde el punto de vista de Coq una función cuenta directamente con contenido computacional y desde luego es una solución que consideramos buena por todo lo anterior.

No obstante (como vimos en la sección 2.3.2) existe otra alternativa, a saber, utilizar la visión unificadora de Khan y utilizar la semántica de paso grande para especificar la compilación, esto es, para nuestro caso se tiene lo siguiente:[20]

$$\frac{}{\text{Const } n \twoheadrightarrow [\text{IConst } n]} \qquad \frac{e_1 \twoheadrightarrow c_1 \qquad e_2 \twoheadrightarrow c_2}{\text{Plus } e_1\ e_2 \twoheadrightarrow c_1 \cdot c_2 \cdot [\text{IAdd}]}$$

Para expresar esto en Coq, como sabemos podemos utilizar una definición inductiva (tal como lo hicimos con la semántica del lenguaje y las de la máquina), así el desarrollo en Coq correspondiente es:

```
Inductive compiler: exp -> code -> Prop :=
| CConst: forall n, compiler (Const n) (IConst n::nil)
| CPlus: forall e1 e2:exp, forall c1 c2:code,
    compiler e1 c1 ->
    compiler e2 c2 ->
    compiler (Plus e1 e2) (c1 ++ c2 ++ IAdd::nil).
```

Ahora siguiendo la verificación debemos ofrecer un programa que cumpla la especificación.

En este caso el programa (la función) `compile` dada arriba nos sirve.

Ahora mostramos que en efecto cumple con la especificación.

```
Lemma compileRtoF: forall e c,
compiler e c ->
compile e = c.
Proof.
intros.
induction H.

(*Caso Const*)
simpl.
trivial.

(*Caso Plus*)
simpl.
rewrite IHcompiler1.
rewrite IHcompiler2.
trivial.
Qed.
```

[20]Nótese que hemos utilizado "⇒" para indicar la semántica de paso grande del lenguaje y la de paso grande de la máquina, con el mismo sentido podríamos utilizar "⇒" para la semántica natural de la compilación (para resaltar aun más este carácter unificador) dejando que se infiera con base en el contexto a cuál de éstas se está haciendo referencia; aquí utilizamos "↠" para indicar explícitamente que se trata de compilación y que no haya lugar a ambigüedad.

Analizando, como vimos contamos con dos opciones para expresar la compilación, a saber por medio una función y a través de la semántica natural. Podemos preguntarnos entonces ¿cuál de éstas debemos utilizar? para responder esta pregunta, a continuación presentamos las características, ventajas y desventajas de cada una de ellas:

1. Función. Primero debemos hacer notar que una función es determinista. Por otra parte, sabemos que al utilizar Coq una función tiene contenido computacional (el tipo del tipo de la función es Set), es decir, por una parte se pueden realizar cálculos sobre ella directamente en Coq y por otra se puede obtener un programa escrito es un lenguaje de programación convencional a partir de ella utilizando el mecanismo de extracción. También sabemos que como una función tiene contenido computacional, toda evaluación de ésta para cualquier valor posible de entrada debe terminar, lo cual implica en particular que una función necesariamente se debe definir para todo caso posible de entrada. Podemos decir adicionalmente que cuando en Coq se define una función no se genera un principio de inducción asociado a ella.[21]

2. Semántica natural (relación definida inductivamente). Aquí primero hacemos notar que una relación en el caso general es no determinista. Por otro lado, cuando utilizamos Coq para expresar una relación definida inductivamente en el caso general y en particular la semántica natural, debemos utilizar una definición inductiva que corresponde a una proposición inductiva (en su tipo figurará el tipo $Prop$). Como sabemos en este caso dicha definición no contará con contenido computacional, es decir, no servirá para realizar cálculos sobre ella, ni se podrá obtener un programa escrito en un lenguaje de programación convencional utilizando el mecanismo de extracción sobre tal definición. No obstante al no contar con contenido computacional no es necesario definirla para todo caso posible de entrada. Por otra parte, como sabemos en Coq cuando damos una definición inductiva se genera un principio de inducción asociado a ella, lo que permite que se puedan utilizar tácticas basadas en este principio al realizar demostraciones.

Por otro lado notemos que en general al utilizar Coq se pueden tener las siguientes visiones:

- Visión matemática. Definir un concepto, establecer propiedades acerca de éste mediante lemas, teoremas y corolarios y realizar sus respectivas demostraciones.

- Visión computacional. Visión matemática junto con un medio para realizar cálculos con base en la definición del concepto.

Entonces podemos generalizar y decir que cuando un concepto se puede expresar mediante una relación definida inductivamente (por ejemplo la semántica natural) en el caso

[21] Aunque existen extensiones experimentales que buscan satisfacer este objetivo, es decir, generar un principio de inducción correspondiente a una función.

general será no determinista y entonces debemos escribirla en Coq utilizando una definición inductiva. Esta definición inductiva corresponde a una proposición inductiva (una donde en su tipo aparecerá el tipo $Prop$). Ahora como caso particular si contamos con una relación determinista entonces además de expresarla en Coq mediante una definición inductiva podemos dar un teorema que enuncie el determinismo de ésta y demostrarlo. En este último caso contaremos ya con la definición inductiva y un teorema que exprese su determinismo y por supuesto podremos establecer propiedades acerca de la definición mediante lemas, teoremas y corolarios y demostrarlos, es decir, cumpliremos ya con la visión matemática, nótese que aquí contaremos con un principio de inducción asociada a la definición y podremos hacer uso de tácticas basadas en éste al realizar las demostraciones. No obstante no contaremos con un medio para realizar cálculos por medio de la definición inductiva, esto es, la definición no tiene contenido computacional, o sea que hasta este punto no se cumple la visión computacional.

Si se desea cumplir la visión computacional lo que podemos hacer es lo siguiente, como sabemos en este caso la definición inductiva corresponde a una proposición inductiva que podemos tomar como especificación y entonces podemos realizar verificación o certificación. En el caso en que realicemos verificación debemos ofrecer un programa (una función) y luego demostrar que ésta cumpla con la especificación, por supuesto esta función tendrá contenido computacional y se podrán realizar cálculos sobre ella directamente en Coq, más aun podemos utilizar el mecanismo de extracción y obtener un programa verificado escrito en un lenguaje de programación convencional, de esta forma cumpliremos ya con la visión computacional. Podemos preguntarnos entonces si contábamos con una relación determinista ¿por qué no escribirla directamente simplemente como una función? analicemos la respuesta tomando como caso particular la semántica natural, la semántica natural en general es no determinista, pero como caso particular puede ser determinista para un lenguaje determinado L, entonces en ese caso ciertamente podemos escribirla directamente como una función y llegaríamos a un intérprete que define (*definitional interpreter*) la semántica del lenguaje, sin embargo de hacerlo así por una parte se pierde la noción de que estamos haciendo uso de la semántica natural y por otra en Coq no contaremos con el principio de inducción y por tanto no podremos hacer uso de las tácticas basada en él al realizar demostraciones, pero ciertamente si esto no es esencial sí podemos expresar directamente nuestra relación determinista como una función.

Ahora analicemos el escenario cuando un concepto se puede expresar como una función, como es el caso de la compilación. En este caso desde luego se puede escribir directamente como una función en Coq y tendríamos contenido computacional, es decir, se cumple la visión computacional. Por otra parte notemos que una función es una caso particular de una relación, entonces en general una función se puede escribir como una relación y en particular una función recursiva se puede escribir como una relación inductiva y como ya sabemos una relación inductiva se expresa en Coq mediante una definición inductiva. Luego ¿por qué deberíamos escribir una función como definición inductiva? y la respuesta es para que en Coq se obtenga un principio de inducción asociado a ella que sirva de ayuda al realizar demostraciones por medio de las tácticas basadas en este prin-

cipio. Si esto último no es realmente importante, entonces simplemente se puede escribir como una función en Coq.

Con base en lo dicho anteriormente, en Coq en general se pueden tener los siguientes escenarios con sus respectivas preferencias:

- Cuando se tiene visión estrictamente matemática. En este caso se prefiere utilizar definiciones inductivas.
- Cuando se tiene visión computacional y las demostraciones no son muy complejas. Aquí se prefiere hacer uso de funciones.
- Cuando se tiene visión computacional y las demostraciones son complejas. En este caso, se prefiere contar tanto con definiciones inductivas (que sirven como especificación al utilizar verificación) como con funciones (que sirven como programas que cumplen la especificación al utilizar verificación) y realizar verificación. De esta forma cuando se enuncien propiedades por medio de lemas, teoremas y corolarios se debe hacer sobre las definiciones inductivas, así se podrá hacer uso de las tácticas basada en el principio de inducción al realizar las demostraciones. Luego se tiene la certeza que el programa verificado cumple con las propiedades que se demostraron, porque cumple con la especificación correspondiente a la definición inductiva sobre la que se enunciaron y demostraron dichas propiedades.

Nótese que se tratan únicamente de preferencias y no de reglas por lo que no necesariamente debe ser así.

Volviendo a nuestro ejemplo, como una función sirve directamente para especificar la compilación y tiene contenido computacional (que es una de las cosas que buscamos), nuestra elección para este caso particular es utilizar la versión funcional únicamente.

No obstante, hemos incluido la versión relacional sólo para ilustrar cómo es posible utilizarla y también damos a continuación como ilustración un lema que muestra que ésta es determinista, esto con el fin de que sirva como referencia en otros escenarios en los que sí se considere adecuado su uso:[22]

```
Lemma compilerDet:
forall e1:exp, forall c1 c2:code,
compiler e1 c1 ->
compiler e1 c2 ->
c1 = c2.
Proof.
intros.
dependent induction e1.

(*Caso Const*)
```

[22]En efecto nosotros utilizaremos tanto una versión relacional como (varias) funcionales en la traducción hacia índices de De Bruijn en nuestro compilador principal.

```
inversion H.
inversion H0.
trivial.

(*Caso Plus*)
inversion H.
inversion H0.
assert(c0 = c4).
apply IHe1_1.
trivial.
trivial.
rewrite H11.
assert(c3= c5).
apply IHe1_2.
trivial.
trivial.
rewrite H12.
trivial.
Qed.
```

Por último nos hace falta demostrar nuestro teorema principal, es decir, el teorema de corrección de nuestro compilador.

Para formular el teorema, debemos elegir una de las semánticas de la máquina, la más conveniente, pues como vimos, las dos semánticas que utilizamos son equivalentes, así demostrar el teorema para la una implica (indirectamente) la demostración para la otra y viceversa.

Sin embargo, con fines ilustrativos demostraremos dos teoremas de corrección: uno utilizando la semántica de paso pequeño y el otro utilizando la semántica de paso grande.

Lo que buscamos es que si dada una expresión e, e se evalúa a un valor v en notación $e \Rightarrow v$ y por otro lado, si al compilar e se obtiene un código c o sea si tiene que $compile(e) = c$, entonces al evaluar c en la máquina debe quedar v en el tope de la pila. Aquí hay un punto a tratar pues v es un valor del lenguaje y no un valor de pila, por lo que no se puede pedir que se deje en el tope de la pila a v. Lo que debemos hacer es, para un valor v del lenguaje definir un valor equivalente en la pila, es decir, compilar[23] un valor del lenguaje a un valor de la pila, de esta manera lo que se debe dejar en el tope de la pila es $compile(v) = v_p$.

Por ejemplo si tenemos la expresión Plus $(\text{Const } 5)(\text{Const } 2)$ y el resultado es el valor Num 7, queremos que al compilar y evaluar quede en el tope de la pila el valor de pila SVNum 7. De esta manera, en general el equivalente de un valor Num n del lenguaje es un valor SVNum n en la pila, por tanto, nuestra compilación de valores del lenguaje a valores de la pila es:

[23] Así como compilamos expresiones del lenguaje a código de la máquina, debemos compilar ahora valores del lenguaje a valores de la máquina.

$CompVal(v) = \begin{cases} \text{SVNum } n & \text{si } v = \text{Num } n \end{cases}$

y su correspondiente desarrollo en Coq es el siguiente:

```
Definition compileval(v:val): sval :=
match v with
| Num n => SVNum n
end.
```

Ahora nos encontramos listos para formular nuestro teorema de corrección.
Comencemos utilizando la semántica de paso grande:

Teorema 3.7.3. Para toda expresión e, para todo valor v, si

$$e \Rightarrow v$$

entonces para toda pila s, para todo código c, para todo valor de pila v_p,

$$s \vdash c \Rightarrow ([], v_p \cdot s)$$

donde $c = Compile(e)$ y $v_p = CompVal(v)$.

En realidad para fortalecer nuestra hipótesis probaremos una versión más fuerte del teorema de corrección (de la cual el teorema anterior es un corolario):

Teorema 3.7.4. Para toda expresión e, para todo valor v, si

$$e \Rightarrow v$$

entonces para toda pila s, para todo código c, d, para todo valor de pila v_p,

$$s \vdash c \cdot d \Rightarrow (d, v_p \cdot s)$$

donde $c = Compile(e)$ y $v_p = CompVal(v)$.

Como podemos observar, d es un código cualquiera que concatenamos a c en la evaluación de la máquina, esta vez en lugar de terminar con el código vacío tras haber ejecutado c debe quedar d, esto con la única intención de fortalecer nuestra hipótesis. Nótese como si tomamos $d = []$ nos queda nuestro teorema original.

Antes del presentar el desarrollo correspondiente a este teorema en Coq, presentamos primero un lema en Coq acerca de la asociatividad de la concatenación de listas (denotada como "++") que nos será de utilidad en la demostración de nuestro teorema de corrección

```
(**
 * Lema que enuncia la asocitividad de la concatenación "++"
 * de listas.
 *
 **)
Lemma AConcatL:
forall A:Type, forall l1 l2 l3:list A,
(l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
Proof.
intros.
firstorder.
Qed.
```

Damos ahora el desarrollo correspondiente a nuestro teorema de corrección utilizando la semántica de paso grande en Coq

```
(**
 * Teorema de corrección de la compilación utilizando
 * la semántica de paso grande de la máquina.
 *
 **)
Theorem CorrecCBS:
forall e:exp, forall v:val,
expBSS e v ->
forall s:stack, forall d:code,
BSMsem (compile e ++ d, s)
(d,(compileval v)::s).
Proof.
intros.
dependent induction H.

(*Caso Const*)
simpl.
constructor.
constructor.

(*Caso Plus*)
simpl.
assert(
(compile e1 ++ compile e2 ++ IAdd :: nil) ++ d
=
compile e1 ++ ((compile e2 ++ IAdd :: nil) ++ d)
).
apply AConcatL.
rewrite H1.
eapply BSMsemTRN.
```

```
eapply IHexpBSS1.

assert(
(compile e2 ++ IAdd :: nil) ++ d
=
compile e2 ++ IAdd :: nil ++ d
).
apply AConcatL.
rewrite H2.
eapply BSMsemTRN.
eapply IHexpBSS2.
simpl.
constructor.
constructor.
Qed.
```

Toca el turno de presentar nuestro teorema utilizando la semántica de paso pequeño de la máquina:

Teorema 3.7.5. Para toda expresión e, para todo valor v si

$$e \Rightarrow v$$

entonces para toda pila s, para todo código c, para todo valor de pila v_p,

$$(c, s) \rightarrow^* ([], v_p \cdot s)$$

donde $c = Compile(e)$ y $v_p = CompVal(v)$.

De igual forma fortalecemos la hipótesis:

Teorema 3.7.6. Para toda e expresión, para todo v valor, si

$$e \Rightarrow v$$

entonces para toda s pila, para todo c, d código, para todo v_p valor de pila,

$$(c \cdot d, s) \rightarrow^* (d, v_p \cdot s)$$

donde $c = Compile(e)$ y $v_p = CompVal(v)$.

Esta vez antes de presentar el desarrollo correspondiente al teorema anterior, presentamos un lema en Coq que enuncia la transitividad de la cerradura transitiva y reflexiva de la semántica de paso pequeño de la máquina. Este lema lo ocuparemos en la demostración del teorema de corrección anterior.

```
(**
 * Lema que enuncia la transtividad de la cerradura transitiva
 * y reflexiva de la semántica de paso pequeño.
 *
 **)
Lemma TRCSSMsemTRN:
forall m1 m2 m3,
TRCSSMsem m1 m2 ->
TRCSSMsem m2 m3 ->
TRCSSMsem m1 m3.
Proof.
intros.
induction H.
trivial.
econstructor.
exact H.
apply IHTRCSSMsem.
trivial.
Qed.
```

Ahora mostramos el desarrollo en Coq correspondiente al teorema de corrección utilizando la semántica de paso pequeño de la máquina

```
(**
 * Teorema de corrección de la compilación utilizando
 * (la cerradura transitiva y reflexiva de) la semántica de
 * de paso pequeño de la máquina.
 *
 **)
Theorem CorrecCSS:
forall e:exp, forall v:val,
expBSS e v ->
forall s:stack, forall d:code,
TRCSSMsem (compile e ++ d, s)
(d,(compileval v)::s).
Proof.
intros.
dependent induction H.

(*Caso Const*)
simpl.
econstructor.
econstructor.
constructor.

(*Caso Plus*)
```

```
simpl.
assert(
(compile e1 ++ compile e2 ++ IAdd :: nil) ++ d
=
compile e1 ++ ((compile e2 ++ IAdd :: nil) ++ d)
).
apply AConcatL.
rewrite H1.
eapply TRCSSMsemTRN.
eapply IHexpBSS1.

assert(
(compile e2 ++ IAdd :: nil) ++ d
=
compile e2 ++ IAdd :: nil ++ d
).
apply AConcatL.
rewrite H2.
eapply TRCSSMsemTRN.
eapply IHexpBSS2.

simpl.
econstructor.
constructor.
constructor.
Qed.
```

Habiendo demostrado ya la corrección de la compilación, podemos ahora utilizar nuestro compilador para calcular la compilación de una expresión aritmética, por ejemplo $5 + 2$, más aun podemos ejecutar el código que genera el compilador en cualquiera de los dos intérpretes de la máquina con los que contamos. Lo anterior se ilustra a continuación:

```
Compute (mexecrec 3 ((compile (Plus (Const 5) (Const 2))), nil)).
     = Some (nil, SVNum 7 :: nil)
     : option mconf

Compute (mexecms 3 ((compile (Plus (Const 5) (Const 2))), nil)).
     = Some (nil, SVNum 7 :: nil)
     : option mconf
```

Podemos ahora utilizar el mecanismo de extracción sobre nuestro compilador y de esta manera obtener un compilador correcto verificado escrito en un lenguaje de programación convencional como OCaml, listo para utilizarse en la vida real, tal como se muestra a continuación:

```
Extraction compile.
(** val compile : exp -> code **)

let rec compile = function
| Const n -> Cons ((IConst n), Nil)
| Plus (e1, e2) -> app (compile e1) (app (compile e2) (Cons (IAdd, Nil)))
```

con lo anterior hemos alcanzado el objetivo principal de nuestro ejemplo, es decir, hemos obtenido un compilador correcto verificado de un lenguaje de expresiones aritméticas escrito en un lenguaje de programación convencional, listo para utilizarse en la vida real. Adicionalmente contamos con dos intérpretes correctos verificados de la máquina abstracta que utilizamos, en los cuales se puede ejecutar el código generado por nuestro compilador.

Con esto damos por concluido el desarrollo de nuestro compilador correcto para un lenguaje de expresiones aritméticas.

Por otra parte damos por finalizado el presente capítulo y nos disponemos ahora a estudiar cómo partiendo del cálculo lambda se puede llegar a un lenguaje de programación, tema que estudiaremos en el capítulo siguiente.

CAPÍTULO 4

Hacia nuestro compilador

En vías de desarrollar nuestro compilador, es necesario recorrer antes un camino que nos brinde conceptos, medios y capacidad de elección que nos serán de utilidad al realizar nuestro compilador.

4.1. Cálculo lambda como lenguaje de programación

Presentamos ya el cálculo lambda en la sección 2.2.2 como un sistema formal, ahora estudiaremos aquí su uso como lenguaje de programación.

Como primer paso hacia este objetivo presentamos el teorema de estandarización (también conocido como Church-Rosser II).

Teorema 4.1.1 (Estandarización)**.** Si $M \twoheadrightarrow N$ entonces existe una reducción estándar de M a N, donde una reducción estándar es aquella donde las contracciones se realizan de izquierda a derecha.

Corolario 4.1.1. Si N es una forma normal de M, entonces existe una reducción normal β de M a N, donde una reducción normal es aquella donde se contrae primero el redex a la extrema izquierda y más externo (*leftmost outermost first*).[1]

Con base en esto, podemos decir que si un programa se evalúa a un valor final utilizando de forma no determinista (es decir, contrayendo cualquier redex en un paso de la evaluación) las reglas de la definición 2.2.11 entonces, también se puede evaluar de forma

[1]Para una definición formal de reducción estándar y reducción normal así como las pruebas de este teorema y corolario consultar [CFC58].

determinista (eligiendo un redex en específico en cada paso) utilizando las mismas reglas. Con esto hemos obtenido una estrategia de evaluación determinista conocida como reducción en orden normal (*normal order reduction*).

Que se siga una estrategia determinista para evaluar es ideal porque da certeza y claridad al programador de cómo se evaluará un programa.

Por otra parte, si partimos del cálculo lambda puro éste no cuenta con constantes. Por ejemplo si quisiéramos tener expresiones aritméticas, entre otras cosas, hace falta agregar constantes para los números naturales, con este fin se pueden codificar diferentes tipos de constantes (por ejemplo, naturales o enteros) y diferentes tipos de operadores (por ejemplo, operadores aritméticos y operadores condicionales) como términos del cálculo lambda; por ejemplo, para codificar los números naturales existe una codificación conocida como los numerales de Church (*Church numerals*) [Chu41; Bar12], aunque se pueden codificar diferentes entidades como términos del cálculo lambda, en general esto representa una labor tediosa. En lugar de eso lo que suele hacerse[2] es agregar constantes y operadores primitivos nativamente al cálculo lambda puro.[3]

Por ejemplo, en la expresión $\lambda x.(x + 2)\ 5$ se han agregado primitivamente constantes para los enteros y el operador "+".[4]

Definición 4.1.1. Un *combinador* es un $\lambda-$término cerrado puro, es decir, un término que no contiene variables libres ni constantes.

Del mismo modo el cálculo lambda puro no cuenta con un soporte especial para la recursión,[5] lo que se hace para obtener recursión es utilizar el combinador Y (que obsérvese que es un $\lambda-$término por lo que no se está agregando nada nuevo al cálculo). El combinador Y cumple con lo siguiente:

$$Yx \twoheadrightarrow x(Yx)$$

lo que intuitivamente significa que dado un operador (o una función) x, Y lo desdobla, es decir, aumenta en uno la profundidad de la recursión y la función se evalúa, si con ese paso basta para terminar se concluye la evaluación; en otro caso se vuelve a desdoblar la función y se repite el proceso hasta que finalmente termina la evaluación o el proceso se repite infinitamente (si el operador o función no terminan para la entrada dada, es decir, si nunca se llega a un caso básico).

[2]En particular en el contexto de un lenguaje de programación para el cual se realizará una implementación.

[3]Es por eso que al estudiar el cálculo lambda generalmente se habla de una familia de cálculos que en realidad se trata del cálculo lambda puro con diferentes extensiones, según sea el caso. Cada una de ellas se considera un cálculo.

[4]Formalmente realizar la operación +, significa hacer una contracción δ y así en realidad estamos agregando la regla δ a la definición 2.2.11 que permite utilizar operadores primitivos nativamente, obteniendo así la reducción $\beta\delta$.

[5]Lo que llama la atención es que el cálculo lambda puro es Turing-computable, por lo que se esperaría que exista una manera de realizar recursión y en efecto es a través del combinador Y.

Existen diferentes formas de definir el combinador Y una debida a Alan Turing es como $Y \equiv UU$, donde $U \equiv \lambda ux.x(uux)$.[6]

Otra forma de obtener recursión es agregar un operador de punto fijo μ nativamente (así como agregamos el operador +) al cálculo lambda. Este método es más adecuado para nuestros objetivos ya que es más directo y claro por una parte, y por otra resulta más eficiente, ya que de esta forma no se requiere de reescritura, por eso es el que nosotros utilizaremos en nuestro compilador. Presentaremos la semántica del operador μ en el siguiente capítulo.

4.1.1. Evaluación

Landin [Lan64] fue uno de los primeros en utilizar el cálculo lambda para modelar un lenguaje de programación y propuso una estrategia de evaluación en la que todo término[7] t tiene un valor, este valor es relativo a cierta información previa, en esta información previa se provee un valor para cada identificador libre en t; a esta información previa se le llama *entorno*. Desde luego este entorno es relativo a la evaluación que se está realizando. Por otro lado, este entorno se puede considerar como una función que asocia a ciertos identificadores (los identificadores libres de un término) un valor.

Por ejemplo, si se tiene el programa:

```
let val x = 5 in let val y = 2 in x + y end end;
```

y se requiere evaluar el término `x + y`, entonces suponiendo que se cuenta con un entorno e, donde $e(x) = 5$ y $e(y) = 2$ el valor del término `x + y` será 7, es decir, el entorno debe contener la información $(x, 5), (y, 2)$ (recabada previamente al evaluar los enunciados `let`). Entonces el entorno se puede representar como una lista de pares $(\text{identificador}, \text{valor})$, de esta manera en nuestro ejemplo sería: $[(x, 5), (y, 2)]$.

Como bajo esta estrategia todo término tiene un valor, también lo tiene una abstracción, es decir, un término de la forma $\lambda x.M$. En este caso dado un entorno e lo que se busca al evaluar es dar como resultado una función tal que el darle un argumento se evalúe su cuerpo M en un nuevo entorno e', donde e' es el entorno que resulta de asignarle al parámetro x el valor que se le pasa como argumento.

Por ejemplo, si tenemos el siguiente programa:

```
let val y = 2 in (fn x => y + x) 5 end
```

[6] Para conocer más acerca del combinador Y consúltese [HS08; Han04; Bar12].

[7] *Applicative expression* en términos de Landin. En el contexto de un lenguaje de programación se prefiere la palabra expresión para referirse a un término del cálculo lambda, nosotros utilizaremos indistintamente término o expresión.

y queremos evaluar la abstracción `fn x => y + x`, entonces nuestro entorno e es $[(y, 2)]$. El parámetro es x y el cuerpo de la función es `y + x`, esta es la información relevante que se necesita, así al evaluar el término `(fn x => y + x) 5` se evalúa primero `(fn x => y + x)` a un valor que guarda la información importante que mencionamos y posteriormente se evalúa `5`, el resultado de evaluar `5` es el valor `5` y finalmente, se crea el nuevo entorno e' como $[(x, 5)(y, 2)]$ y ahí se evalúa `y + x`, y el resultado es 7. El valor de una abstracción se puede representar como un paquete de información al que se llama *cerradura* que consiste en una parte para el entorno y otra para el cuerpo de la función. En la parte del entorno se guarda el entorno relativo al momento en que se evaluó la abstracción y el parámetro de ésta. Así una cerradura también se puede considerar como una triada: (entorno,parámetro,cuerpo), de este modo la evaluación de `(fn x => y + x)` en nuestro ejemplo daría como resultado la cerradura: $([(y, 2)], x, y+x)$ y a partir de ésta para evaluar `(fn x => y + x) 5` se sabe que se debe formar un nuevo entorno e' con base en el de la cerradura. e' se debe construir agregando al identificador correspondiente al parámetro almacenado en la cerradura el valor 5, es decir, $e' = [(x, 5), (y, 2)]$ y en ese nuevo entorno evaluar el código contenido en la cerradura, es decir, `y + x` lo que da como resultado 7, que es justo lo que se esperaba. Una cerradura por supuesto es un valor.

Si e se considera el entorno actual, entonces la estrategia de evaluación consiste en:

- Un identificador x se evalúa a v si $e(x) = v$.
- Una abstracción $\lambda x.M$ se evalúa a la cerradura (e, x, M)
- Una aplicación $e_1\ e_2$ se evalúa evaluando primero e_1, luego e_2 y finalmente aplicando el resultando del primero al segundo.[8]

Nótese que un enunciado `let val x = e1 in e2 end` se puede considerar azúcar sintáctica de una aplicación $(\lambda x.e_2)e_1$.

Landin [Lan64] utiliza una máquina conocida como la máquina SECD para mecanizar la estrategia anterior.[9]

4.1.2. Índices de De Bruijn

Tradicionalmente un término del cálculo lambda se representa con base en su definición, es decir, una ocurrencia de una variable ligada se reconoce utilizando el mismo nombre para la ocurrencia y para la variable de la λ, por ejemplo en el término $\lambda x.\lambda y.x + y$ se sabe que la ocurrencia de x está ligada debido a que x es la variable de λx y está en el alcance de ésta. Esta representación se conoce como basada en nombres.

[8] Es decir, realizando el proceso que ejemplificamos arriba, construyendo el nuevo entorno e' y evaluado el cuerpo de la función en él.

[9] La máquina SECD original como la presenta Landin en [Lan64] trabaja directamente sobre expresiones lambda en lugar de instrucciones. Nosotros presentaremos la semántica formal de una variante de la máquina SECD en el siguiente capítulo, la cual utilizaremos en nuestro compilador.

Existe otra representación que se conoce como notación de De Bruijn [Bru72] o índices de De Bruijn (debido a su creador Nicolaas De Bruijn). En ésta una ocurrencia de una variable se representa con un número natural k que denota la "distancia" o "la variable ligada por la k-ésima" λ. Por ejemplo, el término $\lambda x.\lambda y.x + y$ se representa como $\lambda.\lambda.1 + 0$ ya que la ocurrencia de x en el término $x + y$ está ligada por λx que es la 1-ésima λ o que está a distancia 1, la ocurrencia de y en $x + y$ está ligada por λy que es la 0-ésima λ o que está a distancia 0. Similarmente, el término $\lambda x.\lambda y.x\ y$ se representa como $\lambda.\lambda.1\ 0$. Otro ejemplo es: $(\lambda x.(\lambda y.(\lambda z.(x\ (y\ z)))))$ representado como $(\lambda.(\lambda.(\lambda.(2\ (1\ 0)))))$.

Obsérvese que formalmente siempre estamos trabajando con sintaxis abstracta sólo que usamos sintaxis concreta al explicar algunos ejemplos porque es más clara para el lector, así por ejemplo, si escribimos $\lambda x.x + 1$ en realidad se trata de: Lam x (Plus (Var x) (Const 1)).

Para formalizar esta traducción (o compilación) podemos utilizar (como ya sabemos) la semántica natural.[10] Así tenemos juicios de la forma:

$$\Pi \vdash M \rightarrow T$$

donde Π es un contexto que lleva el seguimiento (contiene el conocimiento previo) de las variables libres de M. Así el juicio se lee: el término M con representación basada en nombres y variables libres en Π se traduce a la representación de De Bruijn T.

A continuación presentamos las reglas de la traducción utilizando semántica natural.

1) $$\frac{}{[x] \cdot \Pi \vdash \text{Var } x \rightarrow \text{Var } 0}$$

es decir, si la última variable que se metió al contexto es x y la variable que se está traduciendo es x entonces se traduce a 0 porque corresponde a la λ más cercana.

2) $$\frac{\Pi \vdash \text{Var } x \rightarrow \text{Var } n \qquad x \neq y}{[y] \cdot \Pi \vdash \text{Var } x \rightarrow \text{Var } S\ n}$$

si la última variable que se metió al contexto es y que es distinta a la variable x que se está traduciendo, entonces la ocurrencia de x no corresponde a la λ más cercana, por tanto se debe buscar la siguiente λ más cercana, lo que se logra sacando y del contexto y por supuesto se debe incrementar en 1 el índice de la variable x debido a que no corresponde a la λ más cercana (este proceso se debe repetir hasta que se encuentre la variable en el contexto).

3) $$\frac{[x] \cdot \Pi \vdash e \rightarrow t}{\Pi \vdash \text{Lam } x\ e \rightarrow \text{Lam } t}$$

si se trata de una abstracción, la variable de la abstracción x se debe meter al contexto y en ese contexto se debe traducir el cuerpo de la abstracción.

[10] Nótese que para especificar la traducción utilizaremos sintaxis abstracta y no la concreta que hemos estado usando en los ejemplos.

Así, claramente $\lambda x.x$ se traduce a $\lambda.0$ porque se aplica esta regla y luego inmediatamente la número 1.

4) $$\frac{\Pi \vdash e_1 \rightarrow t_1 \qquad \Pi \vdash e_2 \rightarrow t_2}{\Pi \vdash \text{App } e_1\ e_2 \rightarrow \text{App } t_1\ t_2}$$

Esta regla simplemente indica que la traducción de una aplicación se define homomorficamente.

5) $$\frac{\Pi \vdash e_1 \rightarrow t_1 \qquad \Pi \vdash e_2 \rightarrow t_2}{\Pi \vdash \text{Plus } e_1\ e_2 \rightarrow \text{Plus } t_1\ t_2}$$

Análoga al caso anterior.

El árbol de demostración para nuestro término de ejemplo: $\lambda x.\lambda y.x + y$ es el siguiente:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[x] \vdash \text{Var } x \rightarrow \text{Var } 0}\,1}{[y,x] \vdash \text{Var } x \rightarrow \text{Var } 1}\,2 \quad \dfrac{}{[y,x] \vdash \text{Var } y \rightarrow \text{Var } 0}\,1}{[y,x] \vdash \text{Plus Var } x \text{ Var } y \rightarrow \text{Plus Var } 1 \text{ Var } 0}\,5}{[x] \vdash \text{Lam } y \text{ (Plus Var } x \text{ Var } y) \rightarrow \text{Lam (Plus Var } 1 \text{ Var } 0)}\,3}{[] \vdash \text{Lam } x \text{ (Lam } y \text{ (Plus Var } x \text{ Var } y)) \rightarrow \text{Lam (Lam (Plus Var } 1 \text{ Var } 0))}\,3$$

De esta forma queda demostrado que la traducción de $\lambda x.\lambda y.x + y$ es $\lambda\lambda 1 + 0$. Nótese que siempre inicialmente el contexto Π es vacío y existe una traducción a índices de De Bruijn sólo si M, el término a traducir, es cerrado. La demostración de la traducción de los otros ejemplos se puede hacer de manera similar.

Analicemos ahora el programa:

```
let val x = 5 in let val y = 2 in x + y end end;
```

que como mencionamos, se puede considerar azúcar sintáctica de $((\lambda x.((\lambda y.x{+}y)2))5)$. Cuando se va evaluar el término `x + y` el entorno es: $[(y,2),(x,5)]$, nótese que al agregar un nuevo elemento (el cual es un par (nombre,valor)) al entorno siempre consideramos que se mete por la izquierda, es decir, en realidad el entorno es una pila cuyo tope está a la izquierda. Esto es, la evaluación de $((\lambda x.((\lambda y.x + y)2))5)$ se realiza de la siguiente manera: como es una aplicación primero se evalúa: $(\lambda x.((\lambda y.x + y)2))$ a la cerradura $([], x, ((\lambda y.x + y)2))$, luego se evalúa 5 al valor 5 y después se evalúa el código de la cerradura: $((\lambda y.x + y)2)$ en el entorno $[(x,5)]$; a su vez se evalúa primero $(\lambda y.x + y)$ a la cerradura $[(x,5), y, x + y]$ y luego 2 al valor 2 y finalmente se evalúa el código de la cerradura: $x + y$ en el entorno $[(y,2),(x,5)]$ lo que da como resultado 7.

Por otra parte, en lugar de considerar un enunciado `let` como azúcar sintáctica, el enunciado `let` se puede agregar nativamente al cálculo lambda. De este modo la evaluación de `let val x = 5 in let val y = 2 in x + y end end` (cuya sintaxis abstracta es: Let x 5 (Let y 2 $x + y$)), es: primero se agrega $(x, 5)$ al entorno: $[(x, 5)]$, y luego se evalúa `let val y = 2 in x + y end`, así se agrega $(y, 2)$ al entorno, quedando el entorno así: $[(y, 2), (x, 5)]$. Posteriormente se evalúa $x + y$ obteniendo 7 como resultado; nótese cómo esta evaluación es más directa porque no se necesitan evaluar aplicaciones (lo que implicaría crear cerraduras) y por tanto es más eficiente. Es por esta razón que cuando un lenguaje se diseña tomando en cuenta que se va implementar, se prefiere soportar nativamente un enunciado `let`.[11]

Si se agrega el enunciado `let` nativamente su regla correspondiente a la traducción a índices de De Bruijn es:

$$6)\ \frac{\Pi \vdash e_1 \rightarrow t_1 \qquad [x] \cdot \Pi \vdash e_2 \rightarrow t_2}{\Pi \vdash \text{Let } x\ e_1\ e_2 \rightarrow \text{Let } t_1\ t_2}$$

En el tratamiento anterior se ilustra cómo el entorno se comporta como una pila (con tope a la izquierda). Ahora, notemos cómo la traducción a índices de De Bruijn de $((\lambda x.((\lambda y.x + y)2))5)$ es:[12] $((\lambda.((\lambda.\bar{1} + \bar{0})2))5)$ obsérvese cómo el índice de De Bruijn correspondiente a una variable también corresponde exactamente a la posición (índice) de la variable en el entorno. Así cuando se va a evaluar el término $x + y$ el entorno es: $[(y, 2), (x, 5)]$ y para obtener el valor de x se debe hacer una búsqueda en el entorno y otra búsqueda para obtener el valor de y, en cambio si se tiene: $\bar{1} + \bar{0}$ se sabe que exactamente el valor de la variable a la izquierda del + (que originalmente era x) está en la posición 1 del entorno y la de la derecha (que originalmente era y) está en la posición 0. De esta forma ya no se deben realizar búsquedas.[13] Nótese que se puede incluso prescindir de guardar el nombre de una variable en el entorno (pues como ya no se realizan búsquedas ya no es necesaria) y guardar únicamente el valor, así el entorno al evaluar $\bar{1} + \bar{0}$ sería: $[2, 5]$ y de esta manera obtendríamos $5 + 2$ y como resultado final 7. Con esta estrategia hemos prescindido por completo del uso de nombres de variables. Veamos el ejemplo completo: $((\lambda x.((\lambda y.x + y)2))5)$ se traduce a: $((\lambda.((\lambda.\bar{1} + \bar{0})2))5)$ de aquí se evalúa primero $(\lambda.((\lambda.\bar{1} + \bar{0})2))$ a la cerradura $([], ((\lambda.\bar{1} + \bar{0})2))$, nótese cómo en esta cerradura ya no es necesario guardar el nombre de la variable porque ya no hay tal y está implícito en la forma de manejar el entorno. De este modo una cerradura bajo este esquema es un par (entorno,cuerpo); luego 5 se evalúa a 5 y después se mete al entorno quedando el entorno así: $[5]$ y ahí se evalúa $((\lambda.\bar{1} + \bar{0})2)$. Para esto se evalúa primero $(\lambda.\bar{1} + \bar{0})$ a la cerradura

[11] Leroy [Ler90, pág. 30] menciona que el caso especial del `let` es tan común que se puede evaluar de la forma en la que lo hicimos en lugar de evaluar una aplicación. Y esta forma de evaluar es más simple y eficiente.

[12] Hemos utilizado $\bar{n}$ para indicar que n se trata de una variable representada por el índice n y no una constante entera del lenguaje.

[13] Y por tanto se mejora la eficiencia en tiempo de ejecución. A cambio se aumenta el tiempo de compilación debido a la traducción a índices de De Bruijn y justo es el objetivo que se busca al hacer optimizaciones disminuir el tiempo de ejecución de un programa aunque se aumente el tiempo de compilación de éste.

$([5], \bar{1} + \bar{0})$ y después 2 al valor 2, posteriormente se mete 2 al entorno, es decir, el entorno queda como: $[2, 5]$ y luego se evalúa $\bar{1} + \bar{0}$ de aquí se llega a $5 + 2$ y finalmente al resultado 7. Nótese cómo el uso de índices simplifica la evaluación.

Por otro lado (habiendo presentado ya la regla para el Let), formalmente la traducción de Let x 5 (Let y 2 (Plus (Var x) (Var y)) es:

$$\dfrac{\dfrac{}{[\,] \vdash 5 \to 5} \quad \dfrac{\dfrac{}{[x] \vdash 2 \to 2} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{[x] \vdash \text{Var } x \to \text{Var } 0}\,1}{[y, x] \vdash \text{Var } x \to \text{Var } 1}\,2 \quad \dfrac{}{[y, x] \vdash \text{Var } y \to \text{Var } 0}\,1}{[y, x] \vdash \text{Plus (Var } x\text{) (Var } y\text{)} \to \text{Plus (Var } 1\text{) (Var } 0\text{)}}\,5}{[x] \vdash \text{Let } y\ 2\ \text{Plus (Var } x\text{) (Var } y\text{)} \to \text{Let } 2\ \text{(Plus (Var } 1\text{) (Var } 0\text{))}}\,6}{[\,] \vdash \text{Let } x\ 5\ \text{(Let } y\ 2\ \text{Plus (Var } x\text{) (Var } y\text{))} \to \text{Let } 5\ \text{(Let } 2\ \text{(Plus (Var } 1\text{) (Var } 0\text{)))}}\,6$$

y entonces en la evaluación de: Let 5 (Let 2 (Plus (Var 1) (Var 0))) primero se mete el 5 al entorno quedando: $[5]$, luego se mete el 2 quedando: $[2, 5]$, posteriormente se evalúa (Plus (Var 1) (Var 0)) a (Plus 5 2) y finalmente se obtiene 7.

4.1.3. Cálculo λ_v

Como vimos Landin [Lan64] delinea una estrategia de evaluación haciendo uso de entornos y cerraduras y, para mecanizarla utiliza la máquina SECD, con base en este trabajo, Plotkin [Plo75] busca qué es lo correspondiente en términos puramente del cálculo lambda, a qué corresponde esta estrategia presentada por Landin haciendo uso de la visión original del cálculo lambda, y lo que encuentra es que la estrategia de evaluación de Landin básicamente corresponde al cálculo lambda con las reglas que presentamos en la definición 2.2.11 pero restringiendo el uso del axioma β de la siguiente manera:

(β) $(\lambda x.M)N \twoheadrightarrow [N/x]M$ si N es un valor.

que en términos de lenguajes de programación significa pedir que se evalúen antes los argumentos de la función (que la función misma). A este cálculo lo llamó cálculo λ_v. Luego, define una noción de reducción estándar para este cálculo y menciona que se puede definir una noción de reducción normal para éste y se llega así a lo que conocemos como llamada por valor (*call by value*)[14] que consiste en contraer primero el redex más a la izquierda más adentro que no esté dentro de una abstracción λ (*leftmost innermost redex not inside a λ abstraction first*).[15]

Nótese cómo no está permitido contraer dentro de una abstracción (justo eso lo pudimos notar cuando evaluamos una abstracción, pues ésta no consistía en evaluar algo en el cuerpo de la lambda sino que se evaluaba a una cerradura). Y lo anterior es sensato en el

[14] Si se sigue el mismo camino pero partiendo del axioma β sin restricciones (a este cálculo lo llama cálculo λ_N) se llega a llamada por nombre (*call by name*).

[15] Mientras que en *call by name* se contrae primero el redex más a la izquierda más afuera que no esté dentro de una abstracción λ (*left most outermost redex not inside a λ abstraction first*).

contexto de un lenguaje de programación en el que no tiene sentido evaluar una función si antes no se le han asignado los argumentos a los parámetros formales.

Por supuesto, bajo este enfoque de Plotkin estamos trabajando directamente sobre un cálculo lambda y entonces podemos formular su semántica haciendo uso de sustituciones, como es natural en el cálculo lambda (sin tener que hacer uso de entornos ni de cerraduras). Podemos dar entonces en semántica natural las reglas para llamada por valor:

$$\frac{}{\text{Var } x \Rightarrow \text{Var } x}\ \text{Variable}$$

$$\frac{}{\text{Lam } x\ e \Rightarrow \text{Lam } x\ e}\ \text{Abstracción}$$

$$\frac{e_1 \Rightarrow \text{Lam } x\ e \qquad e_2 \Rightarrow v_2 \qquad e[v_2/x] \Rightarrow v}{\text{App } e_1\ e_2 \Rightarrow v}\ \text{Aplicación}$$

Desde luego, también podemos dar una semántica natural que formalice el enfoque de Landin, es decir utilizando entornos y cerraduras:

$$\frac{}{[(x_0, v_0), \ldots, (x_i, v_i), \ldots, (x_n, v_n)] \vdash \text{Var } x_i \Rightarrow v_i}\ \text{Variable}$$

$$\frac{}{\Gamma \vdash \text{Lam } x\ e \Rightarrow \text{Clos } x\ e\ \Gamma}\ \text{Abstracción}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Clos } x\ c_1\ \Gamma_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2 \qquad [(x, v_2)] \cdot \Gamma_1 \vdash c_1 \Rightarrow v}{\Gamma \vdash \text{App } e_1\ e_2 \Rightarrow v}\ \text{Aplicación}$$

Aquí como sabemos un entorno Γ almacena pares $(\text{variable}, \text{valor})$, es decir, es de la forma $\Gamma = [(x_1, v_1), \ldots, (x_i, v_i), \ldots, (x_n, v_n)]$ donde v_i es el valor correspondiente a la variable x_i. Por otra parte como estudiamos en la sección 2.3.2, al utilizar la semántica natural se deben especificar los posibles valores finales. Aquí como vimos en la sección 4.1.1 una cerradura (Γ, x, M) donde Γ es un entorno, x una variable y M una expresión es un posible valor final, la sintaxis abstracta correspondiente a esta cerradura es "Clos x M Γ", por su parte v y v_2 en las reglas también denotan valores finales. Entonces en este caso la semántica natural establece una relación que va de entornos y expresiones (del lenguaje, que en este caso son términos del cálculo lambda) a valores finales. Es fácil notar que las reglas anteriores formalizan por medio de la semántica natural la estrategia de evaluación presentada informalmente por Landin en [Lan64] y que dimos al finalizar la sección 4.1.1.

Por supuesto, las dos semánticas anteriores deben ser equivalentes y entonces nos preguntamos cuál debemos utilizar.

Fijémonos primero que un $\lambda-$término visto como un programa sólo tiene sentido si es cerrado (es decir, si no tiene variables libres), por ejemplo, si tenemos el programa: $x+3$ no tiene sentido porque al no conocerse el valor de x no se puede evaluar la expresión, esto sucede porque x es una variable libre en este programa. En cambio si tenemos $\lambda x.x + 3$ entonces este programa si tiene sentido porque es una función y está se evalúa a una cerradura en el enfoque de Landin o a una abstracción en el enfoque de Plotkin, nótese

cómo en este caso no hay variables libres. Por otro lado, si se tiene el programa $\lambda x.((g\ x) + 3)$ no tiene sentido porque no se sabe qué es g, es decir, el valor de g, y por tanto no se puede evaluar. Por lo anterior, un programa sólo nos interesa si se conoce todas sus entidades, o sea, si es un término cerrado.

Por tanto en adelante únicamente consideraremos términos cerrados.

4.2. El problema de captura de variables

En el cálculo lambda original como lo presentamos en la definición 2.2.11 (sin considerar por el momento la regla α) sólo se realiza una sustitución si se usa el axioma β:

$(\beta)\ \ (\lambda x.M)N \twoheadrightarrow [N/x]M$

es decir, cuando se tiene una aplicación de la forma $(\lambda x.M)N$.

Si no se considera la operación de sustitución con cuidado como en la definición 2.2.7 se pueden cometer errores.

Veamos un ejemplo, consideremos el término cerrado:

$$\lambda y.((\lambda x.\underbrace{\lambda y.x + y}_{M})\underbrace{(y + 1)}_{N})3$$

entonces (si no se considera una reducción específica) se puede reducir cualquier redex, así tomando M y N como está indicado, se puede obtener lo siguiente:

$$\lambda y.\lambda y.((\underline{y} + 1) + y)3$$

lo cual es un error porque en el subtérmino $N = y + 1$, y es una variable libre (aunque globalmente está ligada por la λ más externa) y ahora quedó ligada por la λ más interna, esto se conoce como el problema de captura de variables. Obsérvese que esto no puede suceder si se realiza la operación de sustitución según la definición 2.2.7, en esta definición está implícito utilizar una conversión α según se requiera, de este modo en nuestro ejemplo primero tenemos que usar el axioma α (es decir, renombrar), entonces se tienen los siguientes pasos:

1. $\lambda y.((\lambda x.\underbrace{\lambda z.x + z}_{M})\underbrace{(y + 1)}_{N})3$
2. $\lambda y.\lambda z.((y + 1) + z)3$

y de esta forma ya no hay error.

Entonces, para no tener el problema de captura de variables contamos con las siguientes soluciones:[16]

[16]Que son las que consideramos más relevantes y a nuestro alcance pero en la literatura se pueden encontrar otras, aunque generalmente son éstas o algunas otras derivadas a partir de éstas.

1. Utilizar el axioma α según se requiera (es decir, renombrar).

2. Asegurar que el término N no tenga variables libres. Es decir en general cuando en un término t haya un subtérmino de la forma $(\lambda x.M)N$ (o sea una aplicación) y se vaya a realizar la sustitución correspondiente a esa aplicación si el término N es cerrado entonces no se presentará el problema de captura de variables.

3. Utilizar índices de De Bruijn.[17]

Nosotros utilizaremos la solución 2., es decir, aseguraremos que el término N no tenga variables libres. Pero ¿cómo podemos asegurar que N no tenga variables libres?

Definición 4.2.1. Una reducción que no contrae redex dentro de una abstracción λ se le llama *reducción débil* (*weak reduction*).

Es el caso que si se usa una reducción débil[18] entonces el término N al realizar cualquier contracción β no contiene variables libres, tal como lo afirma Leroy en [Ler16a].

Por supuesto, la estrategia de llamada por valor es un caso particular de reducción débil y por tanto se asegura que el término N de cualquier contracción β no contiene variables libres.

Como se puede notar, es afortunado que la estrategia de llamada por valor sea un caso particular de reducción débil y así no tengamos que preocuparnos por el renombre de variables.

Podemos entonces retomar la pregunta de cuál de las dos semánticas para la estrategia de llamada por valor debemos utilizar:

1. Semántica con entornos y cerraduras debida a Landin.

2. Semántica con sustituciones debida a Plotkin.

Como acabamos de ver, al realizar sustituciones se pueden cometer errores si no se realizan con cuidado, al realizar pruebas con papel y lápiz suele considerarse que se hará uso del axioma α según se requiera, es decir, se utilizará renombrado (como lo indica la definición 2.2.7) y así no se tendrá el problema de captura de variables. Aunque este enfoque es el que se considera en pruebas hechas con papel y lápiz, suele dar problemas cuando se formaliza en un asistente de pruebas, debido a que formalizar la sustitución haciendo uso de renombrado resulta ser difícil y complejo en el contexto de los asistentes de pruebas. Por tanto, como respuesta para aliviar este problema, han surgido diferentes soluciones al problema de realizar sustitución con renombrado en un asistente de pruebas. Dos de las más importantes son: sintaxis abstracta de orden superior (*Higher Order Abstract Syntax*) [PE88] e índices de De Bruijn.

[17] En el presente trabajo no mencionamos cómo utilizar lo índices de De Bruijn para evitar el problema de captura de variables porque no los utilizamos con este fin.

[18] Partiendo de un término cerrado.

Afortunadamente en nuestro caso no tenemos ese problema; es decir, podemos realizar sustituciones sin preocuparnos de realizar renombrado ya que no es necesario, porque como mencionamos la estrategia de llamada por valor es un caso especial de reducción débil y en la reducción débil siempre el término por el cual se va a sustituir resulta ser un término cerrado.

Por su parte en la semántica con entornos y cerraduras propuesta por Landin no se realizan sustituciones, pero podemos preguntarnos por cuál sería el problema correspondiente al de captura de variables en el contexto de entornos y cerraduras. ¿Este existe o será que bajo este esquema no existe un problema análogo? En realidad, este problema no nos compete a nosotros pues como mencionamos, afortunadamente utilizando llamada por valor no se tiene el problema de captura de variables.

Tomando esto en cuenta podemos formalizar cualquiera de las dos semánticas en Coq.

Entonces, ¿por cuál nos debemos decantar? Comparemos con un ejemplo muy simple estos dos enfoques. Supongamos que tenemos el programa: $\lambda x.(x + x + x + x + x)9$. Entonces si utilizamos sustituciones esto implica que debemos realizar cinco sustituciones textuales de x por 9 al evaluar el programa. En cambio si utilizamos entornos y cerraduras, basta con asignar a x el valor de 9 en el entorno para realizar el mismo trabajo. Desde luego este ejemplo se puede convertir en uno más catastrófico (que requiera mucho más sustituciones). Esto nos permite ver que el uso de entornos y cerraduras es más eficiente que el de sustituciones. En efecto como lo señala Leroy [Ler16a] utilizar sustitución textual es un enfoque ineficiente. Por tanto hemos elegido utilizar en el presente trabajo la semántica con entornos y cerraduras con base en el criterio de eficiencia.[19]

Con todo el conocimiento que desarrollamos anteriormente, contamos ya con todas las herramientas necesarias para realizar el desarrollo, formalización y verificación de nuestro compilador para nuestro lenguaje principal, labor que llevaremos a cabo en el siguiente capítulo.

[19]No obstante nótese que afirmamos que estas dos semánticas son equivalentes y la demostración se puede formalizar en un asistente de pruebas, en particular en Coq, realizarlo queda fuera del objetivo de este trabajo pero se deja como posible trabajo futuro.

CAPÍTULO 5

Nuestro compilador

5.1. Introducción

En el presente capítulo desarrollaremos nuestro compilador principal y realizaremos la verificación de la corrección del mismo.

Como primer paso debemos saber cuál será nuestro lenguaje fuente y cuál será nuestro lenguaje objetivo.

Como mencionamos en la introducción, nuestro lenguaje fuente será un lenguaje funcional, por tanto podemos tomar como punto de partida el cálculo lambda puro y hacerle las extensiones necesarias para que de soporte a todo nuestro lenguaje fuente.

Por otro lado, debemos tomar como guía nuestro ejemplo motivacional de la sección 1.1, es decir, debemos proveer soporte para poder expresar y calcular el factorial de un número natural cualquiera.

Así que podemos y debemos agregar nativamente constantes para número naturales, además como se necesita realizar operaciones aritméticas sobre ellos, debemos agregar operadores aritméticos (consideramos adición, multiplicación y sustracción). Por otro lado como la función se define por casos, es necesario agregar enunciados condicionales es decir, enunciados If, para lo cual es necesario añadir constantes booleanas. Adicionalmente como es necesario expresar condiciones sobre los naturales se requieren operadores booleanos de comparación (aquí solamente consideramos el operador de igualdad porque es el único necesario en nuestro ejemplo motivacional pero se pueden agregar todos lo que se deseen de manera análoga). Por otro lado, ofreceremos soporte para definiciones locales, es decir, debemos añadir enunciados Let primitivamente.[1] Por último hace

[1]Como mencionamos en la introducción no es estrictamente necesario considerar definiciones locales, sin embargo, las consideramos porque es común que los lenguajes de la vida real las soporten, además de que muchos programas se escriben haciendo uso de ellas. Por otro lado como vimos en la sección 4.1.1 éstas

falta agregar el soporte de recursión para lo cual como mencionamos haremos uso del operador de punto fijo μ.

Por su parte, nuestro lenguaje objetivo será el código de máquina de la SECD moderna.[2]

Teniendo claro ya cuál será nuestro lenguaje fuente y nuestro lenguaje objetivo. Podemos mencionar que nuestro compilador consistirá de dos etapas, que son las siguientes:

1. Compilación de nuestro lenguaje fuente (notación basada en nombres) a notación de De Bruijn.

2. Compilación de nuestro lenguaje fuente en notación de De Bruijn a código de máquina de la SECD moderna.

Como vimos en la sección 4.1.2 resulta más eficiente evaluar una expresión utilizando índices de De Bruijn que evaluar la misma expresión utilizando notación basada en nombres. No obstante es insensato pedir al programador que escriba un programa usando índices con base en ciertas reglas. Es por ello que introducimos una etapa en nuestro compilador que se encarga de llevar a cabo la traducción de nuestro lenguaje fuente que utiliza notación basada en nombres a índices de De Bruijn. De esta forma por un lado liberamos al programador de esta tarea y por el otro se gana eficiencia en tiempo de ejecución. A cambio es necesario agregar (e implementar) una nueva etapa en nuestro compilador y se aumenta el tiempo de compilación (el dedicado a realizar la etapa cuando se compila un programa). Pero nótese que justo este es el enfoque que utilizan los compiladores optimizadores, a saber, agregar etapas de optimización para que disminuya el tiempo de ejecución de los programas que generan aunque se aumente el tiempo de compilación de los mismos debido a que en ese caso se tienen que realizar los cálculos correspondientes a las etapas de optimización al compilar un programa.

Luego más adelante veremos que la máquina SECD moderna también utiliza índices de De Bruijn por lo que resulta natural la traducción de nuestro lenguaje fuente en notación de De Bruijn al código de esta máquina.

Por otra parte, vale la pena mencionar que seguiremos la visión computacional (y no únicamente la matemática). Esto es, en general no nos basta con demostrar propiedades de los conceptos que presentemos sino que deseamos contar con un medio para realizar cálculos con base en ellos. Esto (de manera principal pero no única) en particular quiere decir que no nos basta con demostrar la corrección de nuestro compilador, sino que en última instancia deseamos contar con un compilador del cual tengamos la certeza que es correcto y que podamos utilizar en la vida real para compilar programas escritos en nuestro lenguaje fuente.

Para realizar la verificación de la corrección de nuestro compilador en Coq, la idea general es seguir la misma estrategia que utilizamos al desarrollar nuestro compilador de ejemplo de expresiones aritméticas en la sección 3.7.

---

se pueden considerar azúcar sintáctica de una aplicación, sin embargo, como discutimos en la sección 4.1.2 dar soporte nativo para éstas resulta más eficiente.

[2]La cual presentaremos más adelante en la sección 5.5.

Habiendo mencionado todo lo anterior, damos ahora los puntos que iremos presentando durante el transcurso del presente capítulo:

1. Sintaxis abstracta de nuestro lenguaje fuente (basada en notación de nombres).
2. Semántica natural de nuestro lenguaje fuente haciendo uso de entornos y cerraduras. Aquí utilizaremos verificación para obtener un intérprete correcto verificado de nuestro lenguaje fuente (con notación basada en nombres). Si bien no es estrictamente necesario que desarrollemos este intérprete nos sirve de experimentación.[3]
3. Sintaxis abstracta de nuestro lenguaje fuente basada en notación de De Bruijn.
4. Semántica natural de nuestro lenguaje fuente en notación de De Bruijn haciendo uso de entornos y cerraduras. Aquí también utilizaremos verificación para de esta manera obtener un intérprete correcto verificado de nuestro lenguaje esta vez basado en notación de De Bruijn, de nuevo sólo con fines de experimentación.
5. Compilación de nuestro lenguaje fuente a notación de De Bruijn. Para expresar esta compilación utilizaremos la semántica natural.[4] Después daremos y demostraremos un lema que enuncie el determinismo de esta traducción. Luego utilizaremos verificación para obtener un compilador correcto verificado que sirva para calcular la traducción de cualquier programa de nuestro lenguaje fuente de notación basada en nombres a notación de De Bruijn.
6. Teorema de corrección de la compilación de nuestro lenguaje fuente a notación de De Bruijn.
7. Sintaxis abstracta de las instrucciones de la máquina SECD moderna.
8. Semántica de paso pequeño de la máquina SECD moderna. Utilizaremos verificación para obtener un intérprete correcto verificado de un único paso correspondiente a esta semántica. Posteriormente presentaremos la cerradura transitiva y reflexiva de la misma y realizaremos verificación para obtener así un intérprete correcto verificado de muchos pasos que simule la cerradura transitiva y reflexiva de la semántica de paso pequeño.
9. Semántica de paso grande de la máquina SECD moderna. Utilizaremos verificación para obtener de esta manera un intérprete correcto verificado que siga la semántica de paso grande.
10. Lema de equivalencia entre las dos semánticas. Primero enunciaremos y demostraremos el lema que dicta que de la semántica de paso pequeño se sigue la de paso grande y luego lo haremos en el sentido contrario.

[3] Por ejemplo, para poder observar directamente el resultado que devuelve para cierto programa y constatar que si compilamos dicho programa y lo ejecutamos en el intérprete de la máquina también se obtiene el mismo resultado.

[4] Siguiendo el enfoque unificador de Khan que estudiamos en la sección 2.3.2.

11. Compilación de nuestro lenguaje fuente en notación de De Bruijn a código de máquina de la SECD moderna. Aquí expresaremos la compilación como función (por lo que no hace falta dar un programa para calcular la compilación puesto que ya se cuenta con uno).[5]

12. Teorema de corrección de la compilación de nuestro lenguaje fuente en notación de De Bruijn a código de máquina de la SECD moderna utilizando la semántica de paso pequeño de la máquina.

13. Teorema de corrección de la compilación de nuestro lenguaje fuente en notación de De Bruijn a código de máquina de la SECD moderna utilizando la semántica de paso grande de la máquina.

Cabe mencionar que utilizamos la semántica natural para especificar el comportamiento de nuestro lenguaje fuente, porque consideramos por un lado que es simple e intuitiva y por otro porque ofrece un nivel alto de abstracción.[6] Por otra parte, al utilizar la semántica natural para el lenguaje nos pareció muy claro, simple y adecuado seguir el enfoque unificador de Khan, es por ello que utilizamos la semántica natural para expresar la traducción a índices de De Bruijn y fue también esto lo que nos motivó a desarrollar una semántica de paso grande para la máquina ya que originalmente la semántica de la máquina era una semántica de paso pequeño.

Por otro lado, el utilizar entornos y cerraduras en lugar de sustituciones obedece al criterio de eficiencia (tal como lo estudiamos en la sección 4.2).

El formular una nueva semántica para la máquina desde luego nos llevó a demostrar la equivalencia entre éstas.

Como podemos observar al demostrar que las dos semánticas son equivalentes, basta con dar un teorema de corrección de la compilación utilizando una de ellas. Sin embargo, se enunció y demostró un teorema de corrección de la compilación para cada una de ellas con fines didácticos.

Vale señalar que para cada uno de los puntos que iremos presentado, aunque lo mostremos por completo en abstracto, únicamente presentaremos un fragmento de su correspondiente desarrollo en Coq, esto debido a cuestiones de espacio. Pero desde luego se puede consultar el desarrollo completo en Coq en [Zúñ15b]. Esto es, por ejemplo, para el desarrollo en Coq correspondiente a la semántica natural del lenguaje fuente (en notación de nombres) únicamente se muestran las reglas para: constantes, variables, abstracciones, el operador aritmético de adición, el operador de punto fijo y aplicaciones, pero por ejemplo no se muestran (entre otras) la regla para el let ni la regla para el operador booleano de comparación de igualdad. Esto no quiere decir que no se hayan realizado, como mencionamos se puede consultar el desarrollo completo en [Zúñ15b].

Antes de comenzar nuestro desarrollo, debemos dejar en claro que damos por concedido que se han realizado ya las etapas de análisis léxico, análisis sintáctico y análisis semántico del compilador.

[5] Es decir, en este caso no hace falta realizar verificación como lo explicamos en la sección 3.7.

[6] Lo que nos permite evitar inmiscuirnos en detalles que no son relevantes para el presente trabajo.

Habiendo dicho lo anterior, nos encontramos listos para comenzar a desarrollar nuestro compilador, lo cual haremos en la siguiente sección.

5.2. Notación basada en nombres

5.2.1. Sintaxis abstracta

Comenzaremos presentado la sintaxis abstracta de nuestro lenguaje fuente (basada en notación de nombres).

e	::=	Const n	$n \in \mathbb{N}$
	|	Constb b	$b \in \mathbb{B}$
	|	Op e e	Op$\in$\{Plus, Minus, Times, Eq\}
	|	Var x	
	|	If e e e	
	|	Let x e e	
	|	Lam x e	
	|	Mu f x e	
	|	App e e	

y los valores son:

v	::=	Num n	$n \in \mathbb{N}$
	|	Bool b	$b \in \mathbb{B}$
	|	Clos x e Γ	
	|	Clos$_{rec}$ f x e Γ	

e denota una expresión del lenguaje, es decir un programa y como sabemos cuando se utiliza semántica de paso grande para cada expresión se regresa un valor final,[7] y por tanto es necesario definir los posibles valores finales, es decir, los valores a los que se puede evaluar una expresión. Los valores a los que se pueden evaluar nuestros programas son naturales, booleanos y cerraduras. Recordemos que como vimos en el capítulo 4 una abstracción se evalúa a una cerradura de aquí que se puedan devolver cerraduras como valores finales. Por otro lado debido a que estamos utilizando el operador μ para soportar recursión podemos pensar en μ como una abstracción recursiva, análoga a una abstracción λ así, si una abstracción λ se evalúa a una cerradura, una "abstracción μ" se evalúa a una cerradura recursiva, nótese que lo que diferencia a una cerradura normal de una cerradura recursiva es que en esta última también se guarda el nombre de la función y esto con el objeto de que cuando se evalúe también se puede acceder (es decir, desdoblar la definición de la función) al valor de la función en el cuerpo de la misma.

[7] Claro en el supuesto que la evaluación termina y que hay al menos una regla para cada una de las subexpresiones que conforman la expresión.

Por otra parte, obsérvese que Const n (que representa una constate natural) es una expresión (que es objeto de evaluación), en cambio Num n es un valor que se regresa como resultado y que no es susceptible de evaluación. Así nuestra semántica va de expresiones a valores.

Por su parte Γ es un entorno en el sentido que discutimos en el capítulo 4. Es decir, es un entorno que sirve para llevar el seguimiento de los valores de las variables (donde éstas se representan por medio de identificadores), por lo que almacena pares (id, v) donde id es el nombre de un identificador y v el valor correspondiente a éste. Por otro lado, este entorno (como vimos en la sección 4.1.2) se maneja como una pila (con tope a la izquierda). Con base en lo anterior podemos decir que Γ es de la forma $[(x_0, v_0), \ldots, (x_n, v_n)]$.

Presentamos ahora el desarrollo correspondiente a la sintaxis abstracta y a los valores en Coq.

```
(*Sintaxis abstracta basada en nombres*)
Inductive MMLexp: Set :=
| Const : nat -> MMLexp
| Plus: MMLexp -> MMLexp ->MMLexp
| Var: string -> MMLexp
| Lam: string -> MMLexp -> MMLexp
| Mu: string -> string -> MMLexp -> MMLexp
| App: MMLexp -> MMLexp -> MMLexp.

(*Valores con nombres*)
Inductive MMLval: Set :=
| Bool: bool -> MMLval
| Num: nat -> MMLval
| Clos: string -> MMLexp -> list (string*MMLval) -> MMLval
| Closr: string -> string -> MMLexp -> list (string*MMLval) -> MMLval.

(*Entornos G de la semántica con nombres*)
Definition MMLenv := list (string*MMLval).
```

Daremos ahora en la próxima sección la semántica correspondiente a esta representación.

5.2.2. Semántica natural con notación basada en nombres

En esta sección presentaremos las reglas y axiomas correspondientes a las semántica natural del lenguaje fuente con notación basada en nombres.

Las reglas de la semántica son entonces de la siguiente forma:

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

que se lee partiendo del entorno Γ que contiene los valores de los identificadores libres de la expresión e, la expresión e se evalúa al valor final v.

$$\frac{}{\Gamma \vdash \text{Const } n \Rightarrow \text{Num } n}\ \text{Constante entera}$$

$$\frac{}{\Gamma \vdash \text{Constb } b \Rightarrow \text{Bool } b}\ \text{Constante booleana}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Num } n_1 \qquad \Gamma \vdash e_2 \Rightarrow \text{Num } n_2}{\Gamma \vdash \text{Plus } e_1\ e_2 \Rightarrow \text{Num } (n_1 + n_2)}\ \text{Adición}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Num } n_1 \qquad \Gamma \vdash e_2 \Rightarrow \text{Num } n_2}{\Gamma \vdash \text{Minus } e_1\ e_2 \Rightarrow \text{Num } (n_1 - n_2)}\ \text{Sustracción (con } (n_1 - n_2) \geq 0)$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Num } n_1 \qquad \Gamma \vdash e_2 \Rightarrow \text{Num } n_2}{\Gamma \vdash \text{Times } e_1\ e_2 \Rightarrow \text{Num } (n_1 * n_2)}\ \text{Producto}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Num } n_1 \qquad \Gamma \vdash e_2 \Rightarrow \text{Num } n_2}{\Gamma \vdash \text{Eq } e_1\ e_2 \Rightarrow \text{Bool } n_1 = n_2}\ \text{Comparación de igualdad}$$

$$\frac{}{[(x_0, v_0), \ldots, (x_i, v_i), \ldots, (x_n, v_n)] \vdash \text{Var } x_i \Rightarrow v_i}\ \text{Variable}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad [(x, v_1)] \cdot \Gamma \vdash e_2 \Rightarrow v}{\Gamma \vdash \text{Let } x\ e_1\ e_2\ \Rightarrow v}\ \text{Let}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Bool } true \qquad \Gamma \vdash e_2 \Rightarrow v}{\Gamma \vdash \text{If } e_1\ e_2\ e_3 \Rightarrow v}\ \text{If (true)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Bool } false \qquad \Gamma \vdash e_3 \Rightarrow v}{\Gamma \vdash \text{If } e_1\ e_2\ e_3 \Rightarrow v}\ \text{If (false)}$$

$$\frac{}{\Gamma \vdash \text{Lam } x\ e \Rightarrow \text{Clos } x\ e\ \Gamma}\ \text{Abstracción}$$

$$\frac{}{\Gamma \vdash \text{Mu } f\ x\ e \Rightarrow \text{Clos}_{rec}\ f\ x\ e\ \Gamma}\ \text{Punto fijo}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Clos } x\ c_1\ \Gamma_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2 \qquad [(x, v_2)] \cdot \Gamma_1 \vdash c_1 \Rightarrow v}{\Gamma \vdash \text{App } e_1\ e_2 \Rightarrow v}\ \text{Aplicación}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Clos}_{rec}\ f\ x\ c_1\ \Gamma_1 \quad \Gamma \vdash e_2 \Rightarrow v_2 \quad [(x, v_2), (f, (\text{Clos}_{rec}\ f\ x\ c_1\ \Gamma_1))] \cdot \Gamma_1 \vdash c_1 \Rightarrow v}{\Gamma \vdash \text{App } e_1\ e_2 \Rightarrow v}$$

Presentamos ahora el desarrollo correspondiente a esta semántica en Coq.

```
(**
 * Función que dado un entorno G de la semántica con nombres
 * y dado un nombre x de una variable regresa el valor correspondiente
 * a x en caso de que x se encuentre en G y None en otro caso.
 *
 **)
Fixpoint access (G:MMLenv) (x:string) : option MMLval :=
match G with
| nil => None
| (i,v)::Gs => if string_dec x i then Some v else access Gs x
end.

(**
 *  Semántica de paso grande del lenguaje fuente con notación
 *  basada en nombres.
 *
 **)
Inductive BSSMML : MMLenv -> MMLexp -> MMLval -> Prop :=
| BSSConst: forall G:MMLenv, forall i :nat,
              BSSMML G (Const i) (Num i)
| BSSPlus: forall G:MMLenv, forall e1 e2:MMLexp, forall n1 n2:nat,
             BSSMML G e1 (Num n1) ->
             BSSMML G e2 (Num n2) ->
             BSSMML G (Plus e1 e2) (Num (n1+n2))
| BSSVar: forall G:MMLenv, forall x:string, forall v:MMLval,
            access G x = Some v -> BSSMML G (Var x) v

| BSSLam: forall G:MMLenv, forall x:string, forall e:MMLexp,
            (*La cerradura debe tener la variable que
               liga para cuando se realice una app*)
            BSSMML G (Lam x e) (Clos x e G)

| BSSMu: forall G:MMLenv, forall f x:string, forall e:MMLexp,
           BSSMML G (Mu f x e) (Closr f x e G)

| BSSApp: forall G G1:MMLenv, forall x:string,
          forall e1 e2 c1:MMLexp, forall v v2:MMLval,
              BSSMML G e1 (Clos x c1 G1) ->
              BSSMML G e2 v2 ->
              BSSMML ((x,v2)::G1) c1 v ->
              BSSMML G (App e1 e2) v

| BSSAppr: forall G G1:MMLenv, forall f x:string,
           forall e1 e2 c1:MMLexp, forall v v2:MMLval,
             BSSMML G e1 (Closr f x c1 G1) ->
             BSSMML G e2 v2 ->
             BSSMML ((x,v2)::(f,(Closr f x c1 G1))::G1) c1 v ->
             BSSMML G (App e1 e2) v.
```

Como podemos observar el nombre dado a esta semántica en Coq es `BSSMML`. Por lo que tenemos: `BSSMML G e v` si y sólo si $G \vdash e \Rightarrow v$.

Ahora realizamos verificación. Esto es tomamos la semántica del lenguaje representada como la proposición inductiva `BSSMML` en Coq como una especificación y debemos dar un programa, es decir, un intérprete que la cumpla.

A continuación damos nuestro intérprete en Coq.

```
(**
 *  Intérprete del lenguaje fuente con notación basada en nombres.
 *  Este intérprete cumple con la especificación de la semántica
 *  de paso grande basada en nombres del lenguaje fuente.
 *
 **)
Fixpoint eval (depthR:nat)(G:MMLenv )(e:MMLexp) : option MMLval :=
match depthR with
| O => None
| S m =>
match e with
| Const n => Some (Num n)
| Plus e1 e2 =>
  match eval m G e1 with
    | Some (Num n1) =>
      match eval m G e2 with
        | Some (Num n2) => Some (Num (n1 + n2))
        | _ => None
      end
    | _ => None
  end
| Var x => access G x
| Lam x e  => Some (Clos x e G)
| Mu f x e  => Some (Closr f x e G)
| App e1 e2 =>
  match eval m G e1 with
    | Some (Clos x c1 G1) =>
      match eval m G e2 with
          | Some v2 => eval m ((x,v2)::G1) c1
          | _ => None
      end
    | Some (Closr f x c1 G1) =>
      match eval m G e2 with
        | Some v2 =>
          eval m ((x,v2)::(f,(Closr f x c1 G1))::G1) c1
        | _ => None
      end
    | _=> None
  end
end
end.
```

Y ahora demostramos que en efecto cumple con la especificación, es decir, con la semántica.[8]

```
(**
 * Lema que enuncia que el intérprete "eval" cumple con la especificación
 * de la semántica de paso grande basada en nombres del lenguaje fuente.
 *
 **)
Lemma evalc:
forall G e v,
BSSMML G e v ->
exists n, eval n G e = Some v.
```

Con lo anterior (en concreto en la función `eval`) hemos obtenido un intérprete de nuestro lenguaje. Más aún podemos utilizarlo para calcular el programa correspondiente a nuestro ejemplo motivacional:

```
(*Ejemplo de cálculo intérprete eval*)
Compute (eval 19 nil
(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))).
     = Some (Num 120)
     : option MMLval
```

lo que nos deja con una gran satisfacción al poder observar directamente que se obtiene el resultado esperado, es decir, el factorial de 5 es 120. Lo anterior es gracias a seguir el enfoque computacional. Y nos da la certeza de que vamos por un buen camino.

Mejor aún podemos utilizar el mecanismo de extracción y obtener un intérprete verificado de nuestro lenguaje escrito en OCaml, listo para ser usado en la vida real, como se muestra a continuación.

```
Extraction eval.

(** val eval : nat -> mMLenv -> mMLexp -> mMLval option **)

let rec eval depthR g e =
```

[8]Recordemos que se puede consultar el desarrollo completo en Coq en [Zúñ15b]. En particular la demostración de este lema. Lo mismo vale en adelante.

```
match depthR with
| O -> None
| S m ->
  (match e with
   | Const n -> Some (Num n)
   | Plus (e1, e2) ->
     (match eval m g e1 with
      | Some m0 ->
        (match m0 with
         | Num n1 ->
           (match eval m g e2 with
            | Some m1 ->
              (match m1 with
               | Num n2 -> Some (Num (plus n1 n2))
               | _ -> None)
            | None -> None)
         | _ -> None)
      | None -> None)
   | Var x -> access g x
   | Lam (x, e0) -> Some (Clos (x, e0, g))
   | Mu (f, x, e0) -> Some (Closr (f, x, e0, g))
   | App (e1, e2) ->
     (match eval m g e1 with
      | Some m0 ->
        (match m0 with
         | Clos (x, c1, g1) ->
           (match eval m g e2 with
            | Some v2 -> eval m (Cons ((Pair (x, v2)), g1)) c1
            | None -> None)
         | Closr (f, x, c1, g1) ->
           (match eval m g e2 with
            | Some v2 ->
              eval m (Cons ((Pair (x, v2)), (Cons ((Pair (f, (Closr (f, x,
                c1, g1)))), g1)))) c1
            | None -> None)
         | _ -> None)
      | None -> None))
```

Presentaremos ahora la semántica de nuestro lenguaje fuente en notación de De Bruijn en la siguiente sección.

5.3. Índices de De Bruijn

Por otro lado vimos cómo es más simple y eficiente (desde el punto de vista de espacio y evaluación) trabajar con índices de De Bruijn pues manejando el entorno como lo estudiamos en la sección 4.1.2 éstos resultan ser los índices de los identificadores en el entorno

y así en principio establecer un valor de un identificador toma tiempo constante lo mismo que consultar el valor de un identificador. En efecto podemos dar una semántica natural con entornos y cerraduras que trabaje sobre índices de De Bruijn formalizando nuestra discusión de la sección 4.1.2 de esta manera habremos ganado en principio eficiencia en tiempo de ejecución. Desde luego es insensato pedir al programador que escriba un programa utilizando la notación de De Bruijn. Pero lo que sí podemos hacer es lo siguiente: dado un programa escrito de forma normal, es decir, utilizando nombres para las variables (como lo hicimos en la sección anterior) transformarlo (compilarlo) en un programa que en el se utilicen índices de De Bruijn. Y esta tarea la podemos mecanizar agregando un etapa a nuestro compilador que se encargue de esta traducción y así lo haremos.

Por tanto a continuación presentaremos la sintaxis abstracta de nuestro lenguaje utilizando índices de De Bruijn, así como la semántica natural con entornos y cerraduras utilizando índices de De Bruijn, para luego discutir la traducción a índices de De Bruijn.

5.3.1. Sintaxis Abstracta

Presentamos primero la sintaxis abstracta de nuestro lenguaje fuente utilizando notación de De Bruijn.

e	::=	ConstI n	$n \in \mathbb{N}$
	|	ConstbI b	$b \in \mathbb{B}$
	|	OpI e e	OpI$\in$\{PlusI, MinusI, TimesI, EqI\}
	|	VarI n	$n \in \mathbb{N}$
	|	IfI e e e	
	|	LetI e e	
	|	LamI e	
	|	MuI e	
	|	AppI e e	

y los posibles valores son:

v	::=	NumI n	$n \in \mathbb{N}$
	|	BoolI b	$b \in \mathbb{B}$
	|	ClosI e Ω	
	|	ClosI$_{rec}$ e Ω	

nótese cómo Var x (donde x es un nombre) de la representación en nombres se ha transformado aquí en Var n donde n es un índice. Por su parte los respectivos identificadores de la abstracción y del operador μ han desaparecido debido a que como vimos para el caso de la abstracción en la sección 4.1.2 están implícitos en la forma en que se maneja el entorno. Algo similar sucede para las cerraduras y cerraduras recursivas. Obsérvese cómo también los valores finales cambian.

Por supuesto, como vimos en la sección 4.1.2 bajo este esquema ya no es necesario guardar el nombre de las variables en el entorno. Así nuestra representación del entorno

consiste únicamente en una pila de valores de la forma $[v_0, \ldots, v_n]$ al que llamamos Ω para diferenciarlo del entorno que se utiliza en la representación con nombres.

Nuestra representación de la sintaxis abstracta y de los valores con índices de De Bruijn en Coq es la siguiente:

```
(*Sintaxis abstracta basada en índices de De Bruijn*)
Inductive MMLDBexp: Set :=
| ConstI : nat -> MMLDBexp
| PlusI: MMLDBexp -> MMLDBexp ->MMLDBexp
| VarI: id -> MMLDBexp
| LamI: MMLDBexp -> MMLDBexp
| MuI : MMLDBexp -> MMLDBexp
| AppI: MMLDBexp -> MMLDBexp -> MMLDBexp.

(*Valores con índices de De Bruijn*)
Inductive MMLDBval: Set :=
| BoolI: bool -> MMLDBval
| NumI: nat -> MMLDBval
| ClosI: MMLDBexp -> list MMLDBval -> MMLDBval
| ClosrI: MMLDBexp -> list MMLDBval -> MMLDBval.

(*Entornos O de la semántica con índices de De Bruijn*)
Definition MMLDBenv := list MMLDBval.
```

5.3.2. Semántica natural con índices de De Bruijn

Presentamos ahora la semántica natural con índices de De Bruijn donde las reglas son de la forma:[9]

$$\Omega \vdash e \rightarrowtriangle v$$

Aquí "$\vdash$" denota una relación ternaria entre un entorno Ω (que contiene los valores de los identificadores libres en e, nótese que estos identificadores están representados por índices de De Bruijn en e), una expresión e (que utiliza notación de De Bruijn) y un valor v (que utiliza notación de De Bruijn, al que se evalúa la expresión e).

Así la regla anterior se lee, partiendo del entorno Ω, la expresión e se evalúa al valor final v.

$$\frac{}{\Omega \vdash \text{ConstI } n \rightarrowtriangle \text{NumI } n}\ \text{Constante entera}$$

$$\frac{}{\Omega \vdash \text{ConstbI } b \rightarrowtriangle \text{BoolI } b}\ \text{Constante booleana}$$

[9] Nótese que hemos utilizado "$\rightarrowtriangle$" para denotar la semántica natural con índices de De Bruijn. Esto con el fin de diferenciarla de la semántica natural basda en nombres (para la cual utilizamos "$\Rightarrow$") y no haya lugar a confusión.

$$\frac{\Omega \vdash e_1 \twoheadrightarrow \text{NumI } n_1 \qquad \Omega \vdash e_2 \twoheadrightarrow \text{NumI } n_2}{\Omega \vdash \text{PlusI } e_1\ e_2 \twoheadrightarrow \text{NumI } (n_1 + n_2)}\ \text{Adición}$$

$$\frac{\Omega \vdash e_1 \twoheadrightarrow \text{NumI } n_1 \qquad \Omega \vdash e_2 \twoheadrightarrow \text{NumI } n_2}{\Omega \vdash \text{MinusI } e_1\ e_2 \twoheadrightarrow \text{NumI } (n_1 - n_2)}\ \text{Sustracción (con } (n_1 - n_2) \geq 0)$$

$$\frac{\Omega \vdash e_1 \twoheadrightarrow \text{NumI } n_1 \qquad \Omega \vdash e_2 \twoheadrightarrow \text{NumI } n_2}{\Omega \vdash \text{TimesI } e_1\ e_2 \twoheadrightarrow \text{NumI } (n_1 * n_2)}\ \text{Producto}$$

$$\frac{\Omega \vdash e_1 \twoheadrightarrow \text{NumI } n_1 \qquad \Omega \vdash e_2 \twoheadrightarrow \text{NumI } n_2}{\Omega \vdash \text{EqI } e_1\ e_2 \twoheadrightarrow \text{BoolI } n_1 = n_2}\ \text{Comparación de igualdad}$$

$$\frac{}{[v_0, \ldots, v_i, \ldots, v_n] \vdash \text{VarI } i \twoheadrightarrow v_i}\ \text{Variable}$$

$$\frac{\Omega \vdash e_1 \twoheadrightarrow v_1 \qquad [v_1] \cdot \Omega \vdash e_2 \twoheadrightarrow v}{\Omega \vdash \text{LetI } e_1\ e_2 \twoheadrightarrow v}\ \text{Let}$$

$$\frac{\Omega \vdash e_1 \twoheadrightarrow \text{BoolI } true \qquad \Omega \vdash e_2 \twoheadrightarrow v}{\Omega \vdash \text{IfI } e_1\ e_2\ e_3 \twoheadrightarrow v}\ \text{If (true)}$$

$$\frac{\Omega \vdash e_1 \twoheadrightarrow \text{BoolI } false \qquad \Omega \vdash e_3 \twoheadrightarrow v}{\Omega \vdash \text{IfI } e_1\ e_2\ e_3 \twoheadrightarrow v}\ \text{If (false)}$$

$$\frac{}{\Omega \vdash \text{LamI } e \twoheadrightarrow \text{ClosI } e\ \Omega}\ \text{Abstracción}$$

$$\frac{}{\text{MuI } e \twoheadrightarrow \text{ClosI}_{rec}\ e\ \Omega}\ \text{Punto fijo}$$

$$\frac{\Omega \vdash e_1 \twoheadrightarrow \text{ClosI } c_1\ \Omega_1 \qquad \Omega \vdash e_2 \twoheadrightarrow v_2 \qquad [v_2] \cdot \Omega_1 \vdash c_1 \twoheadrightarrow v}{\Omega \vdash \text{AppI } e_1\ e_2 \twoheadrightarrow v}\ \text{Aplicación}$$

$$\frac{\Omega \vdash e_1 \twoheadrightarrow \text{ClosI}_{rec}\ c_1\ \Omega_1 \qquad \Omega \vdash e_2 \twoheadrightarrow v_2 \qquad [v_2] \cdot [\text{Clos}_{rec}\ c_1\ \Omega_1] \cdot \Omega_1 \vdash c_1 \twoheadrightarrow v}{\Omega \vdash \text{AppI } e_1\ e_2 \twoheadrightarrow v}$$

y su correspondiente desarrollo en Coq es:

```
(**
 * Función que dado un entorno basado en índices de De Bruijn
 * y dado un índice devuelve el valor en la posición señalada
 * por el índice en el entorno. Si es un índice inválido
 * regresa None.
 *
 * Sirve tanto para los entornos O de la semántica del lenguaje
 * basada en notación de De Bruijn como para los entornos D
 * de las semánticas de la máquina SECD moderna.
 *
 **)

Fixpoint accessi {A:Type} (e: list A) (n: nat) : option A :=
match e with
| nil => None
| x::xs => match n with
                | O => Some x
                | S m => accessi xs m
                end
end.

(**
 * Semántica de paso grande del lenguaje fuente con
 * índices de De Bruijn.
 *
 **)

Inductive BSSMMLDB : MMLDBenv -> MMLDBexp -> MMLDBval -> Prop :=
| BSSDBConst: forall O:MMLDBenv, forall i:nat,
             BSSMMLDB O (ConstI i) (NumI i)

| BSSDBPlus: forall O:MMLDBenv, forall e1 e2:MMLDBexp, forall n1 n2:nat,
            BSSMMLDB O e1 (NumI n1) ->
            BSSMMLDB O e2 (NumI n2) ->
            BSSMMLDB O (PlusI e1 e2) (NumI (n1+n2))

| BSSDBVar: forall O:MMLDBenv, forall n:id, forall v:MMLDBval,
           accessi O n = Some v -> BSSMMLDB O (VarI n) v

| BSSDBLam: forall O:MMLDBenv, forall e:MMLDBexp,
           BSSMMLDB O (LamI e) (ClosI e O)

| BSSDBMu: forall O:MMLDBenv, forall e:MMLDBexp,
    BSSMMLDB O (MuI e) (ClosrI e O)

| BSSDBApp: forall O O1: MMLDBenv, forall e1 e2 c1:MMLDBexp,
         forall v v2:MMLDBval,
           BSSMMLDB O e1 (ClosI c1 O1) ->
           BSSMMLDB O e2 v2 ->
```

```
           BSSMMLDB (v2::O1) c1 v ->
           BSSMMLDB O (AppI e1 e2) v

| BSSDBAppr: forall O O1:MMLDBenv, forall e1 e2 c1:MMLDBexp,
            forall v v2:MMLDBval,
    BSSMMLDB O e1 (ClosrI c1 O1) ->
    BSSMMLDB O e2 v2 ->
    BSSMMLDB (v2::(ClosrI c1 O1)::O1) c1 v ->
    BSSMMLDB O (AppI e1 e2) v.
```

Podemos notar que el nombre que le dimos a esta semántica en Coq es `BSSMMLDB`. Por lo que se tiene que: `BSSMMLDB O e v` si y sólo si $O \vdash e \twoheadrightarrow v$.

Aquí también podemos utilizar verificación y así lo haremos . Por lo que damos el programa, es decir, el intérprete que cumplirá con la especificación, que en este caso es la semántica natural con índices de De Bruijn a la que llamamos en Coq `BSSMMLDB`.

```
(**
 * Intérprete del lenguaje fuente con notación de De Bruijn.
 * Este intérprete cumple con la especificación de la semántica
 * de paso grande con notación de De Bruijn del lenguaje fuente.
 *
 **)
Fixpoint evali (depthR:nat )(O:MMLDBenv) (e:MMLDBexp)
: option MMLDBval :=
match depthR with
| 0 => None
| S n => match e with
           | ConstI i => Some (NumI i)
           | PlusI e1 e2 =>
             match (evali n O e1) with
               | Some (NumI n1) => match (evali n O e2) with
                                     | Some (NumI n2) => Some (NumI (n1+n2))
                                     | _ => None
                                   end
               | _ => None
             end
           | VarI x => accessi O x
           | LamI elam => Some (ClosI elam O)
           | MuI emu => Some (ClosrI emu O)
           | AppI e1 e2 =>
             match (evali n O e1) with
               | Some (ClosI c1 O1) =>
                 match evali n O e2 with
                   | Some v2 => evali n (v2::O1) c1
                   | _ => None
                 end
```

```
                | Some (ClosrI c1 O1) =>
                  match evali n O e2 with
                    | Some v2 => evali n (v2::(ClosrI c1 O1)::O1) c1
                    | _ => None
                  end
                | _ => None
              end
          end
end.
```

Recordemos que tal como vimos en la sección 3.7 el parámetro `depthR` corresponde a la profundidad de la recursión ya que se está haciendo uso de recursión acotada. Esto debido a que como sabemos todo cálculo en Coq debe terminar.[10]

Ahora damos el lema que establece que en efecto este intérprete cumple con la especificación, esto es en este caso cumple con la semántica con notación de De Bruijn.

```
(**
 * Lema que enuncia que el intérprete "evali" cumple con la especificación
 * de la semántica de paso grande con notación de De Bruijn del lenguaje
 * fuente.
 *
 **)
Lemma evalic:
forall O:MMLDBenv, forall e:MMLDBexp, forall v:MMLDBval,
BSSMMLDB O e v ->
exists n:nat, (evali n O e) = Some v.
Proof.
```

Con esto hemos obtenido un intérprete de nuestro lenguaje haciendo uso de la notación de De Bruijn.

Podemos ahora utilizarlo para calcular nuestro ejemplo motivacional escrito utilizando índices de De Bruijn:

```
(*Ejemplo de cálculo intérprete evali*)
Compute (evali 19 nil
(AppI (MuI (IfI (EqI (VarI 0) (ConstI 0))
             (ConstI 1)
             (TimesI (VarI 0)
                 (AppI (VarI 1) (MinusI (VarI 0) (ConstI 1)))))) (ConstI 5))
).
```

[10]Los detalles acerca de este tema los estudiamos en la sección 3.7.

```
     = Some (NumI 120)
     : option MMLDBval
```

y vemos como en efecto recibimos el resultado esperado, es decir, 120. También aquí podemos notar cómo es mucho más engorroso escribir un programa utilizando directamente índices de De Bruijn y no es viable pedirle al programador que escriba directamente un programa utilizando esta notación, lo que quisiéramos es que nosotros escribiésemos nuestro programa con nombres como lo hicimos en la sección 5.2.2 y la computadora lo tradujera al que acabamos de presentar porque el tiempo de ejecución de este intérprete (`evali`) es menor que el de `eval`.

Adicionalmente podemos utilizar el mecanismo de extracción y obtener así un intérprete verificado de nuestro lenguaje que utiliza notación de De Bruijn. Dicho intérprete obtenido estará escrito en el lenguaje de programación OCaml y se encontrará listo para ser utilizado en la vida real. A continuación mostramos cómo obtenerlo.

```
Extraction evali.

(** val evali : nat -> mMLDBenv -> mMLDBexp -> mMLDBval option **)

let rec evali depthR o e =
  match depthR with
  | O -> None
  | S n ->
    (match e with
     | ConstI i -> Some (NumI i)
     | PlusI (e1, e2) ->
       (match evali n o e1 with
        | Some m ->
          (match m with
           | NumI n1 ->
             (match evali n o e2 with
              | Some m0 ->
                (match m0 with
                 | NumI n2 -> Some (NumI (plus n1 n2))
                 | _ -> None)
              | None -> None)
           | _ -> None)
        | None -> None)
     | VarI x -> accessi o x
     | LamI elam -> Some (ClosI (elam, o))
     | MuI emu -> Some (ClosrI (emu, o))
     | AppI (e1, e2) ->
       (match evali n o e1 with
        | Some m ->
          (match m with
```

```
          | ClosI (c1, o1) ->
            (match evali n o e2 with
             | Some v2 -> evali n (Cons (v2, o1)) c1
             | None -> None)
          | ClosrI (c1, o1) ->
            (match evali n o e2 with
             | Some v2 ->
               evali n (Cons (v2, (Cons ((ClosrI (c1, o1)), o1)))) c1
             | None -> None)
          | _ -> None)
        | None -> None))
```

En la siguiente sección presentaremos la traducción de nuestro lenguaje fuente con notación basada en nombres a índices de De Bruijn.

5.4. Traducción a índices de De Bruijn

Antes de realizar la traducción a índices de De Bruijn revisemos primero la forma de las reglas para la semántica con nombres:

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

y la de la semántica con índices de De Bruijn:

$$\Omega \vdash e \rightarrowtriangle v$$

sabemos que un entorno Γ almacena pares (nombre,valor), es decir, es de la forma $[(x_0, v_0), \ldots, (x_n, v_n)]$ mientras que un entorno Ω almacena únicamente valores, es decir, es de la forma $[v_0, \ldots, v_n]$. Entonces dado un entorno Γ que se utilice en la semántica con nombres debe haber uno equivalente Ω que se utilice en la semántica con índices de De Bruijn. Lo mismo para los valores, debe haber una equivalencia entre los valores que se utilizan en la semántica con nombres y los que se utilizan en la semántica con índices. Esta equivalencia la daremos un poco más adelante en la sección 5.4.1.

Ahora, recordemos que para especificar una traducción (compilación) podemos hacer uso de la semántica natural, en particular podemos utilizarla para especificar la traducción a índices de De Bruijn, que es justo lo que hicimos en la sección 4.1.2 donde teníamos reglas de la forma:

$$\Pi \vdash e \rightarrow t$$

donde Π es un contexto que contiene el nombre de las variables libres de la expresión e, o sea Π almacena únicamente nombres, es decir, es de la forma $[x_0, \ldots, x_n]$ que es lo único necesario para llevar a cabo la traducción (los valores resultan ser irrelevantes). Luego e

es una expresión (con nombres) y t un término[11] en notación de De Bruijn. Así el juicio se lee la expresión e con variables libres en Π se traduce al término t.

Nótese como los entornos Γ y Ω y el contexto Π son todos distintos.

Dimos ya en la sección 4.1.2 una explicación de cómo se lleva a cabo esta traducción. En ella también presentamos las reglas de traducción (utilizando semántica natural) de variables, abstracciones, aplicaciones y el operador de adición. Adicionalmente ofrecimos algunos ejemplos de cómo se realiza esta traducción.

Presentaremos ahora en el mismo sentido que en la sección 4.1.2 la semántica natural correspondiente a la traducción a índices de De Bruijn (esto es, las reglas de traducción) para los enunciados restantes que conforman nuestro lenguaje fuente.

$$\frac{}{\Pi \vdash \mathsf{Const}\ n \rightarrow\ \mathsf{ConstI}\ n}$$

$$\frac{}{\Pi \vdash \mathsf{Constb}\ b \rightarrow\ \mathsf{ConstbI}\ b}$$

$$\frac{\Pi \vdash e_1 \rightarrow t_1 \qquad \Pi \vdash e_2 \rightarrow t_2}{\Pi \vdash \mathsf{Minus}\ e_1\ e_2 \rightarrow\ \mathsf{MinusI}\ t_1\ t_2}$$

$$\frac{\Pi \vdash e_1 \rightarrow t_1 \qquad \Pi \vdash e_2 \rightarrow t_2}{\Pi \vdash \mathsf{Times}\ e_1\ e_2 \rightarrow\ \mathsf{TimesI}\ t_1\ t_2}$$

$$\frac{\Pi \vdash e_1 \rightarrow t_1 \qquad \Pi \vdash e_2 \rightarrow t_2}{\Pi \vdash \mathsf{Eq}\ e_1\ e_2 \rightarrow\ \mathsf{EqI}\ t_1\ t_2}$$

$$\frac{\Pi \vdash e_1 \rightarrow t_1 \qquad \Pi \vdash e_2 \rightarrow t_2 \qquad \Pi \vdash e_3 \rightarrow t_3}{\Pi \vdash \mathsf{If}\ e_1\ e_2\ e_3 \rightarrow\ \mathsf{IfI}\ t_1\ t_2\ t_3}$$

$$\frac{[x, f] \cdot \Pi \vdash e \rightarrow t}{\Pi \vdash \mathsf{Mu}\ f\ x\ e\ \rightarrow\ \mathsf{MuI}\ t}$$

Nótese cómo aquí nos ha resultado intuitivo utilizar la semántica natural para especificar la traducción a índices.

Presentamos ahora su correspondiente desarrollo en Coq.

[11] Aquí por conveniencia utilizamos expresión para referirnos a un programa escrito utilizando nombres para los identificadores y término para un programa que utiliza la notación de De Bruijn.

```
(*Contextos P de la traducción de nombres a índices de De Bruijn*)
Definition ctx:= list string.

(*Traducción de nombres a índices de De Bruijn*)
Inductive NtoDB: ctx -> MMLexp -> MMLDBexp -> Prop :=
| CNtoDB: forall P n, NtoDB P (Const n) (ConstI n)

| VeNtoDB: forall x P, NtoDB (x::P) (Var x) (VarI 0)
| VneNtoDB: forall x y P n,
    x <> y ->
    NtoDB P (Var x) (VarI n) ->
    NtoDB (y::P) (Var x) (VarI (S n))

| LetNtoDB: forall P e1 e2 e1i e2i x,
    NtoDB P e1 e1i ->
    NtoDB (x::P) e2 e2i ->
    NtoDB P (Letm x e1 e2) (LetmI e1i e2i)

| LamNtoDB: forall x e ei P,
    NtoDB (x::P) e ei ->
    NtoDB P (Lam x e) (LamI ei)

| MuNtoDB: forall x f P e ei,
    NtoDB (x::f::P) e ei ->
    NtoDB P (Mu f x e) (MuI ei)

| AppNtoDB: forall P e1 e2 e1i e2i,
    NtoDB P e1 e1i ->
    NtoDB P e2 e2i ->
    NtoDB P (App e1 e2) (AppI e1i e2i).
```

Como podemos observar hemos llamado en Coq `NtoDB` a la traducción a índices de De Bruijn, por lo que se tiene que: `NtoDB P e t` si y sólo si $P \vdash e \rightarrow t$.

Como hemos utilizado la semántica natural para especificar la traducción a índices de De Bruijn, esta traducción se expresa como una relación definida inductivamente, que en Coq como acabamos de presentar se escribe mediante una definición inductiva. Ahora por otro lado, sabemos que en este caso dicha relación es determinista, es decir, nosotros conjeturamos que la traducción a índices de De Bruijn es determinista pero debemos demostrarlo. Por lo que a continuación presentamos un lema que enuncia el determinismo de esta traducción (su demostración como sabemos podemos consultarla en [Zúñ15b]).

```
(**
 * Lema que enuncia que la traducción de nombres a índices de De Bruijn
 * es determinista.
 *
```

```
 **)
Lemma NtoDBDet:
forall e P ei1 ei2,
NtoDB P e ei1 ->
NtoDB P e ei2 ->
ei1 = ei2.
```

Ahora realizamos verificación para obtener un programa que calcule dicha traducción, es decir, un compilador de nuestro lenguaje con notación basada en nombres a índices de De Bruijn. Entonces debemos ofrecer este programa y luego verificar que cumple con la especificación,[12] que en este caso es la traducción a índices de De Bruijn expresada por medio de semántica natural que acabamos de presentar.

Comenzamos presentando nuestro programa, es decir nuestro compilador.

```
(**
 * Compilador que cumple con la especificación de
 * la traducción de nombres a índices de De Bruijn.
 * Versión que sigue la semántica sin necesidad de
 * recursión acotada (Bounded Recursion).
 *
 **)
Fixpoint NtoDBfwoBR (P:ctx) (e: MMLexp)
: option MMLDBexp :=
match e with
| Const n => Some (ConstI n)
| Var x =>
  let fix vari (Ph:ctx) (exk:MMLexp) :=
      match Ph with
        | y::Ps =>
          match exk with
            | Var z => if string_dec y z then Some (VarI 0)
                       else
                         match (vari Ps (Var z)) with
                           | Some (VarI n) => Some (VarI (S n))
                           | _ => None
                         end
            | _ => None
          end
        | nil => None
      end
```

[12]En realidad podemos dar diferentes programas que cumplan con la especificación, esto con diferentes fines. Por ejemplo uno puede ser más claro y otro más eficiente y se puede utilizar el que más convenga para cierto fin, en este caso particular nosotros desarrollamos diferentes programas que cumplen la especificación, es decir que calculan la traducción a índices. Para visualizarlos todos, consúltese el desarrollo completo de este trabajo en [Zúñ15b].

```
  in vari P (Var x)

| Letm x e1 e2 =>
  match NtoDBfwoBR P e1 with
  | Some e1i =>
    match NtoDBfwoBR (x::P) e2 with
    | Some e2i => Some (LetmI e1i e2i)
    | _ => None
    end
  | _ => None
  end
| Lam x e =>
  match NtoDBfwoBR (x::P) e with
  | Some ei => Some (LamI ei)
  | _ => None
  end
| Mu f x e =>
  match NtoDBfwoBR (x::f::P) e with
  | Some ei => Some (MuI ei)
  | _ => None
  end
end.
```

y ahora demostramos que en efecto cumple con la especificación, es decir, con la traducción a índices de De Bruijn.

```
(**
 * Lema que enuncia que el compilador "NtoDBfwoBR" cumple con la
 * especificación de la traducción de nombres a índices de De Bruijn.
 *
 **)
Lemma NtoDBfwoBRc:
forall P e ei,
NtoDB P e ei ->
NtoDBfwoBR P e = Some ei.
```

Con esto podemos calcular la traducción a índices de De Bruijn de nuestro ejemplo motivacional:

```
(*Ejemplo de cálculo compilador NtoDBfwoBR*)
Compute (NtoDBfwoBR nil
(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))
).
     = Some
         (AppI
            (MuI
               (IfI (EqI (VarI 0) (ConstI 0)) (ConstI 1)
                  (TimesI (VarI 0)
                     (AppI (VarI 1) (MinusI (VarI 0) (ConstI 1))))))
            (ConstI 5))
     : option MMLDBexp
```

que es justo lo que esperábamos. Más aún ahora podemos realizar lo que queríamos, es decir, calcular la traducción y luego evaluarla en el intérprete más eficiente `evali`, como se muestra a continuación.

```
(**
 * Ahora podemos calcular la traducción a índices de De Bruijn
 * y luego evaluar la expresión que se obtiene como resultado
 * en el intérprete basado en índices evali.
 *
 **)
Definition tradaiyeval (e:MMLexp) :=
match NtoDBfwoBR nil e with
| Some ei => evali 19 nil ei
| None => None
end.

Compute (tradaiyeval
(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))).
     = Some (NumI 120)
     : option MMLDBval
```

Por otro lado, también podemos utilizar el mecanismo de extracción para obtener el compilador verificado en OCaml. Lo que se ilustra a continuación.

```
Extraction NtoDBfwoBR.
```

```
(** val ntoDBfwoBR : ctx -> mMLexp -> mMLDBexp option **)

let rec ntoDBfwoBR p = function
| Const n -> Some (ConstI n)
| Var x ->
  let rec vari ph exk =
    match ph with
    | Nil -> None
    | Cons (y, ps) ->
      (match exk with
       | Const n -> None
       | Plus (m, m0) -> None
       | Minus (m, m0) -> None
       | Times (m, m0) -> None
       | Eq (m, m0) -> None
       | Constb b -> None
       | Var z ->
         (match string_dec y z with
          | Left -> Some (VarI O)
          | Right ->
            (match vari ps (Var z) with
             | Some m ->
               (match m with
                | VarI n -> Some (VarI (S n))
                | _ -> None)
             | None -> None))
       | _ -> None)
  in vari p (Var x)
| Letm (x, e1, e2) ->
  (match ntoDBfwoBR p e1 with
   | Some e1i ->
     (match ntoDBfwoBR (Cons (x, p)) e2 with
      | Some e2i -> Some (LetmI (e1i, e2i))
      | None -> None)
   | None -> None)
| Lam (x, e0) ->
  (match ntoDBfwoBR (Cons (x, p)) e0 with
   | Some ei -> Some (LamI ei)
   | None -> None)
| Mu (f, x, e0) ->
  (match ntoDBfwoBR (Cons (x, (Cons (f, p)))) e0 with
   | Some ei -> Some (MuI ei)
   | None -> None)
```

5.4.1. Equivalencia entre valores y entornos

Analicemos ahora la equivalencia entre valores y entornos de las dos semánticas (con nombres y con índices) de la que hablamos arriba.

Recordemos que un entorno Γ de la semántica basada en nombres, se almacenan las variables junto con sus respectivos valores, así al evaluar una variable x, se busca x en Γ y se obtiene su valor. Por su parte en un entorno Ω de la semántica basada en índices, únicamente se almacenan los valores de las variables (pero no su nombre), ya que esta vez la variable estará representada por un índice que corresponde a la posición de su valor en el entorno (por lo que no hace falta hacer una búsqueda en éste), así para obtener el valor de una variable x representada por el índice de De Bruijn i simplemente se accede a la posición i de Ω.

Con base en esto podemos decir que para un entorno $\Gamma = [(x_0, v_0), \ldots, (x_n, v_n)]$ de la semántica con nombres el correspondiente entorno Ω de la semántica con índices es simplemente: $[v_0, \ldots, v_n]$. Es decir, tomar simplemente los valores del entorno Γ (en el mismo orden) descartando los nombres de las variables. Lo anterior formalmente significa realizar una operación proyección sobre la segunda entrada del entorno Γ definida de la siguiente manera:

$$\prod_2([(x_0, v_0), \ldots, (x_n, v_n)]) = map\ \pi_2([(x_0, v_0), \ldots, (x_n, v_n)])$$

y sabemos que:

$$map\ \pi_2([(x_0, v_0), \ldots, (x_n, v_n)]) = [v_0, \ldots, v_n]$$

Por otra parte, analicemos de nuevo la traducción a índices del programa:

$$\text{Let } x \text{ (Const } 5)(\text{Let } y \text{ (Const } 2)(\text{Plus (Var } x)(\text{Var } y))$$

que presentamos en la sección 4.1.2, que es la siguiente:

$$\dfrac{\dfrac{}{[] \vdash 5 \to 5} \quad \dfrac{\dfrac{}{[x] \vdash 2 \to 2} \quad \dfrac{\dfrac{\dfrac{}{[x] \vdash \text{Var } x \to \text{VarI } 0}\,1}{[y, x] \vdash \text{Var } x \to \text{VarI } 1}\,2 \quad \dfrac{}{[y, x] \vdash \text{Var } y \to \text{VarI } 0}\,1}{[y, x] \vdash \text{Plus (Var } x) \text{ (Var } y) \to \text{PlusI (VarI } 1) \text{ (VarI } 0)}\,5}{[x] \vdash \text{Let } y\ 2 \text{ Plus (Var } x) \text{ (Var } y) \to \text{LetI } 2 \text{ (PlusI (VarI } 1) \text{ (VarI } 0))}\,6}{[] \vdash \text{Let } x\ 5 \text{ (Let } y\ 2 \text{ Plus (Var } x) \text{ (Var } y)) \to \text{LetI } 5 \text{ (LetI } 2 \text{ (PlusI (VarI } 1) \text{ (VarI } 0)))}\,6}$$

Recordemos que la traducción a índices utiliza un contexto en el que únicamente son necesarios los nombres. Entonces con un tratamiento análogo a nuestra discusión anterior, podemos decir que para un entorno $\Gamma = [(x_0, v_0), \ldots, (x_n, v_n)]$ de la semántica con nombres el correspondiente contexto Π de la traducción a índices es simplemente: $[x_0, \ldots, x_n]$, es decir, tomamos simplemente los nombres de las variables del entorno Γ (en el mismo orden) descartando los valores correspondientes a éstas. Formalmente es realizar una operación proyección sobre la primera entrada del entorno Γ definida de la siguiente manera:

$$\prod_1([(x_0, v_0), \ldots, (x_n, v_n)]) = map\ \pi_1([(x_0, v_0), \ldots, (x_n, v_n)])$$

y sabemos que:

$$map\ \pi_1([(x_0, v_0), \ldots, (x_n, v_n)]) = [x_0, \ldots, x_n]$$

Ahora podemos utilizar la semántica natural para especificar la traducción (equivalencia) de valores con nombres a valores con índices (para la cual utilizamos "$\twoheadrightarrow$" para denotarla) y de entornos con nombres a entornos con índices (para la que utilizamos "$\hookrightarrow$" como notación) con lo cual tenemos lo siguiente:

$$\frac{}{\text{Bool } b \twoheadrightarrow \text{BoolI } b}$$

$$\frac{}{\text{Num } n \twoheadrightarrow \text{NumI } n}$$

$$\frac{\Gamma \hookrightarrow \Omega \qquad [x] \cdot \prod_1(\Gamma) \vdash c \rightarrow t}{\text{Clos } x\ c\ \Gamma \twoheadrightarrow \text{ClosI } t\ \Omega}$$

$$\frac{\Gamma \hookrightarrow \Omega \qquad [x, f] \cdot \prod_1(\Gamma) \vdash c \rightarrow t}{\text{Clos}_{rec}\ f\ x\ c\ \Gamma \twoheadrightarrow \text{ClosI}_{rec}\ t\ \Omega}$$

$$\frac{}{[] \hookrightarrow []}$$

$$\frac{v \twoheadrightarrow v_t \qquad \Gamma \hookrightarrow \Omega}{[(x, v)] \cdot \Gamma \hookrightarrow [v_t] \cdot \Omega}$$

Podemos notar que la traducción de valores con nombres a valores con índices es mutuamente dependiente con la traducción de entornos con nombres a entornos con índices.

La explicación de las reglas es la siguiente, es claro que las constantes tanto las naturales como las booleanas permanecen igual. Luego, para traducir una cerradura con nombres a una con índices, es necesario por una parte traducir el entorno de la cerradura con nombres Γ al correspondiente con índices Ω (como lo explicamos más arriba). Por otra parte es necesario traducir la expresión de la cerradura con nombres[13] c a su correspondiente término t utilizando índices. Para llevar a cabo la traducción de c a t es necesario que el contexto de la traducción sea el correspondiente con base en el entorno con nombres Γ (según lo explicamos arriba), es decir $\prod_1(\Gamma)$. Recordemos además que cuando en la semántica con nombres se evalúa una aplicación sobre una cerradura si en la expresión c de la cerradura se hace referencia a la variable x de la cerradura, ésta se evaluará al

[13]Utilizamos c (en lugar de e) para denotar esta expresión porque como está en una cerradura por lo general se hace referencia a esta como código pero en realidad es una expresión.

valor que se le pasa como argumento al realizar la aplicación, por ejemplo, si tenemos la aplicación $(\lambda x.x)5$ y la evaluamos inicialmente nuestro entorno es vacío $[]$, luego tendremos la cerradura $(x, x, [])$, después 5 se evalúa a 5 y posteriormente tenemos el entorno $[(x, 5)]$ por lo que la evaluar x obtenemos 5. Entonces, para el caso de la notación de De Bruijn el argumento será el último valor que se metió al entorno Ω, así en el término t (correspondiente a la notación de De Bruijn de la expresión c) a x le corresponde el índice $\bar{0}$, por eso al realizar la traducción se debe meter x al contexto de ésta, es decir, el contexto al realizar la traducción debe ser $[x] \cdot \prod_1(\Gamma)$. El caso para la traducción de las cerraduras recursivas es similar, sólo que esta vez no basta con meter la variable x de la cerradura al contexto de la traducción. Como intuitivamente esta vez se trata de una función recursiva, en el cuerpo c también se hará referencia al nombre de la función f (el valor de esta f al evaluarse en lugar de ser un argumento como en el caso de x, será la misma cerradura recursiva),[14] por lo que se debe meter f y x al contexto. De esta manera el contexto debe ser $[x, f] \cdot \prod_1(\Gamma)$.

Por otra parte para traducir un entorno con nombres Γ a uno con índices Ω, tomamos inductivamente el valor v del par (x, v) más a la izquierda, lo traducimos y lo metemos en Ω (en el mismo orden). Aquí podemos ver claramente cómo descartamos los nombres de las variables en este caso x. Lo anterior lo realizamos hasta llegar al caso básico, es decir, el entorno vacío $[]$ que se traduce a sí mismo.

Ahora el desarrollo correspondiente a esta traducción en Coq es el siguiente.

```
(**
 * Función que dado un entorno G de la semántica basada
 * en nombres del lenguaje toma la primera entrada
 * de cada uno de los pares que lo conforman en el mismo
 * orden (que corresponde a los nombres de las variables)
 * y los devuelve como un contexto P. P es un contexto de
 * la traducción de nombres a índices de De Bruijn.
 **)
Fixpoint envfe(G:MMLenv) : ctx :=
match G with
| (x,v)::Gs => x :: (envfe Gs)
| nil => nil
end.

(**
 * Traducción de valores y entornos con nombres
 * a valores y entornos con índices de De Bruijn.
 *
 **)
Inductive VNtoDB: MMLval -> MMLDBval -> Prop :=
| Cbvt: forall b, VNtoDB (Bool b) (BoolI b)
| Cvt: forall n, VNtoDB (Num n) (NumI n)
```

[14] También podemos pensar esto como si la función recursiva se pasara a sí misma como argumento.

```
| Clvt: forall G O c ci x,
    ENtoDB G O ->
    NtoDB (x::(envfe G)) c ci ->
    VNtoDB (Clos x c G) (ClosI ci O)
| Clrvt:forall G O x f c ci,
    ENtoDB G O ->
    NtoDB (x::f::(envfe G)) c ci ->
    VNtoDB (Closr f x c G) (ClosrI ci O)
with ENtoDB : MMLenv -> MMLDBenv -> Prop :=
| Net : ENtoDB nil nil
| Nnet: forall x v vi G O,
    VNtoDB v vi ->
    ENtoDB G O ->
    ENtoDB ((x,v)::G) (vi::O).
```

Podemos ver que el nombre que le dimos a la traducción de valores con nombres a valores con índices en Coq es `VNtoDB`, mientras que el nombre para la traducción de entornos con nombres a entornos con índices es `ENtoDB`. Por lo que tenemos por una parte que `VNtoDB v vi` si y sólo si $v \twoheadrightarrow v_i$ y por otra que `ENtoDB G O` si y sólo si $G \hookrightarrow O$.

Una vez más podemos realizar verificación. Por lo que damos el programa, es decir, el compilador de valores con nombres a valores con índices y de entornos con nombres a entornos con índices.

```
(**
 * Compilador que cumple la especificación de la traducción
 * de valores y entornos con nombres a valores y entornos
 * con índices de De Bruijn.
 *
 **)
Fixpoint VNtoDBf (depthR:nat) (v:MMLval) : option MMLDBval :=
match depthR with
| O => None
| S m =>
match v with
| Bool b => Some (BoolI b)
| Num n => Some (NumI n)
| Clos x c G =>
  match NtoDBfwoBR (x::(envfe G)) c with
    | Some ci => match ENtoDBf m G with
                   | Some Om => Some (ClosI ci Om)
                   | None => None
                 end
    | None => None
  end
| Closr f x c G =>
  match NtoDBfwoBR (x::f::(envfe G)) c with
```

```
    | Some ci => match ENtoDBf m G with
                     | Some Om => Some (ClosrI ci Om)
                     | None => None
                 end
    | None => None
  end
end
end
with ENtoDBf (depthR:nat) (G:MMLenv) : option MMLDBenv :=
match depthR with
| O => None
| S m =>
match G with
| (x,v)::Gs => match VNtoDBf m v with
                     | Some vi =>
                       match ENtoDBf m Gs with
                         | Some Os => Some (vi::Os)
                         | None => None
                       end
                     | None => None
                 end
| nil => Some nil
end
end.
```

Y ahora demostramos que en efecto cumple con la especificación que en este caso es la traducción de valores con nombres a índices y de entornos con nombres a entornos con índices.

```
(**
 * Lema que enuncia que los compiladores "VNtoDBf" y "ENtoDBf" cumplen
 * respectivamente con la especificación de la traducción de valores con
 * nombres a valores con índices y con la especificación de la traducción de
 * entornos con nombres a entornos con índices de De Bruijn.
 *
 **)
Lemma VNtoDBfc:
forall v vi,
VNtoDB v vi ->
exists n, VNtoDBf n v = Some vi
with ENtoDBfc:
forall G O,
ENtoDB G O ->
exists n, ENtoDBf n G = Some O.
```

Nótese el uso de definiciones inductivas, funciones y lemas mutuamente dependientes

en Coq dados en los desarrollos anteriores.

Ahora podemos utilizar nuestros compiladores de valores y entornos con nombres a índices para calcular algunos ejemplos.

```
(*Ejemplos de cálculo compiladores VNtoDBf y ENtoDBf*)
Compute (VNtoDBf 1 (Num 10)).
     = Some (NumI 10)
     : option MMLDBval

Compute (ENtoDBf 4 (("x",Num 5)::("y",Num 6)::("z",Num 7)::nil)).
     = Some (NumI 5 :: NumI 6 :: NumI 7 :: nil)
     : option MMLDBenv

Compute(VNtoDBf 4 ((Clos "x" (Plus (Const 5) (Const 7))
                           (("y",Num 4)::("z",Bool true)::nil)))).
     = Some
         (ClosI (PlusI (ConstI 5) (ConstI 7)) (NumI 4 :: BoolI true :: nil))
     : option MMLDBval
```

Por otro lado podemos utilizar el mecanismo de extracción y obtener así un compilador verificado de valores con nombres a valores índices y de entornos con nombres a entornos con índices, listo para ser utilizado en la vida real.

```
Extraction VNtoDBf.

(** val vNtoDBf : nat -> mMLval -> mMLDBval option **)

let rec vNtoDBf depthR v =
  match depthR with
  | O -> None
  | S m ->
    (match v with
     | Bool b -> Some (BoolI b)
     | Num n -> Some (NumI n)
     | Clos (x, c, g) ->
       (match ntoDBfwoBR (Cons (x, (envfe g))) c with
        | Some ci ->
          (match eNtoDBf m g with
           | Some om -> Some (ClosI (ci, om))
           | None -> None)
        | None -> None)
     | Closr (f, x, c, g) ->
       (match ntoDBfwoBR (Cons (x, (Cons (f, (envfe g))))) c with
        | Some ci ->
          (match eNtoDBf m g with
```

```
            | Some om -> Some (ClosrI (ci, om))
            | None -> None)
         | None -> None))

(** val eNtoDBf : nat -> mMLenv -> mMLDBenv option **)

and eNtoDBf depthR g =
  match depthR with
  | O -> None
  | S m ->
    (match g with
     | Nil -> Some Nil
     | Cons (p, gs) ->
       let Pair (x, v) = p in
       (match vNtoDBf m v with
        | Some vi ->
          (match eNtoDBf m gs with
           | Some os -> Some (Cons (vi, os))
           | None -> None)
        | None -> None))
```

Habiendo desarrollado todo lo anterior, debemos ahora enunciar y demostrar la corrección de la traducción de nombres a índices de De Bruijn. Lo cual haremos en la siguiente sección.

5.4.2. Corrección

La corrección de la traducción a índices de De Bruijn se formula entonces como sigue. En general si tenemos una expresión con nombres e que se evalúa en un entorno Γ a un valor v (en notación $\Gamma \vdash e \Rightarrow v$), entonces esperaríamos que exista un valor v_i que use índices y que corresponda a la traducción del valor v (en notación $v \twoheadrightarrow v_i$) de tal manera que si evaluamos el término t con notación de De Bruijn en el entorno Ω el resultado sea v_i (en notación $\Omega \vdash t \rightarrow v_i$), esto en el supuesto que por un lado Ω fuese el entorno que use índices de De Bruijn correspondiente a Γ (en notación $\Gamma \hookrightarrow \Omega$) y por el otro que t fuese el resultado de traducir la expresión e a notación de De Bruijn (en notación $\prod_1(\Gamma) \vdash e \rightarrow t$). Lo anterior se enuncia formalmente en el siguiente teorema.

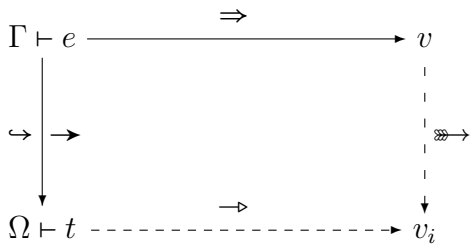
Teorema 5.4.1. Para todo Γ, Ω, e, t, v si

$$\begin{array}{l} \Gamma \vdash e \Rightarrow v, \\ \Gamma \hookrightarrow \Omega, \\ \prod_1(\Gamma) \vdash e \rightarrow t \end{array}$$

entonces existe v_i tal que

$$v \twoheadrightarrow v_i \quad \wedge \quad \Omega \vdash t \rightarrow v_i.$$

El teorema anterior lo podemos representar gráficamente mediante el siguiente diagrama:



En este diagrama las líneas sólidas son las hipótesis y las líneas segmentadas son los proposiciones a demostrar. Esto es, suponiendo por un lado que $\Gamma \vdash e \Rightarrow v$, por otro que $\Gamma \hookrightarrow \Omega$ y por otro más que $\prod_1(\Gamma) \vdash e \twoheadrightarrow t$ entonces debemos demostrar que existe v_i tal que $v \Lsh\!\!\Rightarrow v_i$ y que $\Omega \vdash t \twoheadrightarrow v_i$.

En Coq el teorema anterior, se escribe de la siguiente manera:

```
(**
 * Teorema de corrección de la traducción del lenguaje fuente
 * con notación basada en nombres a notación de De Bruijn.
 *
 **)
Theorem CTNtoDB:
forall e v G,
BSSMML G e v ->
forall O,
ENtoDB G O ->
forall ei,
NtoDB (envfe G) e ei ->
exists vi,
VNtoDB v vi /\
BSSMMLDB O ei vi.
```

5.5. Máquina SECD moderna

Toca el turno de presentar nuestra máquina objetivo que como dijimos será la máquina SECD moderna.

Como mencionamos Landin en [Lan64] introduce la máquina SECD para evaluar expresiones de un cálculo lambda extendido para su uso como lenguaje de programación. El nombre SECD hace alusión a los componentes con los que cuenta: *Stack*, *Environment*,

Control y *Dump*. La pila es una pila de evaluación,[15] el entorno sirve para llevar el seguimiento de los valores de los identificadores, es decir, podemos verlo como una pila de identificadores, el control es donde se encuentran las instrucciones por ejecutar y la primera de éstas es la que determina el siguiente paso de la máquina y dump es una pila donde se almacena la información del contexto actual antes de realizar una llamada a una función así cuando esta llamada termine se puede restablecer este contexto y continuar la ejecución del programa.

Como la exposición de la SECD en [Lan64] es un tanto vaga, una de las presentaciones de la SECD a la que más se hace referencia en la literatura es aquella debida a Henderson [Hen80], donde se expone el funcionamiento de la SECD de manera más precisa y formal.

Por otra parte Leroy [Ler16b] introduce una versión de la SECD con dos características principales que la distinguen de la SECD clásica:

- Utiliza índices de De Bruijn para manejar el entorno.[16]
- En lugar de utilizar una pila exclusivamente para almacenar la información del contexto actual antes de una llamada a una función, es decir, en lugar de tener una pila Dump, utiliza la misma pila Stack con este fin, por lo cual es necesario el uso de marcos de pila (*stack frames*).

De esta manera se obtiene una máquina SECD más eficiente y más cercana a una máquina real. A esta máquina Leroy la denomina máquina SECD moderna.[17]

Para nosotros, como ya contamos con una representación del código fuente que utiliza índices de De Bruijn, resulta natural y claro generar código hacia la SECD moderna.[18]

La máquina SECD moderna que presenta Leroy en [Ler16b], únicamente considera soporte para un lenguaje con variables, let, abstracciones y aplicación. Por tanto nosotros nos encargamos de extenderla para dar soporte a todo nuestro lenguaje fuente basándonos, siempre que fue posible, en la presentación de la SECD de Henderson, para seguir lo más fielmente posible la definición de la SECD como se considera en la literatura.[19]

Un punto donde fue imposible seguir a Henderson fue en el soporte de recursión. El soporte para recursión dado por él es por medio de las instrucciones *DUM* y *RAP*.

[15] En el sentido como las hemos utilizado en nuestros compiladores de ejemplo.

[16] Justo de manera análoga o como nosotros lo hicimos en la semántica con índices de De Bruijn

[17] Que en realidad debería llamarse SEC puesto que ya no cuenta con una pila Dump.

[18] También pudimos considerar generar código directamente a partir de nuestra representación basada en nombres, desde luego esta generación debía contener de forma implícita la traducción a índices de De Bruijn y en efecto así pudimos haberlo hecho, sin embargo, desde luego esta generación hubiese sido mucho más compleja y difícil de comprender. En general actualmente al desarrollar un compilador se considera una mejor opción la claridad y la simpleza que en este caso significa hacer una etapa dedicada únicamente a la traducción a índices. No obstante Boutin [Bou95] sigue el camino difícil, cuando genera código para la CAM a partir de una representación basada en nombres (la CAM utiliza implícitamente índices de De Bruijn). Y en efecto esta generación de código es compleja y difícil de entender.

[19] Por ejemplo, crear un marco de pila en la semántica para la instrucción ISel (que se usa en el soporte de un If) resulta innecesario y se puede definir una versión diferente más eficiente, sin embargo, no lo hicimos por seguir la definición dada por Henderson.

No fue posible seguir la misma técnica debido a que para implementar esta última se utiliza la pseudofunción *rplaca* la cual hace uso de apuntadores. Desde luego no se puede hacer uso de apuntadores en Coq, por lo que tuvimos que buscar otras alternativas y encontramos dos posibles soluciones que se pueden expresar en Coq[20] para dar soporte de recursión, a saber:

1. utilizar cerraduras recursivas

2. utilizar entornos recursivos

de éstas decidimos utilizar cerraduras recursivas y lo hicimos con éxito. Queda como posible trabajo futuro utilizar entornos recursivos y demostrar que ambas soluciones son equivalentes.

Por lo demás, en lo que resta nuestro compilador es muy similar a nuestro compilador de ejemplo para expresiones aritméticas de la sección 3.7 sólo que contemplando todo nuestro lenguaje fuente principal.

A continuación presentamos las instrucciones y los valores de la máquina:

$$
\begin{array}{rcl}
\textit{inst} & ::= & \text{IConst } n \\
& | & \text{IConstb } b \\
& | & \text{IAdd} \\
& | & \text{ISub} \\
& | & \text{IMul} \\
& | & \text{IEq} \\
& | & \text{IAcc } i \\
& | & \text{ILet} \\
& | & \text{IELet} \\
& | & \text{ISel } c\ c \\
& | & \text{IJoin} \\
& | & \text{IClos } c \\
& | & \text{IClos}_{rec}\ c \\
& | & \text{IApp} \\
& | & \text{IRet}
\end{array}
$$

$$
\begin{array}{rcl}
v_m & ::= & \text{MInt } n \\
& | & \text{MBool } b \\
& | & \text{MClos } c\ \Delta \\
& | & \text{MClos}_{rec}\ c\ \Delta
\end{array}
$$

$$
\begin{array}{rcl}
v_p & ::= & \text{SVal } v_m \\
& | & \text{Frame } c\ \Delta
\end{array}
$$

[20] En realidad sólo estamos seguros de que se puede expresar una en Coq, la de las cerraduras recursivas que fue la que nosotros utilizamos. Aunque desde luego conjeturamos que también es posible expresar entornos recursivos.

inst representa una instrucción. Por su parte v_m representa un valor de máquina, es decir, los valores que se almacenarán en el entorno.[21] Por otro lado, como mencionamos, la pila servirá como una pila de evaluación y por tanto en ella habrán valores, pero también dijimos que servirá para almacenar el contexto actual antes de una llamada a función y por tanto también debe poder almacenar marcos de pila. Por otro lado en el entorno como acabamos de decir se almacenan valores pero no marcos de pila porque éstos no son valores. Por eso se hace la distinción entre valores de máquina v_m y valores de pila v_p donde estos últimos son los valores de máquina junto con marcos de pila.

Δ es un entorno de la máquina que se maneja utilizando índices de De Bruijn de la misma manera que un entorno Ω (es decir, se maneja como una pila de valores), como vimos Ω únicamente almacenaba valores, es decir, era de la forma $[v_0, \ldots, v_n]$ y Δ es justo lo análogo para la máquina sólo que desde luego almacena valores de máquina, es decir, es de la forma $[v_{m_0}, \ldots, v_{m_n}]$, donde v_m denota un valor de máquina.

s como dijimos es una pila de evaluación que almacena valores de máquina y que cuenta además con las características que mencionamos arriba.

c denota código de máquina que consideramos como una secuencia de instrucciones.

La correspondiente formalización en Coq es la siguiente:

```
(*Instrucciones de la máquina SECD moderna*)
Inductive instr: Set:=
| IConst: nat -> instr
| IAdd: instr
| IAcc: nat -> instr
| IClos: list instr -> instr
| IClosr: list instr -> instr
| IApp: instr
| IRet: instr.

(*El código de la máquina se implementa como una lista de instrucciones*)
Definition code := list instr.

(*Valores de máquina*)
Inductive mval: Set :=
| MInt: nat -> mval
| MBool: bool -> mval
| MClos: code -> list mval -> mval
| MClosr: code -> list mval -> mval.

(**
 * Un entorno D de la máquina se implementa como una lista de valores
 * de máquina.
 *
 **)
Definition menv := list mval.
```

[21] Y que corresponderían a los valores que puede tomar una variable en la semántica del lenguaje.

```
(*Valores de pila*)
Inductive sval: Set :=
| SVal: mval -> sval
| SFrame: code -> menv -> sval.

(*La pila de la máquina se implementa como una lista de valores de máquina*)
Definition stack := list sval.

(**
 * Una configuración de la máquina se representa como una triada
 * (código,entorno,pila).
 *
 **)
Definition mconf : Set := (code * menv * stack)%type.
```

Ahora especificaremos su comportamiento, utilizando ambas semánticas: la de paso pequeño y la de paso grande y posteriormente probaremos su equivalencia.

5.5.1. Semántica de paso pequeño

Recordemos que una configuración de la máquina nos muestra el estado actual de ésta, es decir, una configuración es una triada (c, Δ, s) donde c es el código por ejecutar, Δ es el entorno actual de la máquina y s es la pila actual, así la semántica de paso pequeño $\rightarrow$ es una relación de transición que va de configuraciones a configuraciones.

Presentamos la semántica de paso pequeño de la máquina en la tabla 5.1.

Por otro lado, el correspondiente desarrollo de esta semántica en Coq es:

```
(**
 * Semántica de paso pequeño de la máquina SECD moderna.
 *
 **)
Inductive SSSSEC: mconf -> mconf -> Prop :=
| SSIConst: forall n:nat, forall c:code, forall D:menv,
  forall s:stack,
  SSSSEC (IConst n::c, D, s) (c, D, SVal (MInt n)::s)

| SSIAdd: forall n1 n2:nat, forall c:code, forall D:menv,
  forall s:stack,
  SSSSEC (IAdd::c,D,SVal (MInt n2)::SVal (MInt n1)::s)
       (c,D, SVal (MInt (n1+n2))::s)

| SSIAccess: forall n:nat, forall c:code, forall D:menv,
  forall s:stack, forall v:mval,
```

Configuración actual			Configuración siguiente		
Código	Entorno	Pila	Código	Entorno	Pila
[IConst n] $\cdot\, c$	Δ	s	c	Δ	[SVal MInt n] $\cdot\, s$
[IConstb b] $\cdot\, c$	Δ	s	c	Δ	[SVal MBool b] $\cdot\, s$
[IAdd] $\cdot\, c$	Δ	[SVal MInt n_2, SVal MInt n_1] $\cdot\, s$	c	Δ	[SVal MInt $n_1 + n_2$] $\cdot\, s$
[ISub] $\cdot\, c$	Δ	[SVal MInt n_2, SVal MInt n_1] $\cdot\, s$	c	Δ	[SVal MInt $n_1 - n_2$] $\cdot\, s$
[IMul] $\cdot\, c$	Δ	[SVal MInt n_2, SVal MInt n_1] $\cdot\, s$	c	Δ	[SVal MInt $n_1 * n_2$] $\cdot\, s$
[IEq] $\cdot\, c$	Δ	[SVal MInt n_2, SVal MInt n_1] $\cdot\, s$	c	Δ	[SVal MBool $n_1 = n_2$] $\cdot\, s$
[IAcc i] $\cdot\, c$	$[v_0, \ldots, v_i, \ldots, v_n] = \Delta$	s	c	Δ	[SVal v_i] $\cdot\, s$
[ILet] $\cdot\, c$	Δ	[SVal v] $\cdot\, s$	c	$[v] \cdot \Delta$	s
[IELet] $\cdot\, c$	$[v] \cdot \Delta$	s	c	Δ	s
[ISel $c_1 c_2$] $\cdot\, c$	Δ	[SVal MBool $true$] $\cdot\, s$	c_1	Δ	[Frame c []] $\cdot\, s$
[ISel $c_1 c_2$] $\cdot\, c$	Δ	[SVal MBool $false$] $\cdot\, s$	c_2	Δ	[Frame c []] $\cdot\, s$
[IJoin] $\cdot\, c$	Δ	[SVal v, Frame c_b []] $\cdot\, s$	c_b	Δ	[SVal v] $\cdot\, s$
[IClos c_1] $\cdot\, c$	Δ	s	c	Δ	[Sval (MClos c_1 Δ)] $\cdot\, s$
[IClos$_{rec}$ c_1] $\cdot\, c$	Δ	s	c	Δ	[Sval (MClos$_{rec}$ c_1 Δ)] $\cdot\, s$
[IApp] $\cdot\, c$	Δ	[SVal v, SVal (MClos c_1 Δ_1)] $\cdot\, s$	c_1	$[v] \cdot \Delta_1$	[Frame c Δ] $\cdot\, s$
[IApp] $\cdot\, c$	Δ	[SVal v, SVal (MClos$_{rec}$ c_1 Δ_1)] $\cdot\, s$	c_1	$[v] \cdot [\text{MClos}_{rec}\ c_1\ \Delta_1] \cdot \Delta_1$	[Frame c Δ] $\cdot\, s$
[IRet] $\cdot\, c$	Δ	[Sval v, Frame c_1 Δ_1] $\cdot\, s$	c_1	Δ_1	[SVal v] $\cdot\, s$

Tabla 5.1: Semántica de paso pequeño de la máquina SECD moderna.

```
              accessi D n = Some v ->
              SSSSEC (IAcc n::c, D, s) (c, D, SVal v::s)

| SSIClos: forall cc c:code, forall D:menv, forall s:stack,
    SSSSEC (IClos cc::c, D, s) (c, D, SVal (MClos cc D)::s)

| SSIClosr: forall cc c:code, forall D:menv, forall s:stack,
    SSSSEC (IClosr cc::c, D, s) (c, D, SVal (MClosr cc D)::s)

| SSIApp: forall c c1:code, forall D D1:menv, forall s:stack,
  forall v:mval,
    SSSSEC (IApp::c, D, SVal v:: SVal (MClos c1 D1)::s)
         (c1, v::D1, SFrame c D::s)

| SSIAppr:forall c c1:code, forall D D1:menv, forall s:stack,
  forall v:mval,
    SSSSEC (IApp::c, D, SVal v:: SVal (MClosr c1 D1)::s)
         (c1, v::(MClosr c1 D1)::D1, SFrame c D::s)

| SSIReturn: forall c c1:code, forall D D1:menv,
  forall v:mval, forall s:stack,
    SSSSEC (IRet::c, D, SVal v:: SFrame c1 D1 ::s)
         (c1, D1, SVal v::s).
```

Como podemos observar el nombre otorgado a esta semántica en Coq es `SSSSEC`, por lo que se tiene que: `SSSSEC (c1,D1,s1) (c2,D2,s2)` si y sólo si $(c_1, D_1, s_1) \rightarrow (c_2, D_2, s_2)$.

Ahora podemos utilizar verificación, tomando como especificación a cumplir la semántica paso pequeño de la máquina y de esa forma obtener un intérprete verificado de la máquina de un único paso.

A continuación presentamos nuestro intérprete de un único paso.

```
(**
 * Intérprete de un único paso que cumple con la especificación
 * de la semántica de paso pequeño de la máquina.
 *
 **)
Fixpoint mexec (m: mconf): option mconf :=
match m with
| (IConst a::cs, D, s) => Some (cs, D, SVal (MInt a)::s)
| (IAdd::c,D,SVal (MInt n2)::SVal (MInt n1)::s) =>
  Some (c,D,SVal (MInt (n1+n2))::s)
| (IAcc n::cs, D, s) =>
   match accessi D n with
     | Some v => Some (cs, D, SVal v::s)
     | None => None
   end
```

```
| (IClos cc::c, D, s) => Some (c, D,
                                SVal (MClos cc D)::s)
| (IClosr cc::c, D, s) => Some (c,D,
                                SVal (MClosr cc D)::s)
| (IApp::c, D, SVal v::SVal (MClos c1 D1)::s) =>
  Some (c1, v::D1, SFrame c D::s)
| (IApp::c, D, SVal v::SVal (MClosr c1 D1)::s) =>
  Some (c1, v::(MClosr c1 D1)::D1, SFrame c D::s)
| (IRet::c,D, SVal v::SFrame c1 D1::s) =>
  Some (c1,D1, SVal v::s)
| (nil,ev,s) => Some (nil,ev,s)
| _ => None
end.
```

Y ahora presentamos el lema que enuncia que en efecto este intérprete cumple con la semántica de paso pequeño de la máquina.

```
(**
 * Lema que enuncia que el intérprete "mexec" cumple con la especificación
 * de la semántica de paso de paso pequeño de la máquina.
 *
 **)
Lemma mexecc:
forall mi mf,
SSSSEC mi mf ->
mexec mi = Some mf.
```

Podemos además utilizar el mecanismo de extracción y así obtener un intérprete de la máquina verificado de un único paso escrito en un lenguaje funcional convencional, en este caso OCaml, listo para utilizarse en la vida real. Lo cual se muestra a continuación.

```
Extraction mexec.

(** val mexec : mconf -> mconf option **)

let rec mexec = function
| Pair (p, s) ->
  let Pair (c0, ev) = p in
  (match c0 with
   | Nil -> Some (Pair ((Pair (Nil, ev)), s))
   | Cons (i, c) ->
     (match i with
```

```
      | IConst a -> Some (Pair ((Pair (c, ev)), (Cons ((SVal (MInt a)), s))))
      | IAdd ->
        (match s with
         | Nil -> None
         | Cons (s0, l) ->
           (match s0 with
            | SVal m0 ->
              (match m0 with
               | MInt n2 ->
                 (match l with
                  | Nil -> None
                  | Cons (s1, s2) ->
                    (match s1 with
                     | SVal m1 ->
                       (match m1 with
                        | MInt n1 ->
                          Some (Pair ((Pair (c, ev)), (Cons ((SVal (MInt
                            (plus n1 n2))), s2))))
                        | _ -> None)
                     | SFrame (c1, m1) -> None))
               | _ -> None)
            | SFrame (c1, m0) -> None))
      | IAcc n ->
        (match accessi ev n with
         | Some v -> Some (Pair ((Pair (c, ev)), (Cons ((SVal v), s))))
         | None -> None)
      | IClos cc ->
        Some (Pair ((Pair (c, ev)), (Cons ((SVal (MClos (cc, ev))), s))))
      | IClosr cc ->
        Some (Pair ((Pair (c, ev)), (Cons ((SVal (MClosr (cc, ev))), s))))
      | IApp ->
        (match s with
         | Nil -> None
         | Cons (s0, l) ->
           (match s0 with
            | SVal v ->
              (match l with
               | Nil -> None
               | Cons (s1, s2) ->
                 (match s1 with
                  | SVal m0 ->
                    (match m0 with
                     | MClos (c1, d1) ->
                       Some (Pair ((Pair (c1, (Cons (v, d1)))), (Cons
                         ((SFrame (c, ev)), s2))))
                     | MClosr (c1, d1) ->
                       Some (Pair ((Pair (c1, (Cons (v, (Cons ((MClosr (c1,
                         d1)), d1)))))), (Cons ((SFrame (c, ev)), s2))))
                     | _ -> None)
```

```
                | SFrame (c1, m0) -> None))
          | SFrame (c1, m0) -> None))
      | IRet ->
        (match s with
         | Nil -> None
         | Cons (s0, l) ->
           (match s0 with
            | SVal v ->
              (match l with
               | Nil -> None
               | Cons (s1, s2) ->
                 (match s1 with
                  | SVal m0 -> None
                  | SFrame (c1, d1) ->
                    Some (Pair ((Pair (c1, d1)), (Cons ((SVal v), s2))))))
          | SFrame (c1, m0) -> None))))
```

Como mencionamos contamos ya con un intérprete de un único paso de la máquina. Para obtener un intérprete de muchos pasos que permita evaluar por completo el código de máquina que se le alimente, debemos considerar primero la cerradura transitiva y reflexiva de la semántica de paso pequeño de la máquina. La cual se expresa en Coq de la siguiente manera.

```
(**
 * Cerradura transitiva y reflexiva de la semántica
 * de paso pequeño de la máquina.
 *
 **)
Inductive TRCSSSSEC: mconf -> mconf -> Prop :=
| TRCR: forall m:mconf, TRCSSSSEC m m
| TRCT: forall m1 m2 m3:mconf,
          SSSSEC m1 m2 -> TRCSSSSEC m2 m3 -> TRCSSSSEC m1 m3.
```

Ahora podemos realizar verificación tomando la cerradura transitiva y reflexiva de la semántica de paso pequeño como especificación. Por lo que a continuación presentamos el intérprete de muchos pasos que sigue esta especificación.

```
(**
 * Intérprete de muchos pasos que simula la cerradura
 * transitiva y reflexiva de la semántica de paso pequeño
 * de la máquina llamando recursivamente al intérprete
 * de un único paso.
```

```
 *
 **)
Fixpoint mexecms (depthR:nat) (m: mconf): option mconf :=
match depthR with
| O => Some m
| S n =>
match mexec m with
| None => None
| Some (nil,D,s) => Some (nil,D,s)
| Some mi => mexecms n mi
end
end.
```

Notemos cómo es intérprete simula la cerradura transitiva y reflexiva de la semántica de paso pequeño ejecutando recursivamente el intérprete de un único paso.

Ahora mostremos que en efecto cumple, con la especificación, es decir, con la cerradura transitiva y reflexiva de la semántica de paso pequeño.

```
(**
 * Lema que enuncia que el intérprete "mexecms" cumple con la especificación
 * de la cerradura transitiva y reflexiva de la semántica de paso pequeño de
 * la máquina.
 *
 **)
Lemma mexecmsc:forall mi mf,
TRCSSSSEC mi mf ->
exists n,
mexecms n mi = Some mf.
Proof.
```

Además como se observa abajo podemos utilizar este intérprete para calcular el código de máquina correspondiente a nuestro ejemplo motivacional y precisamente obtenemos el resultado esperado, es decir `120`.

```
Compute(
mexecms 75
((IClosr (IAcc 0 :: IConst 0 :: IEq :: ISel (IConst 1 :: IJoin :: nil)
 (IAcc 0 :: IAcc 1 :: IAcc 0 :: IConst 1 :: ISub :: IApp::IMul::IJoin::nil)
 :: IRet :: nil)
         :: IConst 5 :: IApp :: nil),nil,nil)
).
     = Some (nil, nil, SVal (MInt 120) :: nil)
     : option mconf
```

Notemos aquí que es de gran relevancia contar con un intérprete verificado de la máquina, pues con esto, podremos (en el caso de desarrollar nuestro compilador de nuestro de lenguaje en notación de De Bruijn a código de la máquina SECD moderna)[22] no sólo compilar los programas escritos en nuestro lenguaje fuente sino que además podremos evaluarlos utilizando este intérprete de la máquina.[23]

Ahora podemos hacer uso del mecanismo de extracción y de esta manera obtener el intérprete de muchos pasos verificado de la máquina escrito en OCaml listo para usarse en la vida real tal como se ilustra a continuación.

```
Extraction mexecms.

(** val mexecms : nat -> mconf -> mconf option **)

let rec mexecms depthR m =
  match depthR with
  | O -> Some m
  | S n ->
    (match mexec m with
     | Some mi ->
       let Pair (p, s) = mi in
       let Pair (c, d) = p in
       (match c with
        | Nil -> Some (Pair ((Pair (Nil, d)), s))
        | Cons (i, l) -> mexecms n mi)
     | None -> None)
```

Para finalizar mencionamos que la semántica original de la máquina era ésta de paso pequeño que nosotros extendimos para dar soporte a los nuevos enunciados que conforman nuestro lenguaje fuente. En la próxima sección presentaremos la nueva semántica de paso grande de la máquina que desarrollamos influidos por la visión unificadora de Khan.

5.5.2. Semántica de paso grande

En esta sección nos encargaremos de presentar la nueva semántica de paso grande que desarrollamos para la máquina.

Para comenzar nuestra presentación, podemos mencionar que una configuración de la máquina es una triada (c, Δ, s), donde c es un código de la máquina, Δ un entorno de la máquina (que utiliza índices de De Bruijn) y s una pila. Así en principio la semántica de paso grande denota una relación que va de configuraciones a configuraciones finales

[22] Como en efecto lo haremos más adelante.

[23] En realidad también desarrollaremos otro intérprete de la máquina más adelante. El correspondiente a la semántica de paso grande. De esta manera contaremos con dos intérpretes verificados de la máquina.

de la máquina. Esto es, si tenemos que $(c, \Delta, s) \Rightarrow m_f$ donde $m_f = (c_f, \Delta_f, s_f)$ es una configuración final, entonces lo anterior se escribe como $\Delta, s \vdash c \Rightarrow m_f$.

Una vez dicho lo anterior presentamos ahora la semántica de paso grande de la máquina.

$$\frac{\Delta, [\text{SVal MInt } n] \cdot s \vdash c \Rightarrow m_f}{\Delta, s \vdash [\text{IConst } n] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{SVal MBool } b] \cdot s \vdash c \Rightarrow m_f}{\Delta, s \vdash [\text{IConstb } b] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{SVal MInt } n_1 + n_2] \cdot s \vdash c \Rightarrow m_f}{\Delta, [\text{SVal MInt } n_2, \text{SVal MInt } n_1] \cdot s \vdash [\text{IAdd}] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{SVal MInt } n_1 - n_2] \cdot s \vdash c \Rightarrow m_f}{\Delta, [\text{SVal MInt } n_2, \text{SVal MInt } n_1] \cdot s \vdash [\text{ISub}] \cdot c \Rightarrow m_f} \text{ (con } (n_1 - n_2) \geq 0)$$

$$\frac{\Delta, [\text{SVal MInt } n_1 * n_2] \cdot s \vdash c \Rightarrow m_f}{\Delta, [\text{SVal MInt } n_2, \text{SVal MInt } n_1] \cdot s \vdash [\text{IMul}] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{SVal MBool } \ n_1 = n_2] \cdot s \vdash c \Rightarrow m_f}{\Delta, [\text{SVal MInt } n_2, \text{SVal MInt } n_1] \cdot s \vdash [\text{IEq}] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta = [v_0, \ldots, v_i, \ldots, v_n] \qquad \Delta, [\text{SVal } v_i] \cdot s \vdash c \Rightarrow m_f}{\Delta, s \vdash [\text{ IAcc } i] \cdot c \Rightarrow m_f}$$

$$\frac{[v] \cdot \Delta, s \vdash c \Rightarrow m_f}{\Delta, [\text{ SVal v}] \cdot s \vdash [\text{ILet}] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, s \vdash c \Rightarrow m_f}{[v] \cdot \Delta, s \vdash [\text{IELet}] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{Frame } c\ [\,]] \cdot s \vdash c_1 \Rightarrow m_f}{\Delta, [\text{SVal MBool } true] \cdot s \vdash [\text{ISel } c_1\ c_2] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{Frame } c\ [\,]] \cdot s \vdash c_2 \Rightarrow m_f}{\Delta, [\text{SVal MBool } false] \cdot s \vdash [\text{ISel } c_1\ c_2] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{SVal } v] \cdot s \vdash c_b \Rightarrow m_f}{\Delta, [\text{SVal } v, \text{Frame } c_b\ []] \cdot s \vdash [\text{IJoin}] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{SVal MClos } c_1\ \Delta] \cdot s \vdash c \Rightarrow m_f}{\Delta, s \vdash [\text{IClos } c_1] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta, [\text{SVal MClosr } c_1\ \Delta] \cdot s \vdash c \Rightarrow m_f}{\Delta, s \vdash [\text{IClosr } c_1] \cdot c \Rightarrow m_f}$$

$$\frac{[v] \cdot \Delta_1, [\text{Frame } c\ \Delta] \cdot s \vdash c_1 \Rightarrow m_f}{\Delta, [\text{SVal } v, \text{SVal (MClos } c_1\ \Delta_1)] \cdot s \vdash [\text{IApp }] \cdot c \Rightarrow m_f}$$

$$\frac{[v] \cdot [\text{ MClosr } c_1\ \Delta_1] \cdot \Delta_1, [\text{Frame c } \Delta] \cdot s \vdash c_1 \Rightarrow m_f}{\Delta, [\text{SVal } v, \text{SVal (MClosr } c_1\ \Delta_1)] \cdot s \vdash [\text{IApp }] \cdot c \Rightarrow m_f}$$

$$\frac{\Delta_1, [\text{SVal } v] \cdot s \vdash c_1 \Rightarrow m_f}{\Delta, [\text{SVal } v, \text{Frame } c_1\ \Delta_1] \cdot s \vdash [\text{IRet }] \cdot c \Rightarrow m_f}$$

Por su parte el desarrollo correspondiente a esta semántica en Coq es el siguiente.

```
(**
 * Semántica de paso grande de la máquina SECD moderna.
 *
 **)
Inductive BSSSEC: mconf -> mconf -> Prop :=
| BSRC: forall k:mconf, BSSSEC k k

| BSIConst: forall n:nat, forall c:code, forall D:menv,
  forall s:stack, forall mf:mconf,
  BSSSEC (c,D, SVal (MInt n)::s) mf ->
  BSSSEC (IConst n::c, D, s) mf

| BSIAdd: forall n1 n2:nat, forall c:code, forall D:menv,
  forall s:stack, forall mf:mconf,
  BSSSEC (c,D, SVal (MInt (n1+n2))::s) mf ->
  BSSSEC (IAdd::c,D,SVal (MInt n2)::SVal (MInt n1)::s) mf

| BSIAccess: forall n:nat, forall v:mval, forall c:code,
  forall D:menv, forall s:stack, forall mf:mconf,
  accessi D n = Some v ->
  BSSSEC (c,D,SVal v::s) mf ->
```

```
  BSSSEC (IAcc n::c,D,s) mf

| BSIClos: forall c cc:code, forall D:menv, forall s:stack,
  forall mf:mconf,
  BSSSEC (c,D,SVal (MClos cc D)::s) mf ->
  BSSSEC (IClos cc::c,D,s) mf

| BSIClosr: forall c cc:code, forall D:menv, forall s:stack,
  forall mf:mconf,
  BSSSEC (c,D,SVal (MClosr cc D)::s) mf ->
  BSSSEC (IClosr cc::c,D,s) mf

| BSIApp: forall v:mval, forall c c1:code,
  forall D D1:menv, forall s:stack, forall mf:mconf,
  BSSSEC (c1,v::D1,SFrame c D::s) mf ->
  BSSSEC (IApp::c,D,SVal v::SVal (MClos c1 D1)::s) mf

| BSIAppr: forall v:mval, forall c c1:code,
  forall D D1:menv, forall s:stack, forall mf:mconf,
  BSSSEC (c1, v::(MClosr c1 D1)::D1,SFrame c D::s) mf ->
  BSSSEC (IApp::c,D,SVal v::SVal (MClosr c1 D1)::s) mf

| BSIReturn: forall v:mval, forall c c1:code,
  forall D D1:menv, forall s:stack, forall mf:mconf,
  BSSSEC (c1,D1,SVal v::s) mf ->
  BSSSEC (IRet::c,D,SVal v::SFrame c1 D1::s) mf.
```

Como se puede notar el nombre dado a esta semántica en Coq es `BSSSEC`, por lo que se tiene que: `BSSSEC (c,D,s) mf` si y sólo si $D, s \vdash c \Rightarrow m_f$.

Ahora utilizamos verificación, por lo que ofrecemos el intérprete de la máquina y posteriormente mostraremos que éste cumple con la especificación, es decir, con la semántica de paso grande de la máquina.

```
(**
 * Intérprete que cumple con la especificación de la semántica
 * de paso grande de la máquina.
 *
 **)
Fixpoint mexecrec (depthR:nat) (m: mconf)
: option mconf :=
match depthR with
| O => Some m
| S k => match m with
           | (IConst a::cs, D, s) =>
             mexecrec k (cs, D, SVal (MInt a)::s)
```

```
            | (IAdd::cs,D,SVal (MInt n2)::SVal (MInt n1)::s) =>
              mexecrec k (cs, D, SVal (MInt (n1+n2))::s)
            | (IAcc n::cs, D, s) =>
              match accessi D n with
                | Some v => mexecrec k (cs, D,SVal v::s)
                | None => None
              end
            | (IClos cc::cs, D, s) =>
               mexecrec k (cs, D, SVal (MClos cc D)::s)
            | (IClosr cc::cs, D, s) =>
               mexecrec k (cs, D, SVal (MClosr cc D)::s)
            | (IApp::cs, D, SVal v::SVal (MClos c1 D1)::s) =>
               mexecrec k (c1, v::D1, SFrame cs D::s)
            | (IApp::cs, D, SVal v::SVal (MClosr c1 D1)::s) =>
               mexecrec k
                 (c1, v::(MClosr c1 D1)::D1,SFrame cs D::s)
            | (IRet::cs,D, SVal v::SFrame c1 D1::s) =>
               mexecrec k (c1, D1, SVal v::s)
            | (nil,ev,s) => Some (nil,ev,s)
            | _ => None
          end
end.
```

Mostraremos ahora que en efecto este intérprete cumple con la semántica de paso grande de la máquina.

```
(**
 * Lema que enuncia que el intérprete "mexecrec" cumple con la
 * especificación de la semántica de paso grande de la máquina.
 *
 **)
Lemma mexecrecc:
forall mi mf,
BSSSEC mi mf ->
exists n, mexecrec n mi = Some mf.
```

Más aún podemos utilizarlo para calcular nuestro ejemplo motivacional escrito en código de máquina como se muestra a continuación.

```
(*Ejemplo de cálculo intérprete de la máquina mexecrec*)
Compute(
mexecrec 75
```

```
((IClosr (IAcc 0 :: IConst 0 :: IEq :: ISel (IConst 1 :: IJoin :: nil)
 (IAcc 0 :: IAcc 1 :: IAcc 0 :: IConst 1 :: ISub :: IApp::IMul::IJoin::nil)
 :: IRet :: nil)
         :: IConst 5 :: IApp :: nil),nil,nil)
).
     = Some (nil, nil, SVal (MInt 120) :: nil)
     : option mconf
```

Como podemos observar se obtiene justamente el resultado esperado, es decir `120`.

Obsérvese que con esto hemos obtenido un nuevo intérprete de la máquina, esta vez uno que sigue la semántica de paso grande.

Más aún podemos utilizar el mecanismo de extracción y obtener un intérprete verificado de la máquina que sigue la semántica de paso grande listo para utilizarse en la vida real. Lo que mostramos a continuación.

```
Extraction mexecrec.

(** val mexecrec : nat -> mconf -> mconf option **)

let rec mexecrec depthR m =
  match depthR with
  | O -> Some m
  | S k ->
    let Pair (p, s) = m in
    let Pair (c, ev) = p in
    (match c with
     | Nil -> Some (Pair ((Pair (Nil, ev)), s))
     | Cons (i, cs) ->
       (match i with
        | IConst a ->
          mexecrec k (Pair ((Pair (cs, ev)), (Cons ((SVal (MInt a)), s))))
        | IAdd ->
          (match s with
           | Nil -> None
           | Cons (s0, l) ->
             (match s0 with
              | SVal m0 ->
                (match m0 with
                 | MInt n2 ->
                   (match l with
                    | Nil -> None
                    | Cons (s1, s2) ->
                      (match s1 with
                       | SVal m1 ->
                         (match m1 with
                          | MInt n1 ->
```

```
                        mexecrec k (Pair ((Pair (cs, ev)), (Cons ((SVal
                          (MInt (plus n1 n2))), s))))
                      | _ -> None)
                  | SFrame (c0, m1) -> None))
          | _ -> None)
      | SFrame (c0, m0) -> None))
| IAcc n ->
  (match accessi ev n with
   | Some v ->
     mexecrec k (Pair ((Pair (cs, ev)), (Cons ((SVal v), s))))
   | None -> None)
| IClos cc ->
  mexecrec k (Pair ((Pair (cs, ev)), (Cons ((SVal (MClos (cc, ev))),
    s))))
| IClosr cc ->
  mexecrec k (Pair ((Pair (cs, ev)), (Cons ((SVal (MClosr (cc, ev))),
    s))))
| IApp ->
  (match s with
   | Nil -> None
   | Cons (s0, l) ->
     (match s0 with
      | SVal v ->
        (match l with
         | Nil -> None
         | Cons (s1, s2) ->
           (match s1 with
            | SVal m0 ->
              (match m0 with
               | MClos (c1, d1) ->
                 mexecrec k (Pair ((Pair (c1, (Cons (v, d1)))),(Cons
                   ((SFrame (cs, ev)), s2))))
               | MClosr (c1, d1) ->
                 mexecrec k (Pair ((Pair (c1, (Cons (v, (Cons
                   ((MClosr (c1, d1)), d1)))))), (Cons ((SFrame (cs,
                   ev)), s2))))
               | _ -> None)
            | SFrame (c0, m0) -> None))
      | SFrame (c0, m0) -> None))
| IRet ->
  (match s with
   | Nil -> None
   | Cons (s0, l) ->
     (match s0 with
      | SVal v ->
        (match l with
         | Nil -> None
         | Cons (s1, s2) ->
           (match s1 with
```

```
                | SVal m0 -> None
                | SFrame (c1, d1) ->
                  mexecrec k (Pair ((Pair (c1, d1)), (Cons ((SVal v),
                    s2))))))
          | SFrame (c0, m0) -> None))))
```

Nótese cómo al desarrollar un programa que sigue la semántica de paso grande hemos obtenido un intérprete que evalúa directamente el código de máquina completo a un valor. En comparación cuando utilizamos la semántica de paso pequeño primero tuvimos que desarrollar un intérprete de un único paso que siguiera la semántica de paso pequeño y luego uno que simulara la cerradura transitiva y reflexiva de ésta. Esto debido a que el intérprete de un único paso no nos servía para evaluar por completo el código que se le alimentaba. Esto lo consideramos como un punto a favor de la semántica de paso grande.

Habiendo dado ambas semánticas para la máquina, esto es, la de paso pequeño y la de paso grande, demostraremos su equivalencia en la siguiente sección.

5.5.3. Equivalencia entre semánticas

En esta sección presentaremos la equivalencia entre las dos semánticas de la máquina, la de paso pequeño y la de paso grande.

Para comenzar recordemos que la semántica de paso pequeño "$\rightarrow$" es una relación de transición que va de configuraciones a configuraciones de la máquina, esto es, si tenemos dos configuraciones $m_1 = (c_1, \Delta_1, s_1)$ y $m_2 = (c_2, \Delta_2, s_2)$ de la máquina entonces $m_1 \rightarrow m_2$ denota una transición que va de la configuración m_1 a la configuración m_2. Nótese que intuitivamente esto equivale a realizar un paso en la evaluación de un programa escrito en código de la máquina. Por lo que si se desea evaluar el programa por completo es necesario considerar la cerradura transitiva y reflexiva "$\rightarrow^*$" de esta semántica. De esta manera podemos decir que partiendo de una configuración inicial $m_i = (c_i, \Delta_i, s_i)$ se llegará a una configuración final $m_f = (c_f, \Delta_f, s_f)$ si se tiene que $(c_i, \Delta_i, s_i) \rightarrow^* (c_f, \Delta_f, s_f)$.

Por otra parte, podemos decir que la semántica de paso grande es una relación que va de configuraciones a configuraciones finales, nótese cómo en esta semántica se llega directamente a configuraciones finales, lo que intuitivamente significa que evalúa por completo un programa. Entonces esta vez partiendo de la configuración inicial m_i se llegará a la directamente a la configuración final m_f si se tiene que $m_i \Rightarrow m_f$ lo que se denota como $\Delta_i, s_i \vdash c_i \Rightarrow (c_f, \Delta_f, s_f)$.

Entonces podemos notar cómo en ambos casos partiendo de una configuración inicial m_i se llega a la misma configuración final m_f. Luego, para probar la equivalencia en un sentido (de la semántica de paso pequeño a la semántica de paso grande) podemos decir que si se tiene que $(c_i, \Delta_i, s_i) \rightarrow^* (c_f, \Delta_f, s_f)$ entonces se tendrá que $\Delta_i, s_i \vdash c_i \Rightarrow (c_f, \Delta_f, s_f)$. En el sentido contrario (de la semántica de paso grande a la de paso pequeño) podemos decir que si se tiene que $\Delta_i, s_i \vdash c_i \Rightarrow (c_f, \Delta_f, s_f)$ entonces se cumplirá que $(c_i, \Delta_i, s_i) \rightarrow^* (c_f, \Delta_f, s_f)$. Lo anterior se enuncia formalmente en el siguiente teorema.

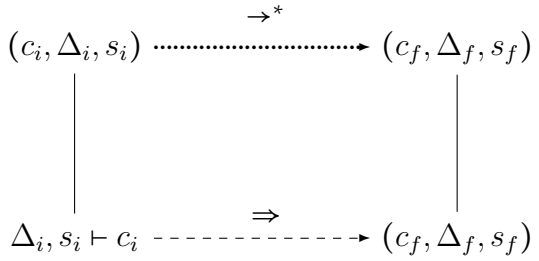
Teorema 5.5.1. Para toda configuración $m_i = (c_i, \Delta_i, s_i)$, $m_f = (c_f, \Delta_f, s_f)$,

$$(c_i, \Delta_i, s_i) \rightarrow^* (c_f, \Delta_f, s_f)$$

si y sólo si

$$\Delta_i, s_i \vdash c_i \Rightarrow (c_f, \Delta_f, s_f)$$

A continuación representamos gráficamente este teorema mediante un diagrama.



Con base en el diagrama podemos decir que para demostrar la equivalencia en un sentido debemos tomar una de las líneas (con flecha) como hipótesis y la otra como la proposición a demostrar. Así para demostrar que de la semántica de paso pequeño se sigue la de paso grande, tomamos como hipótesis la proposición de la línea con puntos $(c_i, \Delta_i, s_i) \rightarrow^* (c_f, \Delta_f, s_f)$ y debemos demostrar la de la línea segmentada $\Delta_i, s_i \vdash c_i \Rightarrow (c_f, \Delta_f, s_f)$. Para demostrar que de la semántica de paso grande se sigue la de paso pequeño tomamos ahora como hipótesis la proposición de la línea segmentada $\Delta_i, s_i \vdash c_i \Rightarrow (c_f, \Delta_f, s_f)$ y debemos demostrar que se cumple la de la línea punteada $(c_i, \Delta_i, s_i) \rightarrow^* (c_f, \Delta_f, s_f)$.

De esta forma queda establecida la equivalencia entre las dos semánticas.

En Coq el teorema anterior se expresa mediante los siguientes lemas:

```
(**
 * Lema que enuncia que de (la cerradura transitiva
 * y reflexiva de) la semántica de paso pequeño de
 * la máquina se sigue la semántica de paso grande.
 *
 **)
Lemma TRCSSSSECTBSSSEC:
forall mi mf,
TRCSSSSEC mi mf ->
BSSSEC mi mf.
```

```
(**
 * Lema que enuncia que de la semántica de paso grande
 * de la máquina se sigue (la cerradura transitiva y reflexiva
 * de) la semántica de paso pequeño.
 *
 **)
Lemma BSSSECTTRCSSSSEC:
forall mi mf,
BSSSEC mi mf ->
TRCSSSSEC mi mf.
```

Habiendo mostrado ya la equivalencia entre las dos semánticas de la máquina, presentaremos ahora en la siguiente sección la compilación de nuestro lenguaje fuente con notación de De Bruijn a código de máquina de la SECD moderna.

5.6. Compilación

Esta sección la dedicaremos al desarrollo de la compilación de nuestro lenguaje fuente en notación de De Bruijn a código de máquina de la SECD moderna.

Esta vez para especificar la compilación utilizaremos una función porque en este caso resulta más simple y natural.

Así la compilación de nuestro lenguaje fuente en notación de De Bruijn a código de máquina es la que se muestra en la figura 5.1.

Y su correspondiente desarrollo en Coq es:

```
(**
 * Función que define la compilación del lenguaje fuente
 * con índices de De Bruijn al código de la maquina SECD
 * moderna.
 *
 **)
Fixpoint compile (e:MMLDBexp) : code :=
match e with
| ConstI n => IConst n::nil
| PlusI e1 e2 => compile e1 ++ compile e2 ++ IAdd::nil
| VarI n => IAcc n::nil
| LamI lexp => (IClos ((compile lexp) ++ (IRet::nil)))::nil
| MuI mexp => (IClosr ((compile mexp) ++ (IRet::nil)))::nil
| AppI e1 e2 => compile e1 ++ compile e2 ++ IApp::nil
end.
```

$$
Compile(e) = \begin{cases}
[\text{IConst } n] & e = \text{ConstI } n \\
[\text{IConstb } b] & e = \text{ConstbI } b \\
Compile\ e_1 \cdot Compile\ e_2 \cdot [\text{IAdd}] & e = \text{PlusI } e_1\ e_2 \\
Compile\ e_1 \cdot Compile\ e_2 \cdot [\text{ISub}] & e = \text{MinusI } e_1\ e_2 \\
Compile\ e_1 \cdot Compile\ e_2 \cdot [\text{IMul}] & e = \text{TimesI } e_1\ e_2 \\
Compile\ e_1 \cdot Compile\ e_2 \cdot [\text{IEq}] & e = \text{EqI } e_1\ e_2 \\
[\text{IAcc } i] & e = \text{VarI } i \\
Compile\ e_1 \cdot [\text{ILet}] \cdot Compile\ e_2 \cdot [\text{IELet}] & e = \text{LetI } e_1\ e_2 \\
Compile\ e_1 \cdot [\text{ISel } (Compile\ e_2 \cdot [\text{IJoin}]) & \\
\qquad (Compile\ e_3 \cdot [\text{IJoin}])] & e = \text{IfI } e_1\ e_2\ e_3 \\
[\text{IClos } (Compile\ e \cdot [\text{IRet}])] & e = \text{LamI } e \\
[\text{IClos}_{rec}(Compile\ e \cdot [\text{IRet}])] & e = \text{MuI } e \\
Compile\ e_1 \cdot Compile\ e_2 \cdot [\text{IApp}] & e = \text{AppI } e_1\ e_2
\end{cases}
$$

Figura 5.1: Compilación del lenguaje fuente en notación de De Bruijn a código de la máquina SECD moderna.

Ahora como ilustramos mediante el siguiente ejemplo, podemos utilizar la función `compile`, es decir nuestro compilador, para realizar la traducción de nuestro ejemplo motivacional escrito en notación de De Bruijn a código de máquina.

```
(*Ejemplo de cálculo compilador compile*)
Compute (compile
(AppI (MuI (IfI (EqI (VarI 0) (ConstI 0))
             (ConstI 1)
             (TimesI (VarI 0)
                 (AppI (VarI 1) (MinusI (VarI 0) (ConstI 1)))))) (ConstI 5))
).
     = IClosr
         (IAcc 0
          :: IConst 0
             :: IEq
                :: ISel (IConst 1 :: IJoin :: nil)
                     (IAcc 0
                      :: IAcc 1
                         :: IAcc 0
                            :: IConst 1
                               :: ISub :: IApp :: IMul :: IJoin :: nil)
                   :: IRet :: nil) :: IConst 5 :: IApp :: nil
     : code
```

Obsérvese que esta vez no es necesario realizar verificación porque estamos utilizando

directamente una función para definir la compilación. Y como sabemos una función tiene contenido computacional en Coq, por eso podemos utilizarla para realizar cálculos sobre ella como en particular lo ilustramos en el ejemplo anterior.

Más adelante después de demostrar la corrección de este compilador utilizaremos el mecanismo de extracción para así obtener un compilador correcto verificado de nuestro lenguaje fuente en notación de De Bruijn a código de máquina de la SECD moderna.

Pero antes en vías de poder formular la corrección de la compilación debemos dar la compilación (equivalencia) entre los valores y entornos que se utilizan en la semántica de nuestro lenguaje fuente en notación de De Bruijn y los valores y entornos de la máquina SECD moderna.

5.6.1. Compilación de valores y entornos

Presentamos ahora la compilación de valores y entornos del lenguaje con índices de De Bruijn a valores y entornos de la máquina. Aquí también para especificar esta traducción utilizamos una función por simpleza y naturalidad.

$$CompVal(v) = \begin{cases} \text{MInt } i & v = \text{NumI } i \\ \text{MBool } b & v = \text{BoolI } b \\ \text{MClos } (Compile\ e \cdot [\text{IRet}])\ (CompEnv\ \Omega) & v = \text{ClosI } e\ \Omega \\ \text{MClos}_{rec}\ (Compile\ e \cdot [\text{IRet}])\ (CompEnv\ \Omega) & v = \text{ClosI}_{rec}\ e\ \Omega \end{cases}$$

$$CompEnv(\Omega) = \begin{cases} [] & \Omega = [] \\ CompVal\ v \cdot CompEnv\ \Omega' & \Omega = [v] \cdot \Omega' \end{cases}$$

Y el desarrollo en Coq correspondiente es:

```
(**
 * Función que define la compilación de valores del lenguaje
 * con índices de De Bruijn a valores de la máquina SECD
 * moderna.
 *
 **)
Fixpoint compileval (v:MMLDBval) : mval  :=
match v with
| NumI i => MInt i
| BoolI b => MBool b
```

```
| ClosI e Om => let fix compenv (O:MMLDBenv) :=
                    match O with
                     | v1::Os => compileval v1::compenv Os
                     | nil => nil
                    end
                    in MClos (compile e ++ (IRet::nil)) (compenv Om)
| ClosrI e Om => let fix compenv (O:MMLDBenv) :=
                   match O with
                     | v1::Os => compileval v1::compenv Os
                     | nil => nil
                   end
                in MClosr (compile e ++ (IRet::nil)) (compenv Om)
end.

(**
 * Función que define la compilación de entornos de la semántica
 * del lenguaje con índices de De Bruijn a entornos de la máquina
 * SECD moderna.
 *
 **)
Fixpoint compileev (O:MMLDBenv): menv :=
match O with
| v :: Os => compileval v :: compileev Os
| nil => nil
end.
```

Presentamos ahora algunos ejemplos simples de esta traducción en Coq.

Veamos por ejemplo que a un valor `NumI 10` de la semántica del lenguaje con índices de De Bruijn le corresponde el valor de máquina `MInt 10`

```
Compute(compileval (NumI 10)).
     = MInt 10
     : mval
```

Por otro lado al entorno `NumI 5::NumI 6::NumI 7::nil` de la semántica con índices de De Bruijn, le corresponde el entorno `MInt 5 :: MInt 6 :: MInt 7 :: nil` de la máquina, como se muestra a continuación:

```
Compute(compileev (NumI 5::NumI 6::NumI 7::nil)).
     = MInt 5 :: MInt 6 :: MInt 7 :: nil
     : menv
```

y a la cerradura `(ClosI (PlusI (ConstI 5) (ConstI 7)) (NumI 4::BoolI true::nil))` de la semántica del lenguaje con notación de De Bruijn le corresponde la cerradura: `MClos (IConst 5 :: IConst 7 :: IAdd :: IRet :: nil) (MInt 4 :: MBool true::nil)` de la máquina, como se puede observar a continuación:

```
Compute (compileval ((ClosI (PlusI (ConstI 5) (ConstI 7))
                            (NumI 4::BoolI true::nil)))).
     = MClos (IConst 5 :: IConst 7 :: IAdd :: IRet :: nil)
         (MInt 4 :: MBool true :: nil)
     : mval
```

En este punto nos encontramos listos ya para enunciar y demostrar la corrección de la traducción de nuestro lenguaje fuente a índices de De Bruijn, labor que llevaremos a cabo en la siguiente sección.

5.6.2. Corrección

Para formular la corrección de la compilación de nuestro lenguaje fuente en notación de De Bruijn a código de máquina de la SECD moderna, podemos comenzar diciendo intuitivamente que si evaluamos una expresión escrita en notación de De Bruijn y se obtiene un valor como resultado, entonces al compilar la expresión y evaluar el código correspondiente en la máquina, se debe obtener el mismo valor como resultado final. Esto es, más formalmente podemos decir que si una expresión e se evalúa a un valor v en un entorno Ω utilizando la semántica con índices de De Bruijn de la máquina, en notación

$$\Omega \vdash e \rightarrow\!\triangleright\, v$$

entonces por una parte si compilamos e y se obtiene como resultado el código c, en notación $Compile(e) = c$ y por otra si compilamos el valor v y se obtiene como resultado el valor de máquina v_m, en notación $CompVal(v) = v_m$ y por otra más si compilamos el entorno Ω y se obtiene como resultado el entorno de máquina Δ, en notación $CompEnv(\Omega) = \Delta$, entonces al evaluar c en la máquina partiendo del entorno Δ se debe obtener como resultado (valor en el tope de la pila al finalizar la evaluación) v_m, en notación

$$(c, \Delta, s) \rightarrow^* ([], \Delta, [\text{SVal } v_m] \cdot s)$$

aquí como podemos observar hemos utilizado la semántica de paso pequeño para denotar la evaluación en la máquina. También podemos utilizar la semántica de paso grande con ese fin, en tal caso lo anterior se denotaría como

$$\Delta, s \vdash c \Rightarrow ([], \Delta, [\text{SVal } v_m] \cdot s)$$

Notemos entonces que podemos expresar la corrección de la compilación utilizando la semántica de paso pequeño o la semántica de paso grande. Por otro lado, mostramos ya

la equivalencia entre estas dos semánticas, por lo que en principio podemos utilizar cualquiera de ellas para demostrar la corrección, sin embargo con fines ilustrativos daremos un teorema de corrección utilizando cada una de ellas.

Damos pues a continuación el teorema (uno para cada semántica de la máquina) que enuncia formalmente la corrección de la compilación de nuestro lenguaje fuente con notación de De Bruijn a código de máquina de la SECD moderna.

Primero utilizamos semántica de paso pequeño.

Teorema 5.6.1. Para toda expresión e, para todo valor v, para todo entorno Ω, para todo código c, para todo entorno de máquina Δ, para toda pila s, para todo valor de máquina v_m. Si

$$\Omega \vdash e \twoheadrightarrow v$$

entonces,

$$(c, \Delta, s) \rightarrow^* ([], \Delta, [\text{SVal } v_m] \cdot s)$$

donde $c = Compile\ e$, $v_m = CompVal\ v$ y $\Delta = CompEnv\ \Omega$.

Ahora utilizamos la semántica de paso grande.

Teorema 5.6.2. Para toda expresión e, para todo valor v, para todo entorno Ω, para todo código c, para todo entorno de máquina Δ, para toda pila s, para todo valor de máquina v_m. Si

$$\Omega \vdash e \twoheadrightarrow v$$

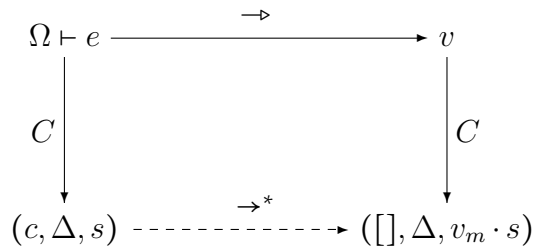
entonces,

$$\Delta, s \vdash c \Rightarrow ([], \Delta, [\text{SVal } v_m] \cdot s)$$

donde $c = Compile\ e$, $v_m = CompVal\ v$ y $\Delta = CompEnv\ \Omega$.

Podemos representar cada uno de estos teoremas mediante un diagrama como lo hacemos a continuación.

Primero presentamos el diagrama correspondiente al teorema de corrección que utiliza la semántica de paso pequeño.



En este diagrama las flechas sólidas representan a las hipótesis y la flecha segmentada la proposición a demostrar. Por otro lado la letra C denota compilación.

Con base en lo dicho, podemos leer el diagrama anterior de la siguiente manera. Si se tiene por una parte que

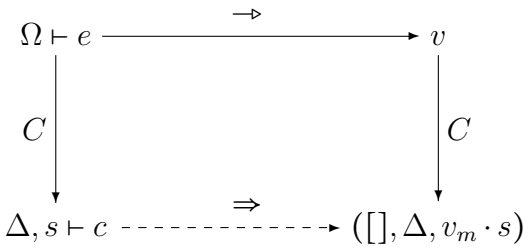
$$\Omega \vdash e \twoheadrightarrow v$$

(lo que corresponde a la línea sólida superior), luego, por otra que $CompEnv(\Omega) = \Delta$ y que $Compile(e) = v$ (lo que corresponde a la línea sólida hacia abajo de la izquierda, porque como dijimos C denota compilación) y por otra parte más que $CompVal(v) = v_m$ (que corresponde a la línea sólida de la derecha) entonces se debe demostrar que

$$(c, \Delta, s) \rightarrow^* ([], \Delta, [\text{SVal } v_m] \cdot s)$$

(que es lo correspondiente a la línea segmentada inferior).

Ahora presentamos el diagrama correspondiente al teorema de corrección que utiliza la semántica de paso grande de la máquina.



Debemos decir en este punto que para poder demostrar nuestro teorema de corrección es necesario primero fortalecer nuestra hipótesis (de la misma manera en que lo hicimos en la sección 3.7 para nuestro compilador de expresiones aritméticas).

Por lo que a continuación presentaremos nuestro teorema de corrección fortalecido.

Primero lo presentamos utilizando la semántica de paso pequeño.

Teorema 5.6.3. Para toda expresión e, para todo valor v, para todo entorno Ω, para todo código c, d, para todo entorno de máquina Δ, para toda pila s, para todo valor de máquina v_m. Si

$$\Omega \vdash e \twoheadrightarrow v$$

entonces,

$$(c \cdot d, \Delta, s) \rightarrow^* (d, \Delta, [\text{SVal } v_m] \cdot s)$$

donde $c = Compile\ e$, $v_m = CompVal\ v$ y $\Delta = CompEnv\ \Omega$.

Ahora lo presentamos utilizando la semántica de paso grande.

Teorema 5.6.4. Para toda expresión e, para todo valor v, para todo entorno Ω, para todo código c, d, para todo entorno de máquina Δ, para toda pila s, para todo valor de máquina v_m. Si

$$\Omega \vdash e \twoheadrightarrow v$$

entonces,

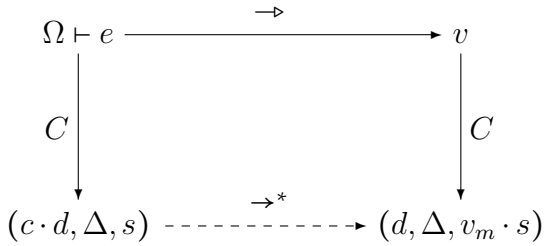
$$\Delta, s \vdash c \cdot d \Rightarrow (d, \Delta, [\text{SVal } v_m] \cdot s)$$

donde c = $Compile\ e$, v_m = $CompVal\ v$ y Δ = $CompEnv\ \Omega$.

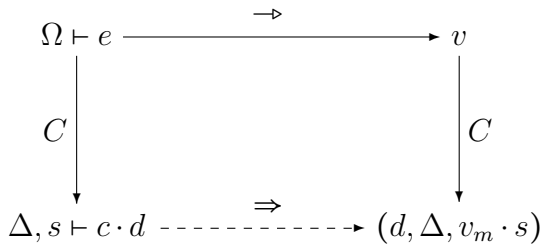
Como podemos observar, intuitivamente lo que hemos hecho es poner un código d cualquiera al final de c, de esta manera cuando se finalice la evaluación de c en lugar de que quede el código vacío quedará d. Nótese que si tomamos d = $[\,]$ obtenemos nuestro teorema como lo formulamos originalmente, esto es, nuestro teorema de corrección original en realidad es un corolario de nuestro teorema de corrección fortalecido.

Ahora presentamos los diagramas correspondientes al teorema de corrección fortalecido.

Presentamos primero el diagrama en el que se utiliza la semántica de paso pequeño.



Ahora presentamos el diagrama en el que se hace uso de la semántica de paso grande.



El teorema de corrección primero utilizando la semántica de paso pequeño y luego la semántica de paso grande se expresa en Coq de la siguiente manera:

```
(**
 * Teorema de corrección de la compilación del lenguaje fuente
 * con notación de De Bruijn a código de la máquina SECD moderna.
 * Este teorema utiliza (la cerradura transitiva y reflexiva de) la
 * semántica de paso pequeño de la máquina.
 *
 **)
```

```
Theorem CorrecCSSS:
forall O:MMLDBenv, forall e:MMLDBexp, forall v:MMLDBval,
BSSMMLDB O e v ->
forall d:code, forall s:stack,
TRCSSSSEC (compile e ++ d, compileev O, s)
(d, compileev O,SVal (compileval v)::s).

(**
 * Teorema de corrección de la compilación del lenguaje fuente
 * con notación de De Bruijn a código de la máquina SECD moderna.
 * Este teorema utiliza la semántica de paso grande de la máquina.
 *
 **)
Theorem CorrecCBSS:
forall O:MMLDBenv, forall e:MMLDBexp, forall v:MMLDBval,
BSSMMLDB O e v ->
forall d:code, forall s:stack,
BSSSEC (compile e ++ d, compileev O, s)
(d,compileev O,SVal (compileval v)::s).
```

Con esto concluimos la corrección de nuestro compilador.

Ahora podemos utilizar el mecanismo de extracción para obtener un compilador correcto verificado de nuestro lenguaje fuente en notación de De Bruijn a la máquina SECD moderna. Este compilador correcto verificado obtenido estará escrito en un lenguaje de programación convencional que es en este caso OCaml y estará listo para ser usado en la vida real. A continuación se muestra cómo obtenerlo.

```
Extraction compile.

(** val compile : mMLDBexp -> code **)

let rec compile = function
| ConstI n -> Cons ((IConst n), Nil)
| PlusI (e1, e2) -> app (compile e1) (app (compile e2) (Cons (IAdd, Nil)))
| MinusI (e1, e2) -> app (compile e1) (app (compile e2) (Cons (ISub, Nil)))
| TimesI (e1, e2) -> app (compile e1) (app (compile e2) (Cons (IMul, Nil)))
| EqI (e1, e2) -> app (compile e1) (app (compile e2) (Cons (IEq, Nil)))
| ConstbI b -> Cons ((IConstb b), Nil)
| VarI n -> Cons ((IAcc n), Nil)
| LetmI (e1, e2) ->
  app (compile e1) (Cons (ILet, (app (compile e2) (Cons (IELet, Nil)))))
| IfI (e1, e2, e3) ->
  app (compile e1) (Cons ((ISel ((app (compile e2) (Cons (IJoin, Nil))),
    (app (compile e3) (Cons (IJoin, Nil))))), Nil))
| LamI lexp -> Cons ((IClos (app (compile lexp) (Cons (IRet, Nil)))), Nil)
| MuI mexp -> Cons ((IClosr (app (compile mexp) (Cons (IRet, Nil)))), Nil)
```

```
| AppI (e1, e2) -> app (compile e1) (app (compile e2) (Cons (IApp, Nil)))
```

Por último ilustramos cómo hemos alcanzado nuestro objetivo principal, utilizando nuestro ejemplo motivacional:

```
(**
 * Ejemplo cálculo de compilación del lenguaje fuente
 * (con notación de nombres) al código de máquina
 * de la SECD moderna y, ejecución de este código
 * utilizando el intérprete "mexecms" el cual simula
 * la cerradura transitiva y reflexiva de la semántica
 * de paso pequeño de la máquina.
 **)
Definition CompAndExSSS (e:MMLexp) :=
match NtoDBfwoBR nil e with
| Some ei => mexecms 75 ((compile ei),nil,nil)
| None => None
end.

Compute (CompAndExSSS
(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))).
     = Some (nil, nil, SVal (MInt 120) :: nil)
     : option mconf

(**
 * Ejemplo cálculo de compilación del lenguaje fuente
 * (con notación de nombres) al código de máquina
 * de la SECD moderna y, ejecución de este código
 * utilizando el intérprete "mexecrec" el cual sigue la
 * semántica de paso grande de la máquina.
 **)
Definition CompAndExBSSS (e:MMLexp) :=
match NtoDBfwoBR nil e with
| Some ei => mexecrec 75 ((compile ei),nil,nil)
| None => None
end.

Compute (CompAndExBSSS
(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))).
= Some (nil, nil, SVal (MInt 120) :: nil)
: option mconf
```

Es decir, escribimos nuestro ejemplo motivacional utilizando nombres y nuestro compilador se encarga de traducirlo a índices de De Bruijn, después de compilarlo al código de la máquina SECD moderna y por último lo puede ejecutar en cualquier de los dos intérpretes que tenemos para nuestra máquina SECD moderna: el que cumple con la cerradura transitiva de la semántica de paso pequeño: `mexecms` o el que cumple con la semántica de paso grande de la máquina: `mexecrec`. En ambos casos obtenemos el resultado esperado, es decir, el factorial de 5 es 120.

Nótese que el mismo ejemplo anterior lo podemos realizar utilizando el compilador correcto verificado escrito en lenguaje OCaml (correspondiente a cada una de las etapas de nuestro compilador) que obtuvimos al utilizar el mecanismo de extracción durante el desarrollo del presente capítulo. Es decir, en lugar de realizar el ejemplo anterior dentro de Coq, lo podemos realizar de manera totalmente independiente de Coq ejecutando nuestro compilador correcto verificado escrito en OCaml como si se tratase de cualquier otro programa en la vida real. Lo mismo vale para los intérpretes de la máquina.

Por otro lado, obsérvese cómo hemos verificado la corrección de nuestro compilador. Más aún, debido a nuestro enfoque computacional, contamos con un compilador que se puede utilizar en la vida real para compilar programas. Todavía más, contamos con un intérprete verificado de la máquina[24] en el cuál podemos ejecutar el programa compilado. Desde luego esto es un resultado óptimo para nuestros propósitos.

Con esto damos por concluido todo el desarrollo de nuestro compilador no sin antes mencionar que hemos quedado completamente satisfechos por haber logrado nuestros objetivos de manera óptima. Para finalizar en el siguiente capítulo presentaremos nuestras conclusiones.

[24] En realidad con dos.

CAPÍTULO 6

Conclusiones

Partimos de la necesidad de desarrollar un compilador correcto y lo obtuvimos. En el camino pudimos darnos cuenta de los aspectos necesarios implicados en el desarrollo de un compilador y sus posibles diseños y arquitecturas. También revisamos algunos de los criterios más importantes que pueden ser tomados como base para elegir algunas de las características de un compilador.

Adicionalmente pudimos darnos cuenta de los pasos necesarios para llevar a cabo la verificación de la corrección de un compilador, en particular cómo ciertos detalles teóricos se reflejan en detalles técnicos al utilizar un asistente de pruebas, herramienta que usamos para realizar la verificación de nuestro compilador.

En concreto elegimos realizar un compilador correcto de Mini-ML a la máquina SECD moderna. Para el desarrollo y la verificación de la corrección de éste utilizamos Coq.

Elegimos Mini-ML con base en nuestro ejemplo motivacional de la sección 1.1 y la máquina SECD porque es un máquina simple y clara, la cual nació como la mecanización de una estrategia de evaluación del cálculo lambda utilizado como lenguaje de programación. Junto con ella nacieron conceptos fundamentales de la programación funcional como entornos y cerraduras, lo que la hace fácil de entender e ideal como base para compararla con otras máquinas (sean virtuales o reales) que pudieran ser más eficientes.

Nuestro compilador lo desarrollamos para contar con un compilador correcto verificado que sirva como base para futuras extensiones, siendo una de las más relevantes dar soporte a un lenguaje completo que se use en la vida real como ML o OCaml. Así se podría utilizar en producción con la certeza que el compilador no introducirá errores en los ejecutables que produzca, algo sin duda deseable.

Cabe mencionar que ML cuenta con una semántica definida formalmente la cual se puede consultar dentro de su especificación [MTM97], lo que lo hace un candidato ideal (y representa una ventaja respecto de otros lenguajes de programación) para poder desarrollar un compilador correcto de este lenguaje utilizando un asistente de pruebas. Pues

como sabemos para poder enunciar la corrección de un compilador en un asistente de pruebas es necesario contar con (al menos) una semántica del lenguaje fuente definida formalmente.[1]

Habiendo elegido nuestro lenguaje fuente, la máquina objetivo y el asistente de pruebas, buscamos el trabajo relacionado desarrollado en Coq.

El trabajo más cercano a nuestro objetivos es el realizado por Samuel Boutin [Bou95]. Boutin desarrolla un compilador correcto de Mini-ML[2] a la CAM en Coq. Sin embargo hay varias razones por las que no tomamos como punto de partida su trabajo para hacer extensiones a éste:

1. Boutin en [Bou95] enuncia lemas y el teorema de corrección a probar en Coq y se da por hecho que realizó las demostraciones de éstos. Sin embargo no está disponible[3] el desarrollo completo de su trabajo, en particular no están disponibles las demostraciones.

2. Su desarrollo lo realizó en la versión 5.10 de Coq (la actual es la 8.4), así que aún cuando estuviera disponible su código no sería posible utilizarlo con la versión actual de Coq, pues desde entonces se han hecho cambios importantes en Coq que hacen que el código escrito para la versión 5.10 no sea compatible con la versión actual. En particular se han hecho cambios a la sintaxis.

3. Boutin sigue un enfoque matemático (ver sección 3.7), es decir, en [Bou95] únicamente presenta versiones sin contenido computacional (del lenguaje, de la máquina y del compilador), por lo que no cuenta con un compilador que sirva para calcular la traducción de programas en la vida real.

4. El utilizar la CAM como máquina objetivo hace que en principio sea menos claro de entender. En particular como mencionamos en la sección 5.5, él hace una traducción directa de una semántica basada en nombres a la CAM, y la CAM utiliza implícitamente índices de De Bruijn, por lo que esta traducción tiene implícita una traducción a índices de De Bruijn al tiempo que genera código. Por tanto esa traducción es compleja y difícil de entender a primera vista.

No obstante, como parte de este trabajo tomamos la definiciones, los lemas y el teorema de corrección ofrecido por Boutin en [Bou95] y escribimos su equivalente en la versión actual de Coq y luego desarrollamos demostraciones propias de los lemas y el teorema de corrección. El código fuente correspondiente al desarrollo completo de este compilador se encuentra disponible en [Zúñ15a], resolviendo de esta manera los inconvenientes 1. y 2., es decir, poniendo al alcance de cualquiera el desarrollo completo en la versión actual

[1] Al parecer ML es el único lenguaje de amplio uso que cuenta con una especificación en la que se define formalmente su semántica.

[2] Un subconjunto de ML muy parecido al nuestro pero no exactamente el mismo.

[3] Se realizó una búsqueda exhaustiva en internet y no se encontró.

de Coq. Como posible trabajo futuro, lo natural es realizar verificación para ofrecer versiones con contenido computacional y así obtener la visión computacional deseada. De esta forma se resolvería el inconveniente 3..

Es el punto número 4. el que difiere de nuestros objetivos, pues como dijimos buscamos simpleza y claridad y al utilizar la CAM como máquina objetivo no podemos asegurar que se cuente con ellas, por eso (entre otras cosas más), nosotros elegimos la SECD como máquina objetivo.

Y nos encontramos con que Leroy en [Lerb] ofrece un desarrollo en Coq de un compilador correcto verificado de un lenguaje mínimo, que consta de variables, declaraciones locales, abstracciones y aplicación; dicho lenguaje utiliza notación de De Bruijn. Como máquina objetivo se usa la máquina SECD moderna (que como mencionamos en la sección 5.5, hace uso de índices de De Bruijn y de marcos de pila). Este desarrollo justo coincidía con nuestros objetivos y por tanto era ideal tomarlo como punto de partida para nuestro trabajo y así lo hicimos.

La semántica que brinda Leroy en [Lerb] del lenguaje es una semántica con entornos e índices de De Bruijn que en efecto coincidía con nuestros propósitos. Por su parte la semántica que ofrece de la SECD moderna es una semántica de paso pequeño. En [Lerb] únicamente la compilación del lenguaje a la SECD cuenta con contenido computacional. Por tanto nuestras aportaciones consisten en lo siguiente:

1. Extendimos el soporte del lenguaje a todo nuestro lenguaje fuente que presentamos en la sección 5.2. Esto es, agregamos constantes booleanas, operadores aritméticos y booleanos de comparación, enunciados condicionales If y recursión nativa por medio del operador de punto fijo μ.

2. Agregamos contenido computacional a la semántica con entornos e índices de De Bruijn, para obtener así de esta manera un intérprete del lenguaje basado en índices de De Bruijn.

3. Agregamos contenido computacional a la semántica de paso pequeño de la máquina[4] obteniendo así un intérprete.

4. Formulamos una semántica de paso grande de la máquina y desarrollamos en Coq, por un lado, su correspondiente versión relacional y, por el otro, su correspondiente versión funcional con contenido computacional, obteniendo en este último caso un intérprete de la máquina (diferente).[5]

5. Realizamos una demostración en Coq de la equivalencia entre las dos semánticas de la máquina: la de paso pequeño y la de paso grande.

[4]Para ser más precisos, un intérprete de un paso correspondiente a la semántica de paso pequeño de la máquina y uno de muchos pasos que cumple con la cerradura transitiva y reflexiva de la misma semántica (tal como lo explicamos en la sección 5.5.1).

[5]Esta vez uno que cumple con la semántica de paso grande de la máquina.

6. Como es insensato pedir que un programador escriba un programa utilizando la notación de De Bruijn, agregamos una etapa a nuestro compilador que se encarga de realizar la traducción del lenguaje utilizando la notación normal basada en nombres a índices de De Bruijn. La traducción la explicamos en la sección 4.1.2 y está inspirada en el trabajo de Pientka [Pie13]. En Coq ofrecimos su correspondiente versión relacional y probamos mediante un lema que ésta es determinista y, por otro lado, desarrollamos diferentes versiones con contenido computacional (una más clara, otra más eficiente y otra un compromiso entre ambas) y de esta manera obtuvimos un medio para calcular esta traducción.

7. Para poder realizar el punto anterior, antes tuvimos que dar una semántica de paso grande basada en nombres y entornos del lenguaje. Aquí también dimos su correspondiente versión relacional y una funcional con contenido computacional, obteniendo así un intérprete del lenguaje con notación de nombres.

8. Enunciamos y demostramos en Coq un teorema de corrección de la traducción a índices de De Bruijn.

9. Enunciamos y dimos una demostración del teorema de corrección de la compilación del lenguaje en notación de De Bruijn a la máquina SECD moderna utilizando la semántica de paso pequeño de la máquina.[6]

10. Enunciamos y dimos una demostración del teorema de corrección de la compilación del lenguaje en notación de De Bruijn a la máquina SECD moderna utilizando la nueva semántica de paso grande que formulamos de la máquina.

De esta manera, obtuvimos un compilador verificado correcto capaz de compilar cualquier programa escrito en nuestro lenguaje fuente. Pero más aún, obtuvimos (entre otras cosas más) un intérprete[7] de la máquina en donde se puede ejecutar el código compilado tal como lo ilustramos con nuestro ejemplo motivacional (de la sección 1.1) al final del capítulo anterior.

Con esto hemos cumplido con nuestros objetivos, pues contamos ahora con un compilador correcto, que nos permitió conocer y comprender los pasos necesarios para desarrollarlo y cómo realizarlos. Este compilador además sirve como base para experimentar y para futuras extensiones.

Hay muchos caminos a seguir como posibles extensiones del presente trabajo. A continuación mencionamos brevemente algunos de éstos.

En primer lugar, en este trabajo dimos por concedido que se han realizado ya las etapas de análisis léxico, sintáctico y semántico. Entonces, lo natural sería implementar estas etapas para obtener un compilador completo de inicio a fin.

En seguida mencionamos el estado actual de la verificación de la corrección de cada una de estas etapas en asistentes de pruebas:

[6] Aquí extendimos el teorema de corrección dado por Leroy en [Lerb].

[7] En realidad contamos con dos intérpretes de la máquina y podemos hacer uso de cualquiera de ellos.

- Análisis léxico: aquí lo ideal sería contar con un generador de analizadores léxicos correcto verificado, es decir, contar con una herramienta como *flex* (del cual ilustramos su uso en nuestro compilador de ejemplo en la sección 2.1.1) pero que contara con la verificación de su corrección. De esta manera, los analizadores léxicos que generara serían en automático analizadores léxicos correctos. Desafortunadamente no existen al día de hoy generadores de analizadores léxicos correctos verificados, aunque existen algunos pocos trabajos en ese sentido, por ejemplo [Nip98].

- Análisis sintáctico: este es un caso similar al anterior. Esta vez quisiéramos contar con generadores de analizadores sintácticos como *bison* (ver la sección 2.1.2) pero que éstos fueran correctos, así generarían analizadores sintácticos correctos. En la actualidad no existen generadores de analizadores sintácticos correctos verificados disponibles para su uso por cualquiera (como es el caso de *bison*), aunque existen varios trabajos cercanos a lograr este objetivo [JPL12; Bar11; BN09].

- Análisis semántico: Como nuestro lenguaje fuente es un subconjunto de ML es un lenguaje estática e implícitamente tipificado, lo que significa que el principal trabajo de nuestro analizador semántico es realizar el algoritmo de inferencia de tipos de ML, es decir, se tendría que desarrollar una verificación de la corrección del algoritmo de inferencia de tipos. En este caso existen diversos trabajos con este fin ([Dub00; Gar10] son algunos de éstos en los que se utiliza Coq como asistente de pruebas), pero la mayoría de estos trabajos consideran la verificación de la corrección del algoritmo de inferencia de tipos como un problema en sí mismo y no están explícitamente desarrollados para que este algoritmo se utilice en un compilador como en el nuestro, por lo que haría falta investigar si se puede utilizar alguno de estos trabajos en nuestro compilador o, en su defecto desarrollar una verificación de la corrección del algoritmo de inferencia de tipos para usarla específicamente en nuestro compilador.

Por otra parte como mencionamos en la sección 4.2, se podría demostrar como trabajo futuro la equivalencia entre la semántica que utiliza sustituciones y la semántica con entornos y cerraduras que presentamos en la sección 4.1.3.

Como mencionamos en la sección 5.5, para ofrecer soporte nativo de recursión se puede hacer uso de cerraduras recursivas o de entornos recursivos.[8] De estas soluciones nosotros utilizamos cerraduras recursivas. Entonces, por otro lado se podría hacer uso de entornos recursivos y luego demostrar que ambas soluciones son equivalentes.

Contando ya con un máquina clara, simple y fácil de entender y su correspondiente compilador, se podría ahora buscar moverse a una máquina más eficiente en vías de obtener un compilador que se pueda utilizar en producción. Así, una vez postulada la máquina y su correspondiente compilador podríamos probar la equivalencia entre ambos compiladores, esto es, dado un programa p, si p se compila en nuestro compilador y

[8]Estamos hablando en el contexto de la formalización de recursión nativa dentro de un asistente de pruebas.

se ejecuta en la máquina SECD moderna y da como resultado v, entonces, si se compila p en el nuevo compilador y se ejecuta en la nueva máquina debe dar como resultado v y viceversa, de esta manera tendríamos la certeza de ir en el camino correcto. Por ejemplo, como primer paso podríamos probar la equivalencia de los dos compiladores con los que contamos, el de la máquina SECD y el que desarrollamos para la CAM con base en el trabajo de Boutin. La CAM se utilizó en las primeras versiones del lenguaje Caml por tanto con esto estaríamos más cerca de llegar a un compilador en producción. Como segundo paso, podríamos formalizar la máquina ZAM de Leroy [Ler90], máquina que se utiliza en la implementación actual de OCaml.[9] Por otra parte también, en lugar de generar código para la máquina SECD podríamos ahora generar código de máquina nativo, por ejemplo para la arquitectura x86, x64 o SPARC ya sea generando código directamente para alguna de éstas o expandiendo el código de la SECD a alguna de éstas. De este modo también estaríamos más cerca de un compilador en producción.

Otro aspecto fundamental que se puede atacar es la concurrencia. Existen varios modelos y primitivas de concurrencia que se han ido desarrollando durante el paso del tiempo y los diferentes lenguajes de programación han ido adoptando y adaptando algunos de ellos. Dentro de la programación funcional (entre otros) uno de los modelos de concurrencia más desarrollado es el de CML (*Concurrent ML*) [Rep92; Rep07] que agrega soporte de concurrencia al lenguaje ML. Sorpresivamente el terreno de la verificación de la corrección de compiladores de lenguajes con soporte de concurrencia en asistentes de prueba es casi nulo. Por tanto, es ideal tomar como base el presente trabajo para extenderlo con soporte de concurrencia y resulta natural tomar como modelo de concurrencia el de CML ya que nuestro lenguaje fuente es Mini-ML.[10] Dos de los aspectos más importantes que se deben considerar en vías de lograr ese objetivo son:

1. desarrollar una semántica que permita expresar concurrencia tanto para el lenguaje fuente como para la máquina, tomando en cuenta que dicha semántica debe ser susceptible de formalizar en un asistente de pruebas

2. diseñar y desarrollar una máquina concurrente.

Es del interés del autor desarrollar este objetivo en un trabajo futuro.

Para finalizar podemos describir brevemente el estado actual de la corrección de compiladores. La corrección de compiladores comenzó a ser considerada desde hace tiempo pues ya McCarthy [MP67] en el año 1967 presenta una demostración en papel y lápiz de la corrección de un compilador para un lenguaje de expresiones aritméticas. Luego en 1972 Milner [MW72] desarrolló un compilador de un subconjunto de Algol a una máquina basada en pila y verificó su corrección en el asistente de pruebas LCF.

Desde entonces se han desarrollado diversos compiladores correctos, pero siendo casi la totalidad de éstos de pequeños lenguajes con fines académicos o de investigación. En

[9] El compilador actual de OCaml puede generar dos posibles salidas: una es generar código de máquina de la ZAM y la otra es generar código de máquina nativo.

[10] Aunque también se podrían considerar otros modelos de concurrencia.

[Dav03] se presenta un conjunto de algunos de estos trabajos junto con sus respectivas referencias hasta el 2003.

En la actualidad quizás el proyecto más influyente y con mayor alcance en el área de verificación de compiladores es el proyecto CompCert [Lera] dentro del cual se desarrolla el compilador CompCert C [Ler06; BDL06; Ler09a; Ler09b], un compilador correcto verificado del lenguaje C. CompCert lo desarrolla un equipo del INRIA dirigido por Xavier Leroy utilizando el asistente de pruebas Coq.

El compilador Compcert C representó un avance significativo en el área de la verificación de la corrección de compiladores, pues hasta entonces como se mencionó únicamente se habían desarrollado pequeños compiladores con fines académicos y de investigación, en cambio CompCert C se diseño y se desarrolló teniendo en cuenta como objetivo que fuera utilizado ampliamente en producción.

En cuanto a la corrección de compiladores de lenguajes funcionales son escasos los trabajos en esta área. Algunos de éstos desarrollados en Coq[11] son Chlipala [Chl07] y Dargaye [Dar09]. Pero ninguno de estos últimos está diseñado para ser utilizado en sí mismo como una compilador de un lenguaje funcional de amplio uso en producción sino que tienen objetivos muy específicos diferentes a éste. Contar con un compilador correcto de un lenguaje funcional de amplio uso producción sigue siendo un objetivo por alcanzar.

Con esto damos por concluido el presente trabajo.

[11] Además de los ya mencionamos anteriormente de Boutin y Leroy.

APÉNDICE

Definición formal de ocurrencia

Definición .1 (Posición)**.** Una *posición* $p = i_1 \dots i_m$ es cualquier cadena finita (posiblemente vacía) de símbolos tal que $i_1, \dots, i_{m-1}$ son enteros y i_m es un entero o un asterisco $*$. Su *longitud* es m, y si $m = 0$ decimos que $p = \varnothing$.

- Si $m \geq 1$ y $i_m = 1$ llamamos a p una posición función;
- si $m \geq 1$ y $i_m = 2$ llamamos a p una posición argumento;
- si $m \geq 1$ y $i_m = 0$ llamamos a p una posición cuerpo y
- si $m \geq 1$ y $i_m = *$ llamamos a p una posición abstractor.

La *concatenación* pq de posiciones $p = i_1 \dots i_m$ y $q = j_1 \dots j_n$ se define como: $p\varnothing = p$, $\varnothing q = q$, y si $m, n \geq 1$ y $i_m \neq *$, definimos

$$pq = i_1 \dots i_m j_1 \dots j_n.$$

(pq no está definida si $m, n \geq 1$ y $i_m = *$).

Definición .2 (Ocurrencia)**.** El enunciado "P ocurre en M en la posición p" (o "M contiene a P en la posición p") se define por inducción sobre la longitud de p, de la siguiente manera:

1. M ocurre en M en la posición $\varnothing$;
2. si $P_1 P_2$ ocurre en M en p, entonces P_i ocurre en M en pi para $i = 1, 2$;
3. Si $\lambda x.Q$ ocurre en M en p, entonces Q ocurre en M en $p0$ y x ocurre en M en $p*$.

Una *ocurrencia* de P en M es una tupla $\langle P, p, M\rangle$ tal que P ocurre en M en p. Si $P \equiv x$ y el último símbolo de p es $*$ nombramos a $\langle x, p, M\rangle$ una ocurrencia ligadora (*binding ocurrence*) de x o una ocurrencia de λx. Un *abstractor* es una ocurrencia de λx para alguna x. Un *subtérmino* de M es cualquier término de P que ocurre en M.

Bibliografía

[AGN09] Andrea Asperti, Herman Geuvers y Raja Natarajan. «Social Processes, Program Verification and All That». En: *Mathematical. Structures in Comp. Sci.* 19.5 (oct. de 2009), págs. 877-896. ISSN: 0960-1295. DOI: 10.1017/S0960129509990041. URL: http://dx.doi.org/10.1017/S0960129509990041 (vid. pág. 50).

[Aho+06] Alfred V. Aho y col. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811 (vid. pág. 22).

[Bac78] John Backus. «Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs». En: *Commun. ACM* 21.8 (ago. de 1978), págs. 613-641. ISSN: 0001-0782. DOI: 10.1145/359576.359579. URL: http://doi.acm.org/10.1145/359576.359579 (vid. pág. 2).

[Bal02] Antonia Balaa. «Fonctions récursives générales dans le calcul des constructions». Tesis doct. INRIA - Sophia Antipolis, 2002 (vid. pág. 99).

[Bar11] Aditi Barthwal. «A formalisation of the theory of context-free languages in higher order logic». Tesis doct. Canberra, Australia: College of Engineering y Computer Science, ANU, jul. de 2011 (vid. pág. 201).

[Bar12] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Vol. 40. Studies in Logic. College Publications, 2012. ISBN: 184890066X (vid. págs. 2, 42, 122, 123).

[Bar92] H. P. Barendregt. «Handbook of Logic in Computer Science (Vol. 2)». En: ed. por S. Abramsky, Dov M. Gabbay y S. E. Maibaum. New York, NY, USA: Oxford University Press, Inc., 1992. Cap. Lambda Calculi with Types, págs. 117-309. ISBN: 0-19-853761-1. URL: http://dl.acm.org/citation.cfm?id=162552.162561 (vid. pág. 47).

[BC04] Yves Bertot y Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. ISBN: 3540208542 (vid. pág. 8).

[BDL06] Sandrine Blazy, Zaynah Dargaye y Xavier Leroy. «Formal Verification of a C Compiler Front-End». En: *FM 2006: Int. Symp. on Formal Methods*. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, págs. 460-475. URL: http://gallium.inria.fr/~xleroy/publi/cfront.pdf (vid. pág. 203).

[BG01] Henk Barendregt y Herman Geuvers. «Handbook of Automated Reasoning». En: ed. por Alan Robinson y Andrei Voronkov. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 2001. Cap. Proof-assistants Using Dependent Type Systems, págs. 1149-1238. ISBN: 0-444-50812-0. URL: http://dl.acm.org/citation.cfm?id=778522.778527 (vid. pág. 50).

[BN09] Aditi Barthwal y Michael Norrish. «Verified, Executable Parsing». English. En: *Programming Languages and Systems*. Ed. por Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, págs. 160-174. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9_12. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_12 (vid. pág. 201).

[Bou95] Samuel Boutin. *Preuve de correction de la compilation de Mini-ML en code CAM dans le système d'aide à la démonstration COQ*. Research Report RR-2536. Projet COQ. INRIA, Avril de 1995. URL: https://hal.inria.fr/inria-00074142 (vid. págs. 166, 198).

[Bru72] 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 (Proceedings)* 75.5 (1972), págs. 381-392. ISSN: 1385-7258. DOI: http://dx.doi.org/10.1016/1385-7258(72)90034-0. URL: http://www.sciencedirect.com/science/article/pii/1385725872900340 (vid. pág. 125).

[CCM85] G. Cousineau, P-L. Curien y M. Mauny. «The categorical abstract machine». English. En: *Functional Programming Languages and Computer Architecture*. Ed. por Jean-Pierre Jouannaud. Vol. 201. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1985, págs. 50-64. ISBN: 978-3-540-15975-9. DOI: 10.1007/3-540-15975-4_29. URL: http://dx.doi.org/10.1007/3-540-15975-4_29 (vid. pág. 7).

[CFC58] Haskell B. Curry, Robert Feys y William Craig. *Combinatory Logic*. Vol. I. North-Holland Publising Company, 1958 (vid. págs. 42, 45, 121).

[CH86] T. Coquand y Gérard Huet. *The calculus of constructions*. Inf. téc. RR-0530. INRIA, mayo de 1986. URL: https://hal.inria.fr/inria-00076024 (vid. pág. 7).

[Chl07] Adam Chlipala. «Implementing Certified Programming Language Tools in Dependent Type Theory». Tesis doct. EECS Department, University of California, Berkeley, ago. de 2007. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-113.html (vid. pág. 203).

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN: 0262026651, 9780262026659 (vid. pág. 8).

[Chu32] Alonzo Church. «A Set of Postulates for the Foundation of Logic». English. En: *Annals of Mathematics*. Second Series 33.2 (1932), págs. 346-366. ISSN: 0003486X. URL: http://www.jstor.org/stable/1968337 (vid. pág. 33).

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Vol. 6. Annals of Mathematics Studies. Princeton, NJ, USA: Princeton University Press, 1941 (vid. págs. 2, 122).

[Coc69] John Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 1969. ISBN: B0007F4UOA (vid. pág. 25).

[Dar09] Zaynah Dargaye. «Vérification formelle d'un compilateur optimisant pour langages fonctionnels». Tesis doct. Université Paris 7, 2009 (vid. pág. 203).

[Dav03] Maulik A. Dave. «Compiler Verification: A Bibliography». En: *SIGSOFT Softw. Eng. Notes* 28.6 (nov. de 2003), págs. 2-2. ISSN: 0163-5948. DOI: 10.1145/966221.966235. URL: http://doi.acm.org/10.1145/966221.966235 (vid. pág. 203).

[Del01] David Delahaye. «Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve: une étude dans le cadre du système Coq». Tesis doct. Université Pierre et Marie-Curie (Paris-6), dic. de 2001 (vid. pág. 70).

[DHS00] Stephan Diehl, Pieter Hartel y Peter Sestoft. «Abstract machines for programming language implementation». En: *Future Generation Computer Systems* 16.7 (2000), págs. 739-751. ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/S0167-739X(99)00088-6. URL: http://www.sciencedirect.com/science/article/pii/S0167739X99000886 (vid. pág. 3).

[Dow+93] Gilles Dowek y col. *The Coq proof assistant user's guide : version 5.8*. Research Report RT-0154. INRIA, 1993, pág. 120. URL: https://hal.inria.fr/inria-00070014 (vid. pág. 7).

[Dow13] Gilles Dowek. «Proof in theories». Course Notes. Sep. de 2013. URL: https://who.rocq.inria.fr/Gilles.Dowek/Cours/pit.pdf (vid. pág. 4).

[Dow99] Gilles Dowek. «La Part du Calcul». Mémoire d'Habilitation. Université de Paris 7, juin de 1999. URL: https://who.rocq.inria.fr/Gilles.Dowek/Publi/habilitation.ps.gz (vid. pág. 4).

[DS15] Charles Donnelly y Richard Stallman. *Bison. The Yacc-compatible Parser Generator*. Manual. 2015. URL: http://www.gnu.org/software/bison/manual/bison.pdf (vid. pág. 25).

[Dub00] Catherine Dubois. «Proving ML Type Soundness Within Coq». English. En: *Theorem Proving in Higher Order Logics*. Ed. por Mark Aagaard y John Harrison. Vol. 1869. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, págs. 126-144. ISBN: 978-3-540-67863-2. DOI: 10.1007/3-540-44659-1_9. URL: http://dx.doi.org/10.1007/3-540-44659-1_9 (vid. pág. 201).

[Ear68] Jay Clark Earley. «An Efficient Context-free Parsing Algorithm». AAI6907901. Tesis doct. Pittsburgh, PA, USA, 1968 (vid. pág. 25).

[Gar10] Jacques Garrigue. «A Certified Implementation of ML with Structural Polymorphism». English. En: *Programming Languages and Systems*. Ed. por Kazunori Ueda. Vol. 6461. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, págs. 360-375. ISBN: 978-3-642-17163-5. DOI: 10.1007/978-3-642-17164-2_25. URL: http://dx.doi.org/10.1007/978-3-642-17164-2_25 (vid. pág. 201).

[Gim96] Carlos Eduardo Giménez. «Un calcul de constructions infinies et son application à la vérification de systèmes communicants». Tesis doct. École Normale Supérieure de Lyon, 1996 (vid. pág. 98).

[Gir71] Jean-Yves Girard. «Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types». En: *Proceedings of the Second Scandinavian Logic Symposium*. Ed. por J.E. Fenstad. Vol. 63. Studies in Logic and the Foundations of Mathematics. Elsevier, 1971, págs. 63-92. DOI: http://dx.doi.org/10.1016/S0049-237X(08)70843-7. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08708437 (vid. pág. 7).

[Gir72] Jean-Yves Girard. «Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur». Tesis doct. Paris VII, 1972 (vid. pág. 7).

[GMW79] M.J.C. Gordon, R. Milner y C.P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Lecture notes in computer science. Springer-Verlag, 1979. ISBN: 9783540097242 (vid. pág. 7).

[GTL89] Jean-Yves Girard, Paul Taylor e Yves Lafont. *Proofs and Types*. New York, NY, USA: Cambridge University Press, 1989. ISBN: 0-521-37181-3 (vid. pág. 7).

[Han04] Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. Vol. 2. Texts in Computing. College Publications, 2004 (vid. págs. 2, 123).

[Hen80] Peter Henderson. *Functional Programming Application and Implementation*. Ed. por C. A. R. Hoare. Computer Science. Prentice Hall International, 1980. ISBN: 0-13-331579-7 (vid. pág. 166).

[HS08] J. Roger Hindley y Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. 2.ª ed. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521898854, 9780521898850 (vid. págs. 2, 34, 43, 123).

[HU79] John E. Hopcroft y Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1979. ISBN: 020102988X (vid. pág. 2).

[Hud89] Paul Hudak. «Conception, Evolution, and Application of Functional Programming Languages». En: *ACM Comput. Surv.* 21.3 (sep. de 1989), págs. 359-411. ISSN: 0360-0300. DOI: 10.1145/72551.72554. URL: http://doi.acm.org/10.1145/72551.72554 (vid. pág. 2).

[Hue89] Gerard Huet. «The Constructive Engine». En: *A Perspective in Theoretical Computer Science: Commerative Volume for Gift Siromoney* 16 (1989), pág. 38 (vid. pág. 7).

[Hug89] J. Hughes. «Why Functional Programming Matters». En: *Comput. J.* 32.2 (abr. de 1989), págs. 98-107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. URL: http://dx.doi.org/10.1093/comjnl/32.2.98 (vid. pág. 2).

[Joh75] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. CSTR 32. Murray Hill, NJ: Bell Laboratories, 1975 (vid. pág. 25).

[JPL12] Jacques-Henri Jourdan, François Pottier y Xavier Leroy. «Validating LR(1) Parsers». En: *Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012*. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, págs. 397-416 (vid. págs. 100, 201).

[Kah87] Gilles Kahn. *Natural semantics*. Research Report RR-0601. INRIA, 1987. URL: https://hal.inria.fr/inria-00075953 (vid. págs. 59, 61, 62).

[Kas66] T. Kasami. *An Efficient Recognition and Syntax-analysis Algorithm for Context-Free Languages*. Inf. téc. R-257. Coordinated Science Laboratory. University of Illinois, 1966 (vid. pág. 25).

[Koz97] Dexter C. Kozen. *Automata and Computability*. 1st. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997. ISBN: 0387949070 (vid. pág. 2).

[Lan64] P. J. Landin. «The Mechanical Evaluation of Expressions». En: *The Computer Journal* 6.4 (1964), págs. 308-320. DOI: 10.1093/comjnl/6.4.308. eprint: http://comjnl.oxfordjournals.org/content/6/4/308.full.pdf+html. URL: http://comjnl.oxfordjournals.org/content/6/4/308.abstract (vid. págs. 7, 123, 124, 128, 129, 165, 166).

[Lera] Xavier Leroy. *CompCert*. URL: http://compcert.inria.fr/ (vid. pág. 203).

[Lerb] Xavier Leroy. *Module Compiler*. URL: http://gallium.inria.fr/~xleroy/mpri/twofour/Compiler.html (vid. págs. 199, 200).

[Ler06] Xavier Leroy. «Formal certification of a compiler back-end or: programming a compiler with a proof assistant». En: *ACM SIGPLAN Notices* 41.1 (2006), págs. 42-54 (vid. pág. 203).

[Ler09a] Xavier Leroy. «A formally verified compiler back-end». En: *Journal of Automated Reasoning* 43.4 (2009), págs. 363-446. URL: http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf (vid. pág. 203).

[Ler09b] Xavier Leroy. «Formal verification of a realistic compiler». En: *Communications of the ACM* 52.7 (2009), págs. 107-115. URL: http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf (vid. pág. 203).

[Ler16a] Xavier Leroy. *Functional programming languages Part I: interpreters and operational semantics*. MPRI 2-4. INRIA Paris-Rocquencourt. 2015-2016. URL: http://gallium.inria.fr/~xleroy/mpri/twofour/semantics.pdf (vid. págs. 131, 132).

[Ler16b] Xavier Leroy. *Functional programming languages Part II: abstract machines*. MPRI 2-4. INRIA Paris-Rocquencourt. 2015-2016. URL: http://gallium.inria.fr/~xleroy/mpri/twofour/machines.pdf (vid. págs. 7, 166).

[Ler90] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA, 1990 (vid. págs. 32, 127, 202).

[Les75] M. E. Lesk. *Lex - A Lexical Analyzer Generator*. CSTR 39. Murray Hill, NJ: Bell Laboratories, 1975 (vid. pág. 22).

[Löf71] Per Martin Löf. *A Theory of Types*. Inf. téc. 71-3. Department of Mathematics, University of Stockholm, 1971 (vid. pág. 7).

[LS90] M. E. Lesk y E. Schmidt. «UNIX Vol. II». En: ed. por A. G. Hume y M. D. McIlroy. Philadelphia, PA, USA: W. B. Saunders Company, 1990. Cap. Lex - a Lexical Analyzer Generator, págs. 375-387. ISBN: 0-03-047529-5. URL: http://dl.acm.org/citation.cfm?id=107172.107193 (vid. pág. 22).

[Mar10] John C. Martin. *Introduction to Languages and the Theory of Computation*. 4.ª ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN: 0073191469, 978-0073191461 (vid. pág. 2).

[MP67] John McCarthy y James Painter. «Correctness of a compiler for arithmetic expressions». En: *Mathematical aspects of computer science* 1 (1967) (vid. pág. 202).

[MTM97] Robin Milner, Mads Tofte y David Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814 (vid. págs. 9, 197).

[MW72] Robin Milner y Richard Weyhrauch. «Proving compiler correctness in a mechanized logic». En: *Machine Intelligence* 7 (1972), págs. 51-70 (vid. pág. 202).

[Nip98] Tobias Nipkow. «Verified Lexical Analysis». En: *Theorem Proving in Higher Order Logics*. Ed. por J. Grundy y M. Newey. Vol. 1479. LNCS. Invited talk. Springer, 1998, págs. 1-15 (vid. pág. 201).

[Pau14] Christine Paulin-Mohring. «Introduction to the Calculus of Inductive Constructions». Nov. de 2014. URL: https://hal.inria.fr/hal-01094195 (vid. pág. 8).

[Pau96] C. Paulin-Mohring. «Définitions Inductives en Théorie des Types d'Ordre Supérieur». Habilitation à diriger les recherches. Université Claude Bernard Lyon I, dic. de 1996. URL: http://www.lri.fr/~paulin/PUBLIS/habilitation.ps.gz (vid. pág. 8).

[PE88] F. Pfenning y C. Elliott. «Higher-order Abstract Syntax». En: *SIGPLAN Not.* 23.7 (jun. de 1988), págs. 199-208. ISSN: 0362-1340. DOI: 10.1145/960116.54010. URL: http://doi.acm.org/10.1145/960116.54010 (vid. pág. 131).

[PEM12] Vern Paxson, Will Estes y John Millaway. *Lexical Analysis with Flex*. 2.5.39. Manual. 2012 (vid. pág. 22).

[Pie+15] Benjamin C. Pierce y col. *Software Foundations*. Electronic textbook, 2015. URL: http://www.cis.upenn.edu/~bcpierce/sf (vid. pág. 8).

[Pie13] Brigitee Pientka. «Representing Binders: there is no such thing as being free». School of Computer Science, McGill University. Course notes, CS523. 2013. URL: http://www.cs.mcgill.ca/~bpientka/cs523/handouts/representing-binders.pdf (vid. pág. 200).

[Plo75] G.D. Plotkin. «Call-by-name, call-by-value and the λ-calculus». En: *Theoretical Computer Science* 1.2 (1975), págs. 125-159. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/0304-3975(75)90017-1. URL: http://www.sciencedirect.com/science/article/pii/0304397575900171 (vid. pág. 128).

[Plo81] Gordon Plotkin. «A Structural Approach to Operational Semantics». Computer Science Department, Aarhus University. DAIMI FN-19. 1981 (vid. págs. 53, 55, 59, 60).

[Pra06] Dag Prawitz. *Natural Deduction. A Proof-Theoretical Study*. Dover Plubications, Inc., 1965, 2006 (vid. pág. 4).

[Rep07] John H. Reppy. *Concurrent Programming in ML*. 1st. New York, NY, USA: Cambridge University Press, 2007. ISBN: 0521714729 (vid. pág. 202).

[Rep92] John Hamilton Reppy. «Higher-order Concurrency». UMI Order No. GAX92-11367. Tesis doct. Ithaca, NY, USA: Cornell University, 1992 (vid. pág. 202).

[Sac11] Jorge Luis Sacchini. «On type-based termination and dependent pattern matching in the calculus of inductive constructions». Theses. École Nationale Supérieure des Mines de Paris, jun. de 2011. URL: https://pastel.archives-ouvertes.fr/pastel-00622429 (vid. pág. 99).

[Sai10] Sepideh Saidi. «Formally Proving the Correctness of Functional Programs A Comparison of Different Methods in the Proof Assistant Coq». Tesis de maestría. Technische Universiteit Eindhoven, ago. de 2010 (vid. pág. 99).

[Sco93] Dana S. Scott. «A type-theoretical alternative to ISWIM, CUCH, OWHY». En: *Theoretical Computer Science* 121.1–2 (1993), págs. 411-440. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/0304-3975(93)90095-B. URL: http://www.sciencedirect.com/science/article/pii/030439759390095B (vid. pág. 7).

[SU06] Morten Heine Sørensen y Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. New York, NY, USA: Elsevier Science Inc., 2006. ISBN: 0444520775 (vid. pág. 46).

[Tea14] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.4pl5. The Coq Development Team, 2014. URL: http://coq.inria.fr (vid. pág. 8).

[Veg84] S.R. Vegdahl. «A Survey of Proposed Architectures for the Execution of Functional Languages». En: *Computers, IEEE Transactions on* C-33.12 (dic. de 1984), págs. 1050-1071. ISSN: 0018-9340. DOI: 10.1109/TC.1984.1676387 (vid. pág. 2).

[Wer94] Benjamin Werner. «Une Théorie des Constructions Inductives». Theses. Université Paris-Diderot - Paris VII, mayo de 1994. URL: https://tel.archives-ouvertes.fr/tel-00196524 (vid. págs. 8, 46).

[You67] Daniel H. Younger. «Recognition and parsing of context-free languages in time n^3». En: *Information and Control* 10.2 (1967), págs. 189-208. ISSN: 0019-9958. DOI: http://dx.doi.org/10.1016/S0019-9958(67)80007-X. URL: http://www.sciencedirect.com/science/article/pii/S001999586780007X (vid. pág. 25).

[Zúñ15a] Angel Zúñiga. *Código en Coq de un compilador correcto de Mini-ML a la CAM descrito por Boutin*. 2015. URL: http://www.mcc.unam.mx/~zuniga.a/CCMiniMLtoCam.v (vid. pág. 198).

[Zúñ15b] Angel Zúñiga. *Un compilador correcto verificado de Mini-ML a la máquina SECD en Coq. Código fuente*. 2015. URL: http://www.mcc.unam.mx/~zuniga.a/CCMiniMLtoSecdDB.v (vid. págs. 17, 136, 142, 153, 154).