UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

"Implementación de un compilador de ECMAScript para la plataforma de desarrollo .NET"

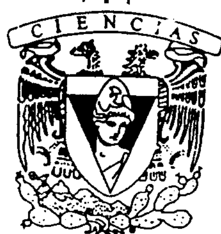
T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

CÉSAR OCTAVIO LÓPEZ NATARÉN

DIRECTORA DE TESIS:
DRA. ELISA VISO GUROVICH



FACULTAD DE CIENCIAS
UNAM

2006

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

VNIVERSIDAD NACIONAL
AVENOMA DE
MEXICO

Autorizo a la Dirección General de Bibliotecas de la UNAM a difundir en formato electrónico e impreso el contenido de mi trabajo recepcional.
NOMBRE: César Octavio López Natarén
FECHA: 11 de enero 2006
FIRMA:

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito: "Implementación de un compilador de ECMAScript para la plataforma de desarrollo .NET"

realizado por López Natarén César Octavio

con número de cuenta 09957666-8 , quien cubrió los créditos de la carrera de: Lic. en Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis
Propietario Dra. Elisa Viso Gurovich

Propietario M. en C. José de Jesús Galaviz Casas

Propietario Lic. en C. C. Manuel Alberto Sugawara Muro

Suplente Lic. en C. C. Karla Ramírez Pulido

Suplente Lic. en C. C. Francisco Lorenzo Solsona Cruz

Consejo Departamental de Matemáticas

Dr. Francisco Hernández Quiroz

Implementación de un compilador de ECMAScript para la plataforma de desarrollo .NET

César Octavio López Natarén

Enero, 2006

Agradecimientos

A mi familia—Papá, Villita y Carlitos—por su apoyo constante e incondicional a lo largo de mi vida, a mi directora de tesis Dra. Elisa Viso Gurovich por todo su apoyo, consejos y dedicación, a Miguel de Icaza por su entusiasmo, motivación y financiar este proyecto, a mis sinodales por su excelente trabajo y finalmente a mis amigos.

Índice general

1. Introducción **4**
1.1. El Proyecto Mono 4
1.2. El compilador para ECMAScript 5

2. Sistemas de tipo **7**
2.1. Introducción 7
2.2. CLI (*Common Language Infrastructure*) 9
2.2.1. Sistema de tipos CTS (*Common Type System*) 10
2.2.2. Metadatos 16
2.2.3. Sistema de Ejecución Virtual 19

3. El lenguaje de programación ECMAScript **32**
3.1. Antecedentes 32
3.2. Características 32
3.2.1. Modelo de ejecución. 33
3.2.2. Programación orientada a objetos basada en prototipos. . 35
3.2.3. Ligado lo más tarde posible 36
3.2.4. Funciones 36
3.2.5. Objetos 39

4. ECMAScript en el CLI **42**
4.1. Implementación del modelo de ejecución 42
4.2. Implementación del ligado lo más tarde posible 44
4.3. Implementación de funciones 45
4.3.1. Declaración de funciones 46
4.3.2. Funciones como tipo de datos 47

5. Algoritmos utilizados en el compilador **49**
5.1. Análisis amortizado 49
5.1.1. Tablas dinámicas 50
5.2. Algoritmos para análisis sintáctico 51
5.2.1. Representaciones intermedias 55
5.3. Algoritmos para análisis semántico 56
5.3.1. Identificación de símbolos 56

5.3.2. Análisis semántico del ligado lo más tarde posible 61
5.4. Algoritmos para generación de código 64
5.4.1. Generación de código para declaración de funciones 65
5.4.2. Generación de código para expresión de funciones 68
5.4.3. Generación de código para expresiones booleanas 68
5.4.4. Generación de código del ligado lo más tarde posible . . . 76
5.4.5. Generación de código de declaración de variables y funciones repetidas 85

6. Conclusiones **89**

Capítulo 1

Introducción

1.1. El Proyecto Mono

En el verano del año 2001, a iniciativa de Ximian, Inc. se inició el proyecto Mono. En un principio el objetivo principal era crear una implementación libre del estándar internacional ECMA-335 [3] conocido como CLI (*Common Language Infrastructure*), el cual define un formato de código y un sistema virtual de ejecución para él. Además de esto, Mono también incluiría un compilador para el lenguaje de programación C#, basado en el estándar ECMA-334 [2]; por último, Mono brindaría una biblioteca de clases, también definida como parte de los estándares de ECMA, que incluye herramientas para la manipulación de XML, creación de hilos de ejecución, manejo de entrada y salida, implementaciones de algunas estructuras de datos, herramientas para seguridad, entre otras cosas. Pero debido al crecimiento del proyecto y al interés por parte de la comunidad del software libre en el desarrollo de componentes basados en C# y el CLI, Mono ahora no está limitado a la implementación de los componentes antes descritos, sino que ya se incluye componentes que brindan interoperabilidad con CORBA y una biblioteca para la creación de aplicaciones gráficas (GTK#), entre otras más; éstas no se encuentran en los estándares de ECMA o en la plataforma de desarrollo .NET.

La importancia del proyecto Mono reside en que brinda un conjunto de herramientas para la construcción de software que promueve una mayor productividad de los desarrolladores de software enfocados a los sistemas libres (GNU/Linux, FreeBSD). Cabe mencionar que los sistemas libres son un mecanismo muy viable para el inicio de una independencia económica y tecnológica de los países en desarrollo. El ahorro monetario que implica la adopción de este tipo de sistemas, permite que dichos recursos se puedan invertir en otro tipo de áreas, tales como la educación e investigación en distintos campos de la ciencia, en particular de las mismas Ciencias de la Computación. Por otro lado, la posibilidad de crecimiento con respecto al conocimiento de técnicas y tecnologías de punta es muy importante.

Además, debido al tipo de tecnologías que se requiere implementar para su funcionamiento óptimo, Mono brinda la oportunidad de tener acceso al código fuente de dichas implementaciones y permite su uso para la exploración de nuevas técnicas y conceptos dentro de los círculos académicos y empresariales, y con ello cataliza la innovación.

1.2. El compilador para ECMAScript

El sistema de tipos del CLI fue diseñado de tal manera que se maximizara la interoperabilidad entre distintos lenguajes de programación, esto con el propósito de reutilizar componentes. Es decir, que componentes escritos en el lenguaje de programación X pudieran ser usados o extendidos por otro componente escrito en el lenguaje de programación Y. Y por "ser usado" queremos decir, poder crear clases que hereden de alguna clase contenida en el otro componente, crear instancias de alguna clase en nuestro componente de manera transparente.

Para ello es necesario:

- La existencia de un conjunto de reglas básicas que permitan y promuevan la interoperabilidad entre los distintos lenguajes.
- Que existan compiladores que traduzcan programas escritos en algún lenguaje de programación a programas expresados en CIL (*Common Intermediate Language*), que pueden ser ejecutados por una implementación del CLI y que además obedecen la semántica definida en dicho estándar.

Es aquí donde este trabajo entra en escena. Implementaré un compilador del lenguaje de programación ECMAScript con miras a tener una implementación de otro lenguaje conocido como JScript .NET.

ECMAScript es un lenguaje de programación cuya especificación se encuentra en el estándard ECMA-262 [1] (actualmente en su tercera edición). ECMAScript es un lenguaje de programación orientado a objetos con dos características importantes:

- La orientación a objetos es provista mediante el concepto de prototipos.
- Nos permite manipular objetos dentro de un ambiente que le brinde alojamiento; sin embargo, dentro del estándar no se especifica a ECMAScript de tal manera que sea computacionalmente autosuficiente, sino que se espera que el ambiente en el cual estará alojado brinde todos los servicios necesarios para entrada y salida, además de otros servicios que se quieran proveer.

Este tipo de lenguajes de programación son conocidos como lenguajes de "scripting". Su característica principal es el poder manipular, adecuar y utilizar de manera automática los servicios de un sistema anfitrión. El escenario más fácil de imaginar es aquel en el que un sistema ya brinda cierta funcionalidad a través de alguna interfaz de usuario y ECMAScript es utilizado para brindar

esos servicios pero a otros programas. El sistema anfitrión provee un ambiente con objetos y servicios que vienen a completar las capacidades de ECMAScript.

Los ejemplos más conocidos son JavaScript, que fue la implementación de ECMAScript de Netscape (actualmente este proyecto continúa con apoyo del proyecto Mozilla; que tiene dos implementaciones, una escrita en C, llamada SpiderMonkey y otra escrita en Java, que es conocida como Rhino), así como JScript que es la implementación de Microsoft Inc., ambas con sus muy particulares extensiones.

En la plataforma de desarrollo .NET, la nueva versión del lenguaje JScript se llama JScript .NET, la cual tiene un conjunto de extensiones a ECMAScript3 (algunas de ellas se encuentran en la especificación aún no completa de ECMAScript4) que le permiten tener acceso a los servicios del CLI.

Debido a su naturaleza dinámica y con una sintaxis bastante relajada, JScript .NET juega un papel importante en el desarrollo de aplicaciones enfocadas al web y para hacer extensiones a aplicaciones ya existentes; esto mediante un lenguaje ya estandarizado y con algún tiempo ya de ser conocido y usado.

Mono, siendo la implementación libre del CLI y otros componentes de la plataforma de desarrollo .NET carece de un compilador de JScript .NET. Ésta es una pieza faltante de importancia considerable, ya que Mono no cuenta con un compilador para un lenguaje de programación con las características de JScript .NET lo que puede traer consigo algunos beneficios. Por ejemplo, algunos de los desarrolladores de Mono han mencionado la idea de que se convierta en el lenguaje a ser utilizado dentro del depurador de programas de Mono. También la migración de aplicaciones escritas para .NET en Windows a los sistemas abiertos en conjunción con Mono, es un punto importante. Es por ello que he decidido tomar esta tarea, que además ofrece una oportunidad para conocer e investigar las características del CLI, así como las distintas maneras en que se pueden mapear las características de los lenguajes dinámicos a una plataforma de desarrollo principalmente diseñada para lenguajes con un sistema de tipos bastante estricto, así como la búsqueda de posibles optimizaciones que nos brinden un mejor desempeño al usar este tipo de lenguajes. Y por último para poder tener a mano esta herramienta dentro del conjunto que nos permitirá construir un mejor sistema libre que sea útil para todo aquel interesado.

Capítulo 2

Sistemas de tipo

2.1. Introducción

La ingeniería de software moderna reconoce un amplio rango de métodos formales que ayudan a asegurar que un sistema se comporte correctamente con respecto a una especificación implícita o explícita. Por un lado existen poderosas técnicas como la lógica de Hoare, lógicas modales y la semántica denotacional. Con ellas se pueden expresar propiedades de correctez muy generales, pero son muy difíciles de usar y demandan que el programador domine un gran número de técnicas sofisticadas. Por otro lado existen técnicas de poder más modesto, de tal manera que se pueden construir programas que realizan la verificación de manera automática; algunos ejemplos de este tipo de métodos formales ligeros[1] son los inspectores de modelo[2] que son herramientas que buscan errores en sistemas con estados finitos como chips y protocolos. Otro ejemplo y que está creciendo en popularidad son los monitores de ejecución[3] que son una colección de técnicas que le permiten a un sistema detectar, dinámicamente, cuando alguno de sus componentes no se está comportando de acuerdo a la especificación. Sin embargo, el más popular de estos métodos es el de los sistemas de tipo.

Existen muchas definiciones de sistema de tipo, pero la que Pierce [10] da es muy adecuada para nuestros propósitos:

> Un sistema de tipo es un método sintáctico tratable para probar la ausencia de ciertos comportamientos en un programa a través de la clasificación de las frases de acuerdo al tipo de valores que calculan.

Esa definición enfatiza las siguientes características de los sistemas de tipo:

- Se les identifica como una herramienta para analizar programas.

[1] El término en inglés es *lightweight formal methods*.
[2] El término en inglés es *model checker*.
[3] El término en inglés es *runtime monitoring*.

- Realizan una clasificación de los términos de acuerdo a las propiedades de los valores que van a calcular cuando se ejecute. Por esta razón muchas veces se dice que los sistemas de tipo calculan una aproximación *estática* del comportamiento de los términos en ejecución.
- Siendo estáticos, los sistemas de tipo son también conservadores; pueden probar la ausencia de algún comportamiento erróneo en el programa, pero no pueden probar su presencia; por lo tanto en ocasiones tienen que rechazar programas que se comportan correctamente en ejecución. La tensión entre conservadurismo y expresividad es un aspecto fundamental en el diseño de un sistema de tipos.

Dentro de los lenguajes de programación se usan los sistemas de tipo por varias razones:

- **Detección de errores.** El beneficio más obvio de realizar el análisis de tipo de forma estática es que nos permite detectar errores de forma temprana, es decir en tiempo de compilación, antes de que el programa se ejecute. En la práctica este tipo de análisis logra evidenciar un rango amplio de errores que va desde simples descuidos de conversión de tipos hasta detectar errores conceptuales más profundos que se manifiestan como inconsistencias al nivel de tipos.
- **Documentación.** Los tipos son útiles cuando se está leyendo un programa. Las declaraciones de tipo en los encabezados de las funciones y las interfaces de módulo constituyen una forma de documentación. Una de las ventajas sobre la documentación escrita como comentarios en el programa es que esta información siempre está en sincronía con la semántica del programa ya que se verifica cada vez que se compila.
- **Seguridad del lenguaje.** Una noción intuitiva de lo que es un lenguaje seguro viene dada por la característica de que todos los lenguajes de programación de alto nivel brindan abstracciones de los servicios de la computadora (por ejemplo el acceso a memoria); la seguridad radica en el poder garantizar la integridad de estas abstracciones. Se dice generalmente que en un lenguaje seguro esas abstracciones se pueden usar de manera *abstracta* sin necesidad de preocuparse de detalles de bajo nivel sobre cómo está implementado, a diferencia de un lenguaje inseguro. No se debe confundir la seguridad del lenguaje con seguridad de tipo estática. La seguridad de lenguaje se puede obtener mediante una verificación estática, pero también mediante verificaciones en tiempo de ejecución que atrapan las operaciones que no tengan sentido en el momento que se intentan realizar y detiene la ejecución del programa o lanza una excepción. La seguridad en tiempo de ejecución generalmente no se puede garantizar mediante tipificación estática; la mayoría de los lenguajes seguros agregan verificaciones que se realizan dinámicamente cuando el programa se ejecuta.

- **Eficiencia.** Los primeros sistemas de tipo se crearon con el fin de poder generar código más eficiente para la evaluación de operaciones aritméticas, ya que poder diferenciar entre una operación entre enteros y números de punto flotante era vital.

El diseño de un lenguaje de programación y el de su sistema de tipos deben ir siempre muy ligados; el intentar unir un sistema de tipo que se diseñó de manera independiente a las características sintácticas y semánticas del lenguaje se puede volver una tarea muy difícil. Una de las razones de esto es que lenguajes de programación sin sistema de tipos - lenguajes seguros - tienden a ofrecer estructuras idiomáticas que hacen la tarea de chequeo de tipos muy difícil o irrealizable. Más aún, en los lenguajes tipificados el sistema de tipos por sí mismo es el fundamento a partir del cual se diseña y organiza cualquier otro aspecto del lenguaje. Otro factor es que la sintaxis concreta de un lenguaje tipificado tiende a ser más complicada que aquella de un lenguaje no tipificado, ya que las anotaciones de tipo deben tomarse en cuenta. Es más fácil diseñar una sintaxis limpia y entendible cuando se toman en consideración los aspectos involucrados.

La afirmación de que los tipos deben ser parte integral del lenguaje de programación está separada de la pregunta de dónde el programador debe físicamente escribir las anotaciones de tipo y dónde pueden ser inferidas por el compilador. Un lenguaje estáticamente tipificado bien diseñado nunca requerirá grandes cantidades de información sobre los tipos, que tendrá que ser mantenida explícitamente por el programador. Sin embargo, no hay acuerdo en la definición exacta de "grandes cantidades" en este caso.

2.2. CLI (*Common Language Infrastructure*)

El CLI es un sistema de ejecución virtual de programas y está definido en el estándar internacional ECMA-335, en él se especifican los siguientes componentes:

- Un sistema de tipos, conocido como CTS (*Common Type System*).
- Un subconjunto del CTS que promueve y facilita la integración entre lenguajes de programación, conocido como CLS (*Common Language Specification*).
- Un conjunto de metadatos que nos permiten describir y hacer referencias a los tipos definidos por el CTS.
- Un sistema virtual de ejecución, el cual implementa el modelo definido por el CTS, conocido como VES (*Virtual Execution System*).

Todos estos componentes forman una plataforma unificada para el diseño, desarrollo, instalación y ejecución de aplicaciones y componentes distribuidos.

El CLI además funciona como capa intermedia entre las aplicaciones diseñadas e implementadas para este sistema virtual y el sistema operativo. Estas

aplicaciones necesariamente debieron haber sido creadas mediante la transformación del programa en el lenguaje de programación en el cual fueron escritas a una representación intermedia muy particular, el CIL, que es el conjunto de instrucciones de la máquina abstracta con pila de ejecución que define; ése es su modelo de cómputo.

2.2.1. Sistema de tipos CTS (*Common Type System*)

El CLI trabaja con dos tipos de entidades, que son: los objetos y los valores. Los valores son simplemente patrones de bits, donde cada valor tiene un tipo que describe su forma de almacenamiento, el significado de los bits en su representación, así como las operaciones que se pueden realizar en él. El objeto por otro lado es autotipificado, es decir, su tipo está explícitamente almacenado en su representación, posee una identidad que lo distingue de los demás objetos, posee ranuras para almacenar otras entidades (valores u objetos).

Tipos

Los tipos describen valores. Todos los lugares donde se almacenan, envían valores con un tipo específico definido o se opera sobre ellos tienen un tipo, por ejemplo: variables, parámetros, posiciones de evaluación en la pila de ejecución y los resultados de invocaciones a métodos. Ellos definen los valores permitidos y las operaciones brindadas por los valores de un tipo.

Valores y referencias

En el CLI existen dos clases de tipos: valores (*value types*) y referencias (*reference types*).

- **Valor**. Describe valores que son representados como patrones de bits.
- **Referencia**. Describe valores que son representados como la posición de una sucesión de bits. Hay cuatro clases de referencias:
  - **Objeto**. Es la referencia de un valor autodescriptible.
  - **Interfaz**. Siempre es la descripción parcial de un valor.
  - **Apuntador**. Es la descripción - en el momento de compilación - de un valor cuya representación es la dirección de una posición.
  - **Primitiva**. Son los tipos básicos como los números enteros o números de punto flotante que vienen interconstruidos.

Clases, interfaces y objetos.

Todo valor tiene un tipo “exacto” que lo describe completamente. Un tipo describe completamente un valor si define de manera no ambigua su representación y las operaciones definidas en él.

Para un valor, definir la representación requiere describir la secuencia de bits que forman la representación del valor. Para una referencia, definir la representación requiere describir la posición y la secuencia de bits que forman la representación del valor.

Un método describe una operación que podría realizarse a valores de un tipo exacto. Definir el conjunto de operaciones permitidas en valores de un tipo exacto requiere especificar métodos con nombre asignado por cada operación.

Algunos tipos son sólo descripciones parciales, por ejemplo, la interfaz. Una interfaz describe un subconjunto de las operaciones pero no describe algo sobre la implementación (incluida la representación interna), por lo tanto no puede ser el tipo exacto de algún valor. Por esta razón, mientras un valor solamente tiene un tipo exacto también podría ser valor de algunos otros tipos. Además, como el tipo exacto describe completamente el valor, también especifica completamente todos los demás tipos que el valor del tipo exacto puede tener.

Como hemos dicho, todo valor tiene un tipo exacto que lo describe, sin embargo no siempre es posible determinar el tipo exacto de un valor simplemente con inspeccionar la representación del valor. En particular, no es posible determinar el tipo exacto de un valor de la clase de los valores. Para algunos valores, llamados objetos, siempre es posible determinar su tipo exacto. A los tipos exactos de los objetos se les llama tipos objeto. Los objetos son valores de tipo referencia, pero no todos los tipos referencia describen objetos.

Localidades

Los valores son almacenados en localidades. Una localidad sólo puede contener un valor en un momento dado. Todas las localidades tienen asignado un tipo. El tipo de la localidad impone los requerimientos que deben cumplir los valores que se quisieran almacenar en ella.

Un valor podrá ser almacenado en una localidad solamente si alguno de los tipos del valor es compatible bajo asignación con el tipo de la localidad. Describiremos la definición de compatibilidad bajo asignación más adelante.

Contratos

Los contratos son un conjunto de firmas que representan métodos que se comprometen a respetarlas. Las firmas son la parte del contrato que puede ser validada e impuesta.

Los contratos no son tipos, en realidad ellos especifican los requerimientos en la implementación de los tipos. Existen varios tipos de contratos:

- **Contrato de clase**. Este contrato se especifica mediante la definición de una clase. Por lo tanto, la definición de una clase define el contrato de clase y el tipo clase. Un contrato de clase especifica la representación de los valores del tipo clase. Además, un contrato de clase especifica los otros contratos para los cuales el tipo clase da soporte, por ejemplo cuáles interfaces, métodos, propiedades y eventos deberían ser implementadas.

Un contrato de clase puede ser cumplido por otras clases. Un tipo clase que brinda el contrato de clase de otro tipo de clase se dice que hereda de ese tipo de clase.

- **Contrato de interfaz**. Este contrato se especifica mediante la definición de una interfaz. Por lo tanto, la definición de una interfaz define el contrato de interfaz y el tipo interfaz. Muchos tipos pueden cumplir un contrato de interfaz. De igual manera que el contrato de clase, el contrato de interfaz especifica que otros contratos brinda.

- **Contrato de método**. Este contrato se especifica con una definición de método. Un contrato de método es una operación que especifica el contrato entre la implementación del método y los invocadores del método. Un contrato de método siempre forma parte de un contrato de tipo (clase, valor o interfaz), y describe cómo una operación interacciona con el invocador. El contrato de método especifica el contrato que cada parámetro deberá cumplir y los contratos que el valor de regreso debe cumplir, en caso de haberlo.

- **Contrato de propiedad** (*property*). Este contrato se especifica con la definición de una propiedad. Un contrato de propiedad especifica los contratos de método para el subconjunto de operaciones `get` y `set` sobre un valor, que podría ser implementado por cualquier tipo que cumpla el contrato de propiedad. Un tipo puede dar soporte para muchos contratos de propiedad, pero cualquier contrato de propiedad puede ser cumplido por exactamente un solo tipo. Por lo tanto, las definiciones de propiedad son parte de la definición del tipo que brinda soporte a esa propiedad.

- **Contrato de evento**. Este contrato se especifica con una definición de evento. Existen un conjunto extendible de operaciones para manejar un evento, las cuales incluyen tres métodos estándar (registrar interés en un evento, revocar interés en un evento y lanzar el evento). Un contrato de evento especifica los contratos de método para este conjunto de operaciones que podrían ser implementadas por cualquier tipo que cumpla el contrato de evento. Un tipo puede cumplir muchos contratos de eventos, pero cualquier contrato de evento puede ser brindado por exactamente un tipo. Por lo tanto, las definiciones de evento son parte de la definición de tipo del tipo que brinda el evento.

Firmas

Las firmas[4] son la parte del contrato que puede ser validada y automáticamente impuesta. Las firmas se forman agregando restricciones en tipos y otras firmas. Una restricción es una limitación en el uso u operaciones permitidas en valores o localidades.

[4] En este contexto, la noción usual de firma que se refiere al tipo de regreso de un método, su nombre, el nombre y tipo de sus parámetros es extendida y denota las restricciones que se pueden imponer sobre tipos y otras firmas.

Todas las localidades tienen firmas, así como todos los valores. La compatibilidad bajo asignación requiere que la firma de un valor sea compatible con la firma de una localidad. Hay cuatro tipos fundamentales de firmas: firma de tipo, firma de localidad, firma de parámetro y firma de método.

- **Firma de tipo**. La firma de tipo define las restricciones sobre un valor y su uso. Un tipo, por sí mismo, es una firma válida. La firma de tipo de un valor no puede determinarse examinando el valor o conociendo el tipo de clase del valor. La firma de tipo de un valor se obtiene a partir de la firma de localidad de la localidad de la cual el valor es obtenido. Normalmente la firma de tipo de un valor es el tipo en la firma de localidad de la cual el valor es cargado.

- **Firma de localidad**. Todas las localidades tienen un tipo. Esto significa que todas las localidades tienen una firma de localidad, la cual define las restricciones sobre ella, su uso y el uso de los valores almacenados en ella. Cualquier firma de tipo válida es una firma de localidad válida también. Por lo tanto, una firma de localidad contiene un tipo y podría adicionalmente contener la restricción constante. La firma de localidad podría también contener restricciones de localidad que dan mayor control en el uso de la localidad. Las restricciones de localidad son:

  - **init-only**. Esta restricción indica que una vez que la localidad ha sido iniciada, su contenido no vuelve a cambiar.
  - **literal**. Esta restricción indica que el valor de esa localidad es en realidad un valor fijo de algún tipo interconstruido.

- **Firma local**. Una firma local especifica el contrato sobre una variable local alojada durante la ejecución de un método. Una firma local contiene una firma de localidad completa y además podría especificar una restricción adicional:

  - **byref.** Esta restricción afirma que el contenido de la correspondiente localidad es un apuntador gestionado. Un apuntador gestionado podría apuntar a una variable local, un parámetro, a un campo de un tipo compuesto o al elemento de un arreglo. Sin embargo, cuando una llamada cruza un límite remoto[5] una implementación podría usar un mecanismo de copia de entrada/copia de salida en lugar del apuntador gestionado. Por lo tanto un programa no podría confiar en el comportamiento de alias de los verdaderos apuntadores.

- **Firma de parámetro**. La firma de parámetro define restricciones sobre cómo un valor individual es introducido como parte de una invocación a un método. Las firmas de parámetro se declaran con las definiciones de métodos.

[5] Un límite remoto existe si no es posible compartir la identidad de un objeto directamente a través del límite. Por ejemplo, si dos objetos existen físicamente en máquinas separadas que no comparten un espacio de direcciones común, entonces existe un límite remoto.

- **Firma de método**. La firma de método está compuesta por:
  - Convención de invocación.
  - Una lista de cero o más firmas de parámetro, una por cada parámetro del método.
  - Una firma de tipo para el resultado, si se produce alguno.

  Las firmas de método son declaradas por las definiciones de método. Solamente una restricción más puede ser agregada además de aquellas de firmas de parámetro:

  - **varargs**. Esta restricción podría ser incluida para indicar que todos los argumentos una vez pasado este punto son opcionales. Cuando aparece, la convención de invocación debe ser aquella que permite una lista de argumentos de longitud variable.

  Las firmas de método se usan de dos maneras distintas: como parte de una definición de método y como una descripción de un sitio de invocación cuando se llama a través de un apuntador a función.

Compatibilidad bajo asignación

Las restricciones en las firmas de tipo y de localidad afectan la compatibilidad bajo asignación de un valor y una localidad. La compatibilidad bajo asignación de un valor (descrito por una firma de tipo) con una localidad (descrita por una firma de localidad) está definida de la siguiente manera:

- Alguno de los tipos brindados por el tipo exacto del valor es el mismo que el tipo en la firma de localidad.

Esto permite, por ejemplo, que un ejemplar de una clase que hereda de una clase base (y por lo tanto da soporte al contrato de tipo de la clase base) sea almacenado en una localidad cuyo tipo es aquel de la clase base.

Verificación y código seguro en el contexto del sistema de tipos

Los tipos especifican contratos, por lo que es importante saber si una implementación cumple el contrato. Una implementación que cumple con la parte que se puede imponer y verificar de un contrato se dice que es *segura bajo tipo*. Las características que definen una implementación *segura bajo tipo* se expresan a continuación:

- Solamente almacenan valores descritos por una firma de tipo en localidades que son compatibles bajo asignación con la firma de localidad de la misma.
- Nunca aplican una operación a un valor que no esté definida por el tipo exacto del valor.
- Acceden únicamente a localidades que son visibles y accesibles a ella.
- El tipo exacto de un valor no puede cambiar.

Definición de tipos

Las definiciones de tipos construyen un tipo nuevo a partir de tipos existentes. Los tipos implícitos (tipos primitivos, arreglos y apuntadores, incluidos los apuntadores a funciones) se definen cuando son usados. La mención de un tipo implícito en una firma es en sí misma una definición completa del tipo. Los tipos implícitos le permiten al sistema virtual de ejecución fabricar instancias con un conjunto estándar de miembros, interfaces, etc. Los tipos implícitos no necesitan nombres provistos por el usuario.

Cualquier otro tipo deberá estar definido de manera explícita usando una definición explícita de tipo. Los constructores de tipos explícitos son:

- Definición de interfaz.
- Definición de clase - usados para definir:
  - Objetos.
  - Valores y sus tipos encapsulados (boxed) asociados.
- Arreglos. Un arreglo podría ser definido especificando el tipo de elementos que almacenará, su rango (número de dimensiones) y cotas superior e inferior de cada dimensión. Por lo tanto, no se necesita de una definición separada para arreglos.
- Apuntadores no gestionados. Un apuntador no gestionado se define especificando la firma de localidad para la localidad a la cual el apuntador hace referencia. Cualquier firma de un apuntador incluye esa firma de localidad. Por lo tanto, no se necesita una definición de apuntador separada. Aunque los tipos apuntador son referencias, los valores de un apuntador no son objetos, por lo tanto no es posible determinar dado un valor de un apuntador su tipo exacto. Sin embargo el CTS brinda dos operaciones seguras sobre apuntadores: una carga el valor de la localidad referida por el apuntador y la otra almacena un valor que es compatible bajo asignación en la localidad. Además también brinda tres operaciones más: suma y resta de enteros con apuntadores, y resta entre apuntadores.
- Delegados (*delegates*). Los delegados son en la programación orientada a objetos el equivalente a los apuntadores a funciones en la programación estructurada. Sin embargo, como los delegados derivan de una clase no son inseguros como los apuntadores a funciones. Los delegados son creados definiendo una clase que deriva de un tipo base llamado System.Delegate. Cada delegado puede proveer un método llamado Invoke con parámetros apropiados, y cada instancia delega las llamadas a su método Invoke a un método estático o de instancia en un objeto particular. El objeto y el método a los cuales delega el llamado se escogen cuando la instancia del delegado se crea.

2.2.2. Metadatos

Cualquier tipo nuevo - ya sea referencia o valor - se introduce al CTS mediante una declaración de tipo expresada en metadatos. Además, los metadatos proveen un mecanismo estructurado para representar toda la información que el CLI usa para localizar y cargar clases, crear instancias en memoria, resolver invocaciones a métodos, traducir CIL a código nativo, forzar la seguridad y establecer barreras de contexto al momento de ejecutar los programas. Todo módulo CLI PE/COFF lleva consigo una porción compacta de metadatos que es incrustada dentro del módulo por la herramienta o compilador compatible con el CLI.

Cada lenguaje de programación que permita el uso de los servicios del CLI brindará la sintaxis apropiada para la declaración de tipos y la anotación de ellos con atributos que expresen qué servicios requieren del CLI.

Componentes y montajes (*assemblies*)

Cada componente CLI porta los metadatos de declaraciones, implementaciones y referencias de ese componente específico. Por lo tanto, esos metadatos específicos al componente son llamados metadatos del componente, y el componente resultante se dice que es autodescriptible.

Colecciones de componentes CLI y otros archivos se empaquetan para su distribución en montajes. Un montaje es la unidad lógica de funcionalidad que sirve como la unidad primaria de reuso en el CLI. Los montajes establecen fronteras de nombre de los tipos.

Tipos declarados e implementados en componentes individuales se exportan para ser usados vía un montaje en el cual el componente participa. Toda referencia a un tipo está sujeta a la identidad del montaje al cual pertenece dicho tipo. El CLI provee servicios para localizar un montaje referido y para solicitar la resolución de un tipo referido.

Acceso a metadatos

Los metadatos pueden ser escritos o leidos de un módulo CLI a través de acceso directo al formato del archivo - especificado más adelante - o mediante los servicios de introspección (Reflection).

Cuando una clase se carga en tiempo de ejecución, el cargador (loader) del CLI importa los metadatos a sus propias estructuras de datos en memoria—éstas pueden ser vistas vía los servicios de introspección: estos servicios pueden considerarse similares a un compilador; ellos automáticamente recorren la jerarquía de herencia para obtener información sobre métodos y campos heredados—pueden proporcionarse distintos parámetros para especificar de manera más especializada los discriminantes a utilizar.

Átomos de metadatos

Un átomo de metadatos es un mecanismo de codificación dependiente de la implementación. Los átomos de metadatos se incrustan en el código CIL y nativo para codificar la invocación de métodos y el acceso a campos; el átomo es usado en varios servicios de infraestructura para obtener información a partir de los metadatos sobre la referencia y el tipo en el cual se encuentra declarado para resolver la referencia.

Un átomo de metadatos es un identificador tipificado de un objeto metadato (declaración de tipo, de miembro, etc.). Dado un átomo, su tipo puede ser determinado y es posible obtener los atributos específicos de ese objeto metadato. Sin embargo, un símbolo de metadatos no es un identificador persistente.

Código no gestionado (*unmanaged code*)

Es posible trasladar datos del universo del código gestionado a aquel que no lo está. Esto siempre involucra una transición que tiene un costo en el desempeño, pero la mayoría de las veces la información puede trasladarse sin necesidad de copiado. Cuando la información tiene que volver a su formato original el Sistema de Ejecución Virtual (VES por sus siglas en inglés) brinda una especificación razonable del comportamiento genérico; sin embargo es posible usar los metadatos para requerir explícitamente otras formas de realizar la conversión (marshalling).

Metadatos de la implementación de métodos

Por cada método para el cual se brinda una implementación en el módulo CLI actual, el compilador generará información que será usada por el JITter (compilador Just In Time), el cargador del CLI y otros servicios de infraestructura. Esta información se refiere a:

- Si el método es gestionado o no.
- Si la implementación es en código nativo o en CIL.
- La posición del cuerpo del método en el módulo actual.

Montajes (*assemblies*)

Un montaje es una colección de recursos que están construidos para trabajar juntos y que brindan cierta funcionalidad. Es la unidad de acceso a recursos en el CLI.

Visto desde fuera, un montaje es una colección de recursos que se exportan, incluidos los tipos. Se exporta el nombre del recurso. Internamente, un montaje es una colección de recursos públicos y privados. El montaje es quien determina qué recursos son expuestos al exterior y cuáles son accesibles solamente dentro del montaje actual. El montaje es quien controla cómo es mapeada una referencia a un recurso, público o privado, a la representación binaria que lo implementa. Para los tipos, por ejemplo, el montaje podría brindar información

de configuración. Un módulo CLI se puede pensar como un paquete de declaraciones de tipo y sus implementaciones, donde las decisiones de empaquetado podrían cambiar pero que no afecta a los clientes del montaje.

La identidad de un tipo es su alcance de montaje y el nombre con el que fue declarado. Un tipo definido idénticamente en dos montajes distintos son considerados dos tipos diferentes.

- **Dependencias.** Un montaje puede depender de otros montajes. Esto ocurre cuando implementaciones en el alcance de un montaje hacen referencia a recursos que pertenecen a otro montaje.

  - Todas las referencias a otros montajes son resueltas bajo el control del alcance del montaje actual. Esto permite controlar la versión particular del recurso que se está referenciando.
  - Siempre es posible conocer el alcance de montaje al cual pertenece una implementación que está siendo ejecutada. Todas las peticiones que lance se resuelven en ese contexto.

Desde una perspectiva de distribución, un montaje podría ser distribuido de manera aislada, con la suposición de que cualquier montaje referido por él estará presente en el ambiente en el que será distribuido. En otro caso podría ser distribuido con los montajes de los que depende.

Un concepto importante dentro de la discusión sobre montajes es el de manifiestos (manifests):

- **Manifiestos.** Todo montaje tiene un manifiesto que declara cuáles archivos forman el montaje, cuáles tipos son exportados y cúales otros montajes se requieren para resolver todas las referencias de tipos dentro del montaje.

Los montajes introducen semántica de aislamiento para las aplicaciones. Una aplicación es simplemente un montaje que tiene un punto de entrada externo que activa la creación de un nuevo dominio de aplicación (application domain). Este punto de entrada es la raíz de un árbol de peticiones de invocación y resoluciones de éstas.

Metadatos extendibles

Los metadatos del CLI son extendibles. Hay tres razones por lo cual esto es importante:

- El CLS (*Common Language Specification*) es una especificación de convenciones para las cuales lenguajes y herramientas acuerdan brindar soporte de una manera uniforme, esto con el fin de lograr una mejor integración entre lenguajes. El CLS restringe partes del modelo del CTS, además introduce abstracciones de más alto nivel las cuales están diseñados en capas que van sobre el CTS. Es importante que los metadatos puedan capturar este tipo de abstracciones las cuales son usadas por las herramientas aún cuando no son reconocidas explícitamente por el CTS.

- Debería ser posible representar abstracciones que son muy particulares al lenguaje de programación en forma de metadatos sin importar si no son abstracciones del CLI o el CLS.
- Debería ser posible codificar tipos y modificadores de tipos que son usados para hacer sobrecarga.

Esta extensibilidad está dada de las siguientes maneras:

- Todo objeto metadato puede poseer atributos especializados, y la interfaz de programación (API) provee las maneras de declarar, enumerar y obtener dichos atributos especializados. Estos atributos podrán ser identificados con un nombre simple, donde el valor de codificación es opaco y conocido para la herramienta, lenguaje o servicio que lo definió. A su vez los atributos especializados podrán ser identificados por una referencia, donde la estructura del atributo es autodescriptible.
- Además de la extensibilidad del CTS, es posible generar modificadores especializados en miembros de la firmas.

2.2.3. Sistema de Ejecución Virtual

El Sistema de Ejecución Virtual (VES, por sus siglas en inglés) brinda todo un ambiente de ejecución de código gestionado. Da soporte directo para un conjunto de tipos primitivos, define una máquina hipotética con un modelo y estado asociado, un conjunto de construcciones para el control de flujo y un modelo para el manejo de excepciones. A grandes rasgos, el propósito del Sistema de Ejecución Virtual es proveer el soporte requerido para ejecutar el conjunto de instrucciones del Lenguaje Común Intermedio (CIL, por sus siglas en inglés).

Soporte de tipos

El CLI brinda soporte directo para los tipos: int8, unsigned int8, int16, unsigned int16, int32, unsigned int32, int64, unsigned int64, float32, float64, native int, native unsigned int; son los tipos que pueden ser manipulados directamente usando el conjunto de instrucciones definidos en el CIL.

El modelo del CLI usa una pila de evaluación. Las instrucciones que copian valores de la memoria a la pila de evaluación son operaciones de carga; las que copian valores de la pila a la memoria son operaciones de almacenamiento. Sin embargo, el CLI solamente da soporte a un subconjunto de ellos en las operaciones que puede realizarse sobre valores que están en la pila de evaluación.

Instrucciones CIL y tipos numéricos

La mayoría de las instrucciones que manejan números toman sus operandos de la pila de evaluación, estas entradas tienen un tipo asociado que es conocido por sistema virtual de ejecución. Como resultado, una sola operación como add

puede tener entradas de cualquier tipo numérico, aunque no todas las operaciones pueden manejar todas las combinaciones de tipos. Las operaciones binarias que no son la suma y la resta requieren que ambos operandos sean del mismo tipo.

Las instrucciones se pueden dividir en las siguientes categorías:

- **Numérica**. Estas instrucciones trabajan con enteros o números de punto flotante, y consideran a los enteros con signo. A esta categoría pertenecen instrucciones de aritmética simple, saltos condicionales y comparación.
- **Entera**. Estas instrucciones trabajan con enteros. Pertenecen a esta categoría operaciones bit a bit, división y residuo entre enteros sin signo.
- **De punto flotante**. Estas instrucciones trabajan con números de punto flotante.
- **Específica**. Estas instrucciones trabajan con enteros y números de punto flotante, pero tienen variantes que trabajan específicamente con diferentes tamaños y enteros sin signo. Pertenecen a esta categoría las operaciones entre enteros con detección de desbordamiento, instrucciones de conversión de datos y operaciones que transmiten datos de la pila de evaluación a otras partes de la memoria.
- **Sin signo**. Hay instrucciones de comparación y de salto que tratan a los enteros como si no tuvieran signo y consideran números de punto flotante de manera especial.
- **Cargado de constantes**. Las instrucciones de cargado de constantes (ldc.*) se usan para cargar constantes de tipo int32, int64, float32 o float64.

Instrucciones CIL y apuntadores.

El uso de referencias está fuertemente restringido en el CIL. Se usan casi exclusivamente con las instrucciones del sistema virtual de objetos, las cuales están diseñadas específicamente para trabajar con objetos. Además, algunas de las instrucciones base del CIL trabajan con referencias a objetos. En particular, las referencias a objetos pueden ser:

- Cargadas a la pila de evaluación para ser pasadas como argumentos a métodos (ldloc, ldarg), y almacenadas desde la pila de evaluación a su localidad (stloc, starg).
- Duplicadas o eliminadas de la pila de evaluación (dup, pop).
- Usadas para probar igualdad con otra referencia del mismo tipo (beq, beq.s, bne, bne.s, ceq).
- Cargados de o almacenadas en memoria no gestionada (ldind.ref, stind.ref).
- Creada como una referencia nula (ldnull).

- Regresada como valor (ret).

Los apuntadores gestionados tienen algunas operaciones básicas adicionales:

- Suma y resta de enteros, en unidades de bytes; regresa un apuntador gestionado (add, add.ovf.u, sub, sub.ovf.u).
- Resta de dos apuntadores gestionados a elementos de un mismo arreglo, regresa el número de bytes entre ellos.
- Comparación sin signo y salto condicional basado en dos apuntadores gestionados (bge.un, bge.un.s, bgt.un, bgt.un.s, entre otras).

Información de módulo

El CLI depende de la siguiente información sobre cada método definido en un archivo PE[6]:

- Las instrucciones que componen el cuerpo de la función, incluyendo todos los manejadores de excepciones.
- La firma de los métodos, la cual especifica el tipo de regreso, el número, orden y la convención en que los parámetros serán pasados y el tipo de los argumentos. También especifica la convención de invocación nativa.
- El arreglo de manejo de excepciones. Este arreglo almacena información que marca los rangos sobre los cuales las excepciones se filtran y capturan.
- El tamaño de la pila de evaluación del método.
- El tamaño del arreglo de variables locales del método.
- Una bandera que indica si las variables locales y la bodega de memoria deben ser iniciadas por el CLI.
- El tipo de cada variable local con la forma de la firma del arreglo de variables locales.

Estado de la máquina

Uno de los propósitos en el diseño del CLI es esconder los detalles del marco de una llamada a función[7]. Esto permite al Sistema de Ejecución Virtual escoger las convenciones más eficientes para realizar el llamado y la composición física de la pila. Para lograr esta abstracción, el marco de la llamada está integrada en el CLI. Las definiciones que vienen a continuación reflejan ese diseño, donde el estado de la máquina consiste principalmente de un estado global y un estado del método.

[6]Del término en inglés *Portable Executable*.
[7]Del término en inglés *method call frame*.

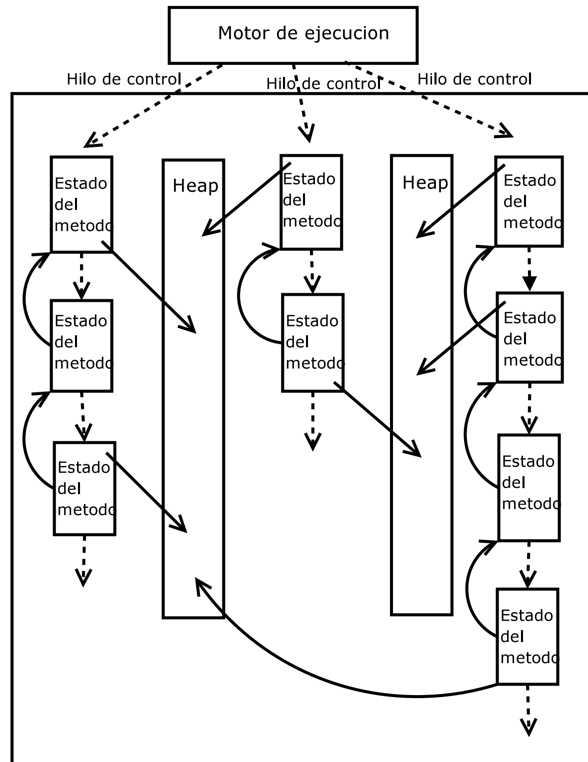


Figura 2.1: Modelo de estado en el CLI.

Estado Global

El CLI se encarga de manejar varios hilos de control concurrentemente, ejecuciones múltiples y un espacio de direcciones de memoria compartido, como se muestra en la figura 2.1.

Registro de activación

El estado del método describe un ambiente en el cual un método se ejecuta. El estado del método en el CLI consiste de:

- Un *apuntador de instrucción*[8]. Apunta a la siguiente instrucción a ser ejecutada en el método actual.
- Una *pila de evaluación*. La pila está vacía cuando el método inicia. Su contenido es estrictamente local al método y se preserva a través de instrucciones de invocación. La pila de ejecución no es direccionable. En todo momento es posible deducir cual tipo - de un conjunto reducido de tipos

[8] *IP*, por sus siglas en inglés *instruction pointer*.

- se encuentra guardado en cualquier posición de la pila en un punto específico del flujo de instrucciones.

- Un *arreglo de variables locales* (que inicia en el índice 0). Una variable local puede guardar datos de cualquier tipo. Sin embargo, deberá ser usada de manera consistente de acuerdo a su tipo. Las variables locales son iniciadas a 0 antes de la entrada al método si la bandera de iniciar se encuentra activada. La dirección de una variable local individual se puede obtener usando la instrucción ldloca.

- Un *arreglo de argumentos*. Son los valores de los argumentos de entrada del método actual (que inician en el índice 0). Se puede acceder a ellos mediante un índice. La dirección de un argumento puede obtenerse usando la instrucción ldarga.

- Un *manejador de la información del método*[9]. Contiene información - de solo lectura - del método. En particular contiene la firma del método, los tipos de sus variables locales e información de los manejadores de excepciones.

- *Memoria dinámica local*. El CLI incluye instrucciones para el alojamiento dinámico de objetos en ella (localloc). La memoria requerida de esta forma es direccionable. Esta memoria se libera cuando el método termina.

- Un *manejador del estado de regreso*. Se usa para restablecer el estado del método cuando termina el método actual. Típicamente, éste sería el estado del método desde el cual se realizó la invocación.

- Un *descriptor de seguridad*. Este descriptor no se puede alcanzar directamente desde el código gestionado pero usa el sistema de seguridad del CLI para registrar escrituras de seguridad (assert, permit-only y deny).

Las cuatro áreas del registro de activación - el arreglo de argumentos, el arreglo de variables locales, memoria local y la pila de evaluación - se especifican como si fueran áreas lógicamente distintas.

La pila de evaluación

Con el registro de activación está asociada una pila de evaluación. La mayoría de las instrucciones del CLI obtienen sus argumentos de ella y colocan su valor de regreso ahí. Los argumentos para otros métodos y sus valores de regreso también son colocados en ella. Cuando se efectúa una llamada a función, los argumentos del método invocado, se convierten en el arreglo de argumentos del método. Esto podría requerir una copia de memoria o simplemente compartir estas áreas.

La pila de evaluación está formada por ranuras que pueden contener cualquier tipo de dato, incluyendo valores. El estado de la pila (su profundidad y los

[9]Del término en inglés *method info handle*.

tipos de cada uno de sus elementos) en cualquier punto dado de un programa será idéntico para todos los posibles caminos de flujo.

Mientras el CLI, en general, brinda soporte para el conjunto completo de tipos descritos, trata de manera especial la pila de evaluación. Aunque algunos compiladores podrían registrar la información de los tipos que se encuentran en la pila, el CLI sólo requiere que el valor sea alguno de los siguientes:

- int64.
- int32.
- int primitivo.
- F, un valor de punto flotante (ya sea float32 o float64 u otra representación soportada por el hardware).
- &, un apuntador gestionado.
- O, una referencia a un objeto.
- *, un apuntador temporal. Son creados exclusivamente por el CLI, nunca por un usuario.
- Un valor definido por el usuario.

Los otros tipos se pueden sintetizar a través de una combinación de técnicas:

- Los tipos enteros más pequeños en otras localidades de memoria se extienden con ceros o con el signo cuando se llevan a la pila de evaluación; y se truncan cuando se almacenan de regreso en su localidad nativa.
- Las instrucciones especiales realizan conversiones numéricas, con detección de derrame o sin ella, entre diferentes tamaños y entre enteros con signo y sin él.
- Las instrucciones especiales tratan valores enteros que se encuentran en la pila de evaluación como si no tuvieran signo.

Variables locales y argumentos

Como parte del registro de activación hay un arreglo que contiene las variables locales y un arreglo que contiene los argumentos. Igual que la pila de ejecución, cada elemento de estos arreglos puede almacenar cualquier tipo de datos o ejemplar de algún valor. Ambos arreglos inician en el índice 0. La dirección de una variable local se puede calcular usando la instrucción ldloca, y la dirección de un argumento con la instrucción ldarga.

Asociado a cada método hay metadatos que especifican:

- Si las variables locales y la memoria local serán iniciadas cuando el método entre en ejecución.
- El tipo de cada argumento y la longitud del arreglo que los contiene.
- El tipo de cada variable local y la longitud del arreglo que los contiene.

Lista de argumentos de longitud variable

El CLI y la biblioteca de clases trabajan en conjunto para implementar métodos que aceptan listas de argumentos de longitud y tipo desconocido. El acceso a esos argumentos es a través de un iterador conocido como System.ArgIterator.

El CLI incluye una instrucción especial para dar soporte al iterador de argumento, arglist. Esta instrucción podrá ser usada solo dentro de un método que en su declaración especificó que tomaría un número variable de argumentos. Regresa un valor que es usado por el constructor de un objeto de tipo System.ArgIterator. Básicamente, el valor creado por arglist brinda acceso a la dirección de la lista de argumentos y a una estructura de datos (creada en tiempo de ejecución) que especifica el número y el tipo de los argumentos que se pasaron al método. Esto es suficiente (desde el punto de vista de la biblioteca de clases) para implementar el mecanismo de iteración.

Desde el punto de vista del CLI, este tipo de métodos tienen un arreglo de argumentos como cualquier otro método. Pero solamente la porción inicial del arreglo tiene un conjunto de tipos fijo y solo ellos pueden ser accedidos mediante las instrucciones ldarg, starg y ldarga. El iterador de argumento permite el acceso al segmento inicial y al resto de las entradas del arreglo.

Memoria local

Como parte del estado de un método hay una porción de memoria local. La memoria que pertenece a esta sección puede ser alojada con la instrucción localloc. Este tipo de memoria se reclama en el momento en que el método termina de ejecutarse y es la única manera de hacerlo (no existe una instrucción que libere la memoria local durante la invocación del método). La memoria local se usa para alojar objetos cuyo tipo o tamaño se desconocen en tiempo de compilación y el programador no quiere alojar en el heap[10].

Flujo de control

El conjunto de instrucciones brinda instrucciones para alterar el flujo de control de una instrucción a la siguiente. Tenemos las siguientes instrucciones:

- Instrucciones de *salto condicional* e *incondicional* para ser usadas dentro de un método.
- Prefijos de recursión de cola, que indican que un método deberá liberar su registro de activación antes de ejecutar una llamada a función.
- Instrucción de regreso de un método, regresando un valor de ser necesario.
- Instrucciones de salto de método para transferir los argumentos del método actual a un método destino.

[10] *heap*: Mecanismos de administración de espacio disponible. A falta de una traducción que consideremos apropiada usaremos el término directamente en inglés.

- Instrucciones relacionadas con el manejo de excepciones. Incluye instrucciones para iniciar una excepción, transferir el control a una región protegida, terminar un filtro, una cláusula de captura o de final.

Aunque el CLI brinda soporte para la transferencia del control dentro de un método, hay varias restricciones que se deben observar:

- La transferencia de control no está permitida que sea a una clausula de captura o de final, excepto si es mediante el mecanismo de manejo de excepciones.
- La tranferencia del control fuera de una región protegida está permitida solamente si es a través de una instrucción de excepción (leave, endfilter, endcatch, endfinally).
- La pila de evaluación deberá estar vacía después de que el valor de regreso es eliminado por una instrucción de regreso (ret).
- Cada ranura de la pila deberá contener el mismo tipo de dato en cualquier punto dentro del cuerpo del método, sin importar el control de flujo que permita llegar a él.
- Para que los compiladores puedan conocer de manera eficiente los tipos de datos almacenados en la pila, la pila deberá estar vacía en la instrucción que le sigue a una instrucción de transferencia de control incondicional (br, br.s, ret, jmp, throw, endfilter, endcatch o endfinally). La pila podría no estar vacía en esa instrucción solamente si en algún punto anterior dentro del método hay un salto hacia esa instrucción.
- No está permitido que el control “simplemente” llegue hasta el final de un método. Todos los caminos de ejecución deberán terminar con alguna de las siguientes instrucciones: ret, throw, jmp o tail seguida de call, calli o callvirt.

Llamadas a métodos

Las instrucciones generadas por un generador de código CIL contienen suficiente información para que diferentes implementaciones del CLI usen diferentes convenciones de llamado. Todas las llamadas a métodos inician las áreas del estado del método, como sigue:

- Los elementos contenidos en el arreglo de argumentos son especificados por el invocador.
- El arreglo de variables locales siempre tiene null como valor, para objetos y para campos dentro de valores que contienen objetos. Además, si la bandera zero init se encuentra activada, entonces el arreglo de variables locales es iniciado a 0 para todos los tipos enteros y 0.0 para los tipos de punto flotante. Los valores no son iniciados por el CLI, sin embargo código

verificado proveerá un llamado a un iniciador como parte del punto de entrada del método.

- La pila de evaluación inicia vacía.

Descriptores del punto de invocación

Los puntos de invocación especifican información adicional que permite a un intérprete o a un compilador JIT (*Just In Time*) sintetizar cualquier convención de invocación nativa. Todas las instrucciones de invocación (call, calli y callvirt) incluyen una descripción del punto de invocación. Esta descripción puede tomar una de dos formas posibles. La forma más simple, usada con la instrucción calli, es una descripción del punto de invocación (representada como un átomo de metadatos para una firma de invocación autosuficiente) que brinda:

- El número de argumentos que se pasan.
- El tipo de dato de cada argumento.
- El orden en el cual han sido colocados en la pila de invocación.
- La convención de llamada nativa a ser usada.

La forma más complicada, usada por las instrucciones call y callvirt, es una referencia al método (un átomo de metadatos methodref) que aumenta la descripción del punto de invocación con un identificador del destino de la instrucción de invocación.

Instrucciones de invocación

El CIL tiene tres instrucciones de invocación que se usan para transferir nuevos valores de argumentos a un método destino. Bajo circunstancias normales, el método invocado terminará y el control regresará al método invocador.

- call está diseñado para ser usado cuando la dirección destino es fija en el momento en que se genera el CIL. En este caso se coloca una referencia a un método directamente en la instrucción. Esto es comparable a una invocación directa a un método estático en C. Puede ser usada para invocar métodos estáticos o de ejemplar o el método de superclase dentro del cuerpo de un método de ejemplar.

- calli está diseñada para usarse cuando la dirección destino se calcula en tiempo de ejecución. Se pasa un apuntador a función a la pila, y la instrucción contiene solamente la descripción del punto de invocación.

- callvirt usa el tipo exacto de un objeto (conocido solamente en tiempo de ejecución) para determinar el método a ser invocado. La instrucción incluye una referencia al método, pero el método en particular no es calculado sino hasta que la invocación ocurre. Esto permite que la instancia de una subclase sea enviada y el método apropiado para la subclase sea invocado. Se usa para ambos tipos de métodos: de ejemplar y de interfaz.

Además, cada una de estas instrucciones podrá ser precedida por el prefijo **tail**. Este prefijo le indica al compilador (JIT) que libere el estado del método invocador antes de realizar la invocación. Cuando el método invocado ejecuta una instrucción de regreso (**ret**), el control regresa a donde ese método lo haga, no necesariamente al método invocador.

Por último, hay una instrucción que indica una optimización del caso tail, y que es la instrucción jmp, seguida de un átomo methodref o methoddef, e indica que el estado del método actual deberá ser liberado, sus argumentos deberán ser transferidos intactos al método destino y el control deberá ser transferido al método destino. La firma del método invocador deberá ser identica a la firma del método invocador.

Destinos calculados

El destino de una invocación a un método puede estar codificado directamente en el flujo de instrucciones CIL (haciendo uso de call y jmp) o se puede calcular (si usamos callvirt y calli). La dirección destino para una instrucción callvirt es automáticamente calculada por el CLI basado en el átomo del método y el valor del primer argumento (el valor de this). El átomo del método deberá hacer referencia a un método virtual en una clase que es un ancestro directo de la clase del primer argumento. El CLI calcula el destino correcto localizando al ancestro más cercano de la clase a la cual pertenece el primer argumento y que ofrece la implementación del método deseado.

Para la instrucción calli el código CIL es responsable de calcular la dirección destino y de colocarla en el tope de la pila de evaluación. Esto se hace a través del uso de algunas de las instrucciones como ldftn y ldvirtfn en un momento anterior. La instrucción ldftn incluye un átomo de metadatos en el flujo de código CIL que especifica un método, y la instrucción carga la dirección del método. La instrucción ldvirtfn toma un átomo de metadatos para un método virtual en el flujo de código CIL y un objeto en la pila. Calcula el destino como lo hace callvirt y coloca la dirección en la pila en lugar de realizar la invocación.

La instrucción calli incluye una descripción del punto de invocación que incluye información sobre la convención de invocación nativa que debe ser usada. Un código CIL correcto deberá especificar una convención de llamado para una instrucción calli que sea igual a la convención de llamado del método.

Convención virtual de llamado

El CIL provee una convención virtual de llamado que el compilador convierte a una convención de llamado nativa. El compilador determina la convención de llamado óptima para la arquitectura en específico. Esto permite que exista flexibilidad en el uso de los registros, el hogar de las variables locales, convenciones de copiado de valores. Así mismo deja al compilador reordenar los valores colocados en la pila para igualar la posición y orden de los argumentos pasados a la llamada nativa. EL CLI usa una sola convención de llamado. Es responsabilidad del compilador convertir eso en la convención de llamada nativa apropiada.

El contenido de la pila en el momento en que una instrucción de llamado es invocada es:

1. Si el método que está siendo llamado es un método de ejemplar o un método virtual, el apuntador this es el primer objeto en la pila en el momento de la llamada a la instrucción de invocación. Para métodos en objetos, el apuntador this es de tipo O (objeto). Para métodos en valores, el apuntador this es provisto por un paso de parámetros por referencia; esto es, el valor es un apuntador (gestionado (&) o no gestionado (*) o un entero nativo) del ejemplar.

2. Los argumentos restantes aparecen en la pila en orden de izquierda a derecha.

Transferencia de parámetros

El CLI da soporte para tres maneras de transferencia de parámetros, todos ellos indicados por medio de metadatos como parte de la firma del método. Cada parámetro de un método tiene su propia convención de transferencia. Un parámetro podrá ser transferido en alguna de las siguientes maneras:

- Parámetros **por valor**, donde el valor de un objeto es transferido del invocador al invocado.

- Parámetros **por referencia**, donde la dirección del dato es transferida del invocador al invocado, y el tipo del parámetro es por lo tanto un apuntador gestionado o no gestionado.

- Parámetros por **referencia tipificada**, donde una representación en tiempo de ejecución del tipo de dato es transferida junto con la dirección del dato, y el tipo del parámetro es por lo tanto uno especificado para este propósito. Con parámetros por referencia y valor es suficiente para dar soporte a lenguajes de programación estáticos (C++, Pascal, etc.). También brindan soporte para lenguajes dinámicamente tipificados con un costo en el desempeño debido a la conversión de valores a referencias (boxing) antes de pasárselos a métodos polimórficos (Lisp, Scheme, Smalltalk, etc.). Desafortunadamente, estos recursos no son suficientes para dar soporte para lenguajes como VB que requieren transferencia de parámetros por referencia de datos (unboxed) a métodos que no son estáticamente restringidos con respecto al tipo de datos que reciben. Este lenguaje requiere de una manera de transferir la dirección del hogar de los datos y el tipo estático del hogar. Ésta es exactamente la información que sería provista si los datos fueran como referencias, pero sin el alojamiento en el *heap* requerido por la operación de envoltura (box).

Manejo de excepciones

El manejo de excepciones está soportado en el CLI a través de objetos de tipo excepción y bloques protegidos. Cuando una excepción ocurre, se crea un

objeto para representar la excepción. Todos los objetos excepción son ejemplares de algunas clases. Los usuarios podrían crear sus propias clases de excepciones, típicamente creando subclases de System.Exception.

Hay cuatro clases de manejadores para bloques protegidos. Un bloque deberá tener exactamente un manejador asociado:

- Un manejador de finalización (finally handler) que deberá ser ejecutado siempre que el bloque es abandonado, sin importar si ocurrió por un flujo de control normal o por una excepción no capturada.
- Un manejador de error, que deberá ser ejecutado si la excepción ocurre, en caso contrario termina como resultado de un control de flujo normal.
- Un manejador de excepciones de filtrado por tipo, que maneja cualquier excepción de una clase especificada o cualquiera de sus subclases.
- Un manejador de excepciones de filtrado definido por el usuario ejecuta un conjunto de instrucciones CIL especificadas por el usuario para determinar si la excepción debería ser ignorada, capturado por el manejador asociado ó transferida al siguiente bloque protegido.

  Las regiones protegidas, el tipo del manejador asociado, la posición del manejador y de ser necesario el código de filtrado se describen a través la tabla de manejadores de excepciones asociada a cada método.

Cada método tiene asociado un arreglo de información de manejadores de excepciones. Cada entrada en el arreglo describe un bloque protegido, su filtro y su manejador. Cuando una excepción ocurre, el CLI busca en el arreglo el primer bloque protegido que:

- Protege una región incluyendo al apuntador de la instrucción actual,
- Que es un manejador de captura y
- Cuyo filtro desea manejar la excepción.

Si no se encuentra al candidato adecuado en el método actual, se pasa a buscar en el método invocador y así sucesivamente. Si no encuentra al manejador adecuado, el CLI producirá una representación del estado de la pila y abortará el programa.

Si se encuentra, el CLI recorre la pila hasta el punto donde fue lanzada la excepción, pero esta vez invoca los manejadores de error y/o finalización. A continuación el manejador inicia su ejecución.

Hay que hacer hincapié en lo siguiente:

- El orden en el que se encuentran las cláusulas de excepción en la tabla es importante. Si éstas están anidados, el bloque protegido más anidado estará antes del bloque protegido que lo envuelve.

- Los manejadores de excepciones tienen acceso a las variables locales y al depósito de memoria local de la rutina que captura la excepción, pero cualquier resultado intermedio que estaba en la pila de evaluación en el momento en el que la excepción fue lanzada se pierde.
- Un objeto excepción que describe la excepción es creado automáticamente por el CLI y cargado a la pila de evaluación como el primer elemento en el momento en que un filtro o una cláusula de captura se inicia.
- La ejecución no puede ser reiniciada en la posición en que la excepción ocurrió, excepto a través de un manejador de filtrado definido por el usuario.

El CIL tiene construcciones especiales para:

- Lanzar (throw) o relanzar (rethrow) excepciones definidas por el usuario.
- Abandonar (leave) un bloque protegido y ejecutar la cláusula finalizadora apropiada dentro del método, sin lanzar la excepción. Esto también se usa para abandonar una cláusula de captura. Abandonar un bloque protegido no causa que la cláusula de error sea invocada.
- Terminar un filtro (endfilter) - provisto por el usuario - y regresar un valor indicando si la excepción debe ser manejada.
- Terminar una cláusula de finalización (endfinally) y continuar desenrollando la pila.

Un región protegida está descrita por dos direcciones: la dirección de la primera instrucción que estará protegida (trystart) y la dirección que le sigue a la última instrucción protegida (tryend). La región de un manejador está descrita por dos direcciones: la dirección de la primera instrucción del manejador (handlerstart) y la dirección que está inmediatamente después de la última instrucción del manejador (handlerend).

Existen tres tipos de manejadores: de captura (catch), final (finally) y de error (fault). Un entrada de excepción consiste de:

- Opcional: un átomo de tipo (el tipo de la excepción a manejar) o una referencia a la primera instrucción del código de filtrado (filterstart).
- Requerido: una región protegida.
- Requerido: un manejador.

Todo método tiene asociado un conjunto de excepciones. Ninguna región (protegida, del manejador, de filtrado) de una misma excepción podrán traslaparse entre sí.

Una vez que terminamos con las instrucciones del CLI pasamos ahora a describir nuestra aportación para el compilador de ECMAScript.

Capítulo 3

El lenguaje de programación ECMAScript

3.1. Antecedentes

En 1995 Netscape, con la intención de brindar soporte de manera más accesible a las aplicaciones web basadas en Java[1], contrató a Brendan Eich para que diseñara un lenguaje de scripting que permitiera tener acceso a todos los servicios, pero que además permitiera que diseñadores y desarrolladores de web pudieran usarlo fácilmente. La decisión fue crear un lenguaje con un sistema de tipos relajado, con el paradigma de programación orientado a objetos pero basado en prototipos, a diferencia de la mayoría de lenguajes orientados a objetos que están basados en el uso de clases. La primera versión fue bautizada como LiveScript pero posteriormente se cambió a JavaScript. Sin embargo, el lenguaje estandarizado lleva por nombre ECMAScript.

3.2. Características

ECMAScript se basa en objetos: el lenguaje básico y los servicios que brinda el sistema anfitrión son objetos y un programa ECMAScript es un cúmulo de objetos comunicándose. Un objeto en ECMAScript es una colección desordenada de propiedades con cero o más atributos que determinan cómo se usan. Las propiedades son contenedores que pueden alojar otros objetos, valores primitivos o métodos. Un valor primitivo es un miembro de los tipos interconstruidos Undefined, Null, Boolean, Number y String; un objeto es un miembro del tipo Object y un método es una función asociada a un objeto a través de una propiedad. ECMAScript define una colección de objetos interconstruidos que completan la definición de sus entidades. Estos objetos incluyen a los objetos Global, Object, Function, Array, String, Boolean, Number, Math, Date, RegExp y los objetos de

[1] *Java applets.*

error como Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError y URIError. La sintaxis de ECMAScript es intencionalmente muy parecida a la de Java, sin embargo permite construcciones más relajadas con el propósito de que sea fácil de usar; por ejemplo, en la declaración de una variable no se anota el tipo de ésta, como tampoco en el caso de las propiedades.

3.2.1. Modelo de ejecución.

El modelo de ejecución de ECMAScript está basado en la noción de contextos de ejecución. Los contextos de ejecución activos pueden formar una pila donde el que se encuentra en el tope se está ejecutando actualmente. Es importante mencionar que los tipos de código ejecutable que existen en ECMAScript son el global, de evaluación y de función. El global es texto que es tratado como un *programa*[2], pero éste no incluye el código fuente que es reconocido como el cuerpo de una función. El de evaluación es el texto que se le pasa como parámetro a la función interconstruida eval; si el parámetro que se le proporciona a la función eval es una cadena, se le trata como un programa; el código de una invocación a eval es el código global del parámetro. El de función es el texto que es reconocido como el cuerpo de una función, pero no incluye aquel texto que es reconocido como un cuerpo de función anidado. Con esto en mente, podemos decir que cada vez que el control es tranferido a código ejecutable ECMAScript, el control entra a un contexto de ejecución. Todo contexto de ejecución tiene asociado un objeto variable y las variables y funciones declaradas se agregan como propiedades del objeto variable; a este mecanismo se le conoce como creación de variables[3]. En el caso de código de función, los párametros se agregan como propiedades también. Depende del tipo de código que se use, para determinar cual objeto se usa como variable objeto así como los atributos para cada propiedad, sin embargo el resto del comportamiento es genérico. Cuando se entra a un nuevo contexto de ejecución, las propiedades se ligan en el siguiente orden:

- En el caso de código de función, por cada parámetro formal creamos una propiedad en el objeto variable cuyo nombre es su identificador y cuyos atributos se determinan por el tipo de código. Los valores de los parámetros los proporciona el invocador de la función; si el invocador proporciona menos valores que el número de parámetros formales, los parámetros formales tienen el valor de undefined.
- Por cada declaración de función, en el orden de aparición, creamos una propiedad del objeto variable cuyo nombre es el identificador en la declaración de función, su valor es el resultado de crear un objeto función y sus atributos se determinan por el tipo de código; si el objeto variable ya tiene una propiedad con ese nombre, se reemplaza su valor y atributos con los nuevos.

[2] De acuerdo a la definión BNF dada en ECMA-262.
[3] Del término en inglés *variable instantiation*.

- Por cada declaración de variable, se crea una propiedad en el objeto variable cuyo nombre es su identificador y su valor es indefinido; los atributos se determinan por el tipo de código.

Cada contexto de ejecución tiene asociado una cadena de alcance. La cadena de alcance es una lista de objetos en la cual se puede resolver la aparición de un identificador. Cuando el control entra en el contexto de ejecución nuevo, una cadena de alcance se crea y es ocupada con un conjunto inicial de objetos, dependiendo del tipo de código. Durante la ejecución en un contexto en específico, la cadena de alcance puede ser afectada solamente por un enunciado with o cláusulas catch.

Existe un objeto global único, el cual se crea antes de que el control entre a algún contexto de ejecución. En principio el objeto global tiene las siguientes propiedades:

- Los objetos interconstruidos Math, String, Date, etc. Éstos tienen los atributos DontEnum.
- Propiedades definidas por el sistema anfitrión.

Conforme se entra a nuevos contextos de ejecución y ésta se realiza, podrían agregarse propiedades adicionales al objeto global así como las propiedades iniciales podrían cambiar.

Cuando el control entra a un contexto de ejecución de función, se crea y asocia a él un objeto conocido como el objeto de activación. El objeto de activación se inicia con una propiedad de nombre arguments y atributo DontDelete. Su valor inicial es el objeto argumento. El objeto argumento se crea e inicia de la siguiente manera:

- El valor de su propiedad interna Prototype es Object.prototype.
- Se crea una propiedad de nombre callee y atributos DontEnum. El valor inicial de esta propiedad es el objeto función que está siendo ejecutado. Esto permite que las funciones anónimas puedan ser recursivas.
- Se crea otra propiedad de nombre length y atributos DontEnum. El valor inicial de esta propiedad es el número de parámetros que fueron proporcionados por el invocador.
- Por cada número entero no negativo *arg*, menor que length, se crea una propiedad con ese nombre y atributos DontEnum. El valor inicial de esa propiedad es el valor correspondiente del parámetro provisto por el invocador.

Todo contexto de ejecución tiene asociado un valor para this. Su valor depende del invocador y el tipo de código a ejecutarse y se determina cuando se entra al contexto de ejecución. El valor de this asociado a un contexto de ejecución es inmutable.

Toda llamada a función y constructor entra a un nuevo contexto de ejecución, aún si la función se está llamando recursivamente. Cada enunciado de regreso libera un contexto de ejecución. Una excepción lanzada, si no es manejada, podría también liberar uno o más contextos de ejecución.

Cuando el control entra a un nuevo contexto de ejecución, la cadena de alcance se crea e inicia, se crean las variables y se determina el valor de this.

Dado que esas operaciones dependen del tipo de código, se realiza de la siguiente manera:

- **Código global**
    - La cadena de alcance se crea e inicia conteniendo unicamente el objeto global.
    - La creación de variables se realiza usando al objeto global como variable objeto y usando los atributos DontDelete.
    - El valor de this es el objeto global.
- **Código de evaluación.** Cuando el control entra a un contexto de ejecución de evaluación, el contexto previo, al cual nos referiremos como el contexto invocador, se usa para determinar la cadena de alcance, objeto variable y el valor de this. Si no existe un contexto invocador entonces todo esos valores se toman como si estuviéramos en un código global.
    - La cadena de alcance se inicia conteniendo los mismos objetos y en el mismo orden que la cadena de alcance del contexto invocador. Esto incluye objetos agregados con with y catch.
    - La creación de variables se realiza usando al contexto invocador como objeto variable y se usan atributos vacíos.
    - El valor de this es el mismo que el del contexto invocador.
- **Código de función.**
    - La cadena de alcance inicia conteniendo el objeto de activación seguido por los objetos en la cadena de alcance del objeto función.
    - La creación de variables se realiza usando el objeto de activación como el objeto variable y atributos DontDelete.
    - El invocador provee el valor de this. Si ese valor es nulo o no es un objeto entonces se le asigna el valor del objeto global.

3.2.2. Programación orientada a objetos basada en prototipos.

ECMAScript no posee clases propiamente como aquellas en C++, Smalltalk o Java; sin embargo, se proporcionan constructores que crean objetos ejecutando código que se encarga de alojar el espacio para él, así como también asigna

valores iniciales a todas sus propiedades. Todos los constructores son objetos, pero no todos los objetos son constructores. Cada constructor tiene una propiedad Prototype que se usa para implementar herencia; todo objeto creado por un constructor tiene una referencia implícita al prototipo (conocido como el prototipo de objeto) asociado con su constructor. Además, un prototipo podría tener una referencia implícita no nula a su prototipo y así sucesivamente; a lo que se conoce como cadena de prototipos. En los lenguajes orientados a objetos basados en clases, en general, el estado se mantiene en los ejemplares, los métodos en las clases, siendo la herencia sólo de estructura y comportamiento. En ECMAScript, el estado y los métodos están en los objetos y la estructura, comportamiento y estado se heredan. Todo objeto que no contiene directamente una propiedad particular que su prototipo contiene, comparte esa propiedad y su valor. A diferencia de lenguajes orientados a objetos basados en clases, los lenguajes orientados a objetos basados en prototipos pueden agregar propiedades a objetos dinámicamente simplemente asignándole un valor.

3.2.3. Ligado lo más tarde posible

El mecanismo de ligado lo más tarde posible[4] consiste en permitir que el proceso de resolución de una invocación a un método, el acceso a una propiedad, el uso del nombre de una variable en el contexto de una aplicación de método, entre otras cosas, se realice en el momento de ejecutar el programa y no durante la compilación del mismo, en caso de ser necesario. El uso de este mecanismo— consecuencia del diseño del lenguaje con un sistema de tipos dinámico en el cual no existen reglas sintácticas para la anotación de tipos explícita de las variables o funciones - estimula un estilo de programación más flexible y menos verboso, pero también abre la posibilidad de errores de tipo en tiempo de ejecución. Hay que mencionar que no todas las expresiones requieren evaluación lo más tarde posible; el compilador intenta al máximo resolver todas las construcciones y así generar código específico para su ejecución; cuando esto no es posible se recurre a este tipo de evaluación. Existen técnicas para la inferencia de tipos, aunque esto no se incluye en este compilador, siendo esa tarea un buen candidato para trabajo futuro.

3.2.4. Funciones

Las funciones en ECMAScript juegan un papel muy importante ya que pueden ser usadas y definidas de distintas maneras. Empezaremos por describirlas desde el punto de vista sintáctico y luego pasaremos a describir sus propiedades como tipo de datos.

En la especificación ECMA-262 tercera edición de ECMAScript en la sección de construcciones gramaticales está definido cómo se forma una declaración de función, que usando sintaxis BNF es de la siguiente manera:

[4]Del término en inglés *late binding*.

$$FunctionDeclaration ::=$$
$$\textbf{function}\ Identifier\ ((Identifier\ (,\ Identifier)^*)\ |\ )\ \{\ (SourceElement)^*\ \}$$

donde *SourceElement* puede ser una declaración de función anidada o un *Statement.* Debido a que ECMAScript es un lenguaje dinámicamente tipificado, en la especificación no se incluye verificación alguna sobre el tipo esperado de los parámetros; con respecto al número en caso de que sean menos se asume que el valor de los faltantes es indefinido y en caso de ser más simplemente se ignoran los que sobran.

La declaración de función no es la única manera de definir funciones nuevas. ECMAScript permite definir funciones dinámicamente mediante el constructor de funciones, quien espera cualquier número de cadenas como argumentos, con la característica de que el último parámetro es el cuerpo de la función (este último parámetro puede contener un número no definido de enunciados separados por punto y coma). Los otros parámetros especifican los nombres de los parámetros de la función que estamos definiendo. Una cosa que cabe recalcar es que en ningún momento se especifica el nombre de la función creada de esta manera, por lo que se crea una función anónima.

Otra manera de definir funciones es como literales. Una literal de función es una expresión que define funciones sin nombre. La sintaxis de una literal de función es muy similar a la de una declaración de función, pero con la diferencia que se usa en el contexto de una expresión y no como un enunciado, además de que el nombre es opcional.

Funciones como datos

Una de las características más importantes de las funciones de ECMAScript es que pueden ser tratadas como datos. Es decir, se pueden asignar a variables, a propiedades de objetos, a elementos de un arreglo y ser usadas como parámetros de otras funciones. El listado 3.2.1 muestra un ejemplo de este tipo de uso.

Las funciones crean su propio ambiente de ejecución (que es local). Este ambiente se crea agregando el objeto invocador a la cadena de alcance. Debido a que el objeto invocador es parte de la cadena de alcance cualquier propiedad de ese objeto es accesible como variable dentro del cuerpo de la función; las variables locales declaradas se crean como propiedades de ese objeto y los parámetros de la función también están disponibles como propiedades del objeto. Además de las variables y parámetros, el objeto invocador define una propiedad especial que se llama *arguments.* Esa propiedad hace referencia a un objeto especial conocido como el objeto Arguments. El objeto Arguments es como un arreglo que permite obtener mediante índices los valores de los argumentos que se le pasan a una función, sin ser un objeto de tipo arreglo. Una de las propiedades del objeto arguments es length que nos dice el número de elementos que contiene. Además de poder acceder a sus elementos, el objeto arguments define la propiedad callee que hace referencia a la función que está siendo invocada actualmente;

Listado 3.2.1 Funciones como datos.

```
function Suma (x, y) { return x + y; }
function Resta (x, y) { return x - y; }
function Multiplica (x, y) { return x * y; }
function Divide (x, y) { return x / y; }

function Opera (operador, operando1, operando2)
{
  return operador (operando1, operando2);
}

var i = Opera (Suma, Opera (Suma, 2, 3),
                     Opera (Multiplica, 4, 5));
```

esta característica es importante para poder crear funciones anónimas recursivas. El listado 3.2.2 muestra la implementación recursiva de la función factorial haciendo uso de la propiedad callee.

Listado 3.2.2 Los miembros arguments y callee de una función.

```
function (x)
{
  if (x <= 1)
    return 1;
  else
    return x * arguments.callee (x - 1);
}
```

Como explicamos arriba, las funciones pueden usarse como datos y pueden ser creadas mediante un constructor. Esto nos indica que las funciones en realidad se representan mediante algún tipo de objeto, el objeto Function. Como consecuencia, las funciones tienen propiedades y métodos como cualquier otro objeto. La propiedad length del objeto Function tiene un significado un poco diferente con respecto a arguments.length, ya que en este caso nos indica el número de argumentos que la función espera que se le pasen al momento de una invocación. Toda función tiene definidos dos métodos call y apply. Estos métodos permiten invocar una función como si fuese el método de otro objeto. El primer argumento de ambas funciones es el objeto sobre el cual se realizará la invocación de función, que se convierte en el valor de this dentro del cuerpo de la función; todos los argumentos adicionales que se pasan a call se convierten en los argumentos de la función que se está invocando. *apply* es similar a call con la diferencia de que los argumentos se le pasan en un arreglo.

3.2.5. Objetos

En ECMAScript un objeto es una colección no ordenada de propiedades, cada una de ellas con un nombre y un valor. Los valores pueden ser valores primitivos u objetos. Los objetos se crean mediante el operador new, que viene seguido por el nombre de un constructor que sirve para iniciar el objeto. Diferentes constructores reciben parámetros distintos, y podemos definir nuestros propios constructores. Existe otra manera de crear objetos, esto es mediante las literales objeto; ellas nos permiten expresar explícitamente la lista de propiedades y sus valores. Cada especificación de propiedad en una literal de objeto consiste del nombre de la propiedad seguida por dos puntos y el valor. El listado 3.2.3 muestra cómo crear objetos a través de un constructor o mediante una literal de objeto.

Listado 3.2.3 Objetos y literales objeto.

```
var obj = new Object ();
var obj_lit = { x : 1, y : ''hola''};
```

Para acceder a las propiedades de un objeto usamos el operador '.'. El valor del lado izquierdo del '.' debe ser una referencia a un objeto (generalmente basta con el nombre de la variable que contiene la referencia al objeto). El valor a la derecha es el nombre de la propiedad. La propiedad está representada por un identificador, no una cadena o expresión. El ejemplo en el listado 3.2.4 es particularmente interesante ya que como vemos estamos creando una propiedad y posteriormente a la nueva propiedad podemos agregarle propiedades también. Esto se traduce en un uso intenso de evaluación lo más tarde posible en el código CIL al cual se está compilando.

Listado 3.2.4 Objetos dinámicos.

```
var casa = new Object ();
casa.cuarto = new Object ();
casa.cuarto.escritorios = 3;
```

Hay que enfatizar que la única manera de crear propiedades nuevas en un objeto es asignando un valor a ellas y podemas cambiar su valor en cualquier momento asignando un valor nuevo. También podemos eliminar completamente una propiedad antes creada mediante el operador *delete*, como se muestra en el listado 3.2.5.

Listado 3.2.5 Eliminar propiedades.

```
delete obj.prop;
```

Constructores y métodos

Mencionamos que podemos crear e iniciar objetos usando el operador new en conjunción con constructores predefinidos como Object (), Date (), entre otros. Sin embargo, tenemos la posibilidad de crear nuestros propios constructores que nos permitan manipular un tipo especial o clase de objetos. Un constructor en ECMAScript es una función con dos características especiales:

1. Se invoca a través del operador new.
2. Se le pasa como parámetro un objeto recién creado y vacío como valor de this y es responsable de asignar los valores por omisión de las propiedades del objeto nuevo, como se muestra en el listado 3.2.6.

Listado 3.2.6 Creando nuevos constructores.

```
function Esfera (x, y, z, radio)
{
  this.x = x;
  this.y = y;
  this.z = z;
  this.radio = radio;
}

var esfera = new Esfera (1, 2, 3, 5);
```

Un método es una función que se invoca a través de un objeto. Sin embargo, debido a que las funciones de ECMAScript son datos, no hay nada especial con respecto a los nombres con los que se definen—una función se puede asignar a una variable o a la propiedad de un objeto. Si tenemos una función *g* y un objeto *obj*, podemos definir un método *meth* como se muestra en el listado 3.2.7. y podemos invocarlo, como en el listado 3.2.8.

Listado 3.2.7 Asignación de una función como método miembro.

```
obj.meth = g;
```

Listado 3.2.8 Invocación del nuevo método miembro.

```
obj.meth ();
```

Una característica importante es que el objeto a través del cual se invoca el método se convierte en el valor de *this* dentro del cuerpo del método.

Prototipos y herencia

En ECMAScript podemos crear métodos, constantes y propiedades que se comparten entre objetos de una misma clase. Los objetos en ECMAScript heredan propiedades de su objeto prototipo. Todo objeto tiene un prototipo y todas las propiedades del prototipo aparecen como propiedades de los objetos de los cuales es prototipo. El prototipo de un objeto lo define el constructor que fue usado para crear e iniciar el objeto. Todas las funciones en ECMAScript tienen una propiedad prototype que hace referencia a un objeto. En principio ese objeto se encuentra vacío, pero cualquier propiedad que se le defina será compartida por los objetos creados por el constructor.

La herencia ocurre en el momento de resolver el valor de una propiedad. En ningún momento se copian las propiedades del prototipo a los nuevos objetos, sino que cuando se intenta leer una propiedad p de un objeto *obj*, en primera instancia se verifica si *obj* tiene definida esa propiedad. Si no es el caso, se verifica si el prototipo de *obj* tiene esa propiedad. De esta manera es como funciona la herencia basada en prototipos.

Sin embargo hay que agregar que la herencia ocurre sólo en el caso de la lectura de propiedades, no cuando se agregan. Si agregamos o alteramos una propiedad a un objeto, ella se agrega solamente al objeto en cuestión no al prototipo; cuando queremos volver a leer esa propiedad, debido a la forma en que se resuelven las referencias a propiedades, la propiedad nueva agregada al objeto siempre oculta a la propiedad con el mismo nombre que se encuentra en el prototipo.

Pasamos ahora a describir la manera en que las características de ECMAScript se implementan en el CLI.

Capítulo 4

ECMAScript en el CLI

El sistema de objetos y el modelo de ejecución de ECMAScript y el CIL poseen características que nos obligan a crear un mecanismo de interoperabilidad entre ambos. Algunas de esas características son:

- En ECMAScript podemos tener código ejecutable a nivel global, a diferencia del CIL donde todo el código ejecutable tiene que estar dentro de un método y ese método dentro de una clase.
- En ECMAScript el código global de un programa es el que se ejecuta, mientras que en el CIL se necesita especificar el punto de entrada de la ejecución.
- En ECMAScript los objetos son dinámicos a diferencia de la clases estáticas del CIL.
- En ECMAScript podemos pasar funciones como parámetros de otras funciones[1].
- En ECMAScript las funciones pueden regresar funciones [2].

4.1. Implementación del modelo de ejecución

En la sección 3.2.1 se mencionó la especificación abstracta del modelo de ejecución de ECMAScript. Sin embargo, es necesario mencionar algunos otros aspectos involucrados en la implementación. En primer lugar no olvidemos que el código fuente se encuentra en archivos, pero debido a que podemos tener código global en cada uno de ellos y necesitamos llevar a cabo el proceso de creación de variables, se realiza el siguiente mapeo:

[1] En CIL podemos hacerlo a través de delegates sin embargo eso no implica que las funciones sean de primera clase

[2] En CIL podemos hacerlo a través de delegates sin embargo eso no implica que las funciones sean de primera clase

- El código fuente contenido en un archivo se mapea a una clase que extiende a la clase Microsoft.JScript.GlobalScope - que entre otras nos brinda la facilidad de agregar y borrar propiedades, lo que en la especificación formal es el objeto variable—y cuyo nombre viene dado por la concatenación de la cadena "JScript " y un número mayor o igual que cero que se va incrementando secuencialmente[3].

- Dicha clase siempre tiene como miembro un método de ejemplar llamado "Global Code" y en éste se encuentran codificadas las operaciones que se encuentran a nivel global.

- El mecanismo de creación de variables se lleva a cabo de la siguiente forma:

  - Los parámetros formales de una función se codifican como tales en su equivalente en CIL.
  - Con respecto a la declaración de variables, se tienen dos casos:
    - Las variables declaradas a nivel global pasan a ser campos estáticos de la clase.
    - Las variables declaradas en el cuerpo de una función se codifican como variables locales.

- En el caso de la declaración de funciones se divide entre el caso global y el caso anidado:

  - En el caso global, se crea un campo estático de tipo ScriptFunction y además se añade a la clase un método estático, que además de incluir los parámetros formales ya definidos en el código fuente tiene dos parámetros formales extras codificados del lado del CIL; estos parámetros son: el valor que tomará this dentro de la función y el otro representa el motor de ejecución[4] asociado a la clase "JScript N"; ambos llevan por nombre el nombre de la función declarada.
  - En el caso anidado, se crea una variable local de tipo ScriptFunction y además se añade un método estático cuyo nombre está formado por la concatenación de los nombres de las funciones que contienen a la declaración actual, cada nombre separado por un punto; se agregan los dos parámetros formales extras como en el caso anterior.

- Se crea una clase privada con nombre "JScript Main". La tarea que cumple dicha clase es la de ser el punto de entrada para la ejecución del código global compilado; es decir, crear un ejemplar de cada una de las clases "JScript N" con un ejemplar de GlobalScope nuevo y que será común a todos; luego invocará el respectivo método "Global Code" de cada uno de los ejemplares.

[3]En ésta implementación no manejamos de manera adecuada varios archivos, por lo cual se asume que el código a compilar se encuentra en un único archivo.

[4]El motor de ejecución se llama *VsaEngine*, de la contracción *Visual Studio for Applications Engine*.

La cadena de alcance se implementa a través de los métodos PushScriptObject, PopScriptObject y ScriptObjectStackTop de la clase VsaEngine.

4.2. Implementación del ligado lo más tarde posible

Empezaré por decir qué tipo de expresiones requieren ligado lo más tarde posible: aquellas que involucran acceso a miembros del objeto mediante el uso del operador punto y las expresiones de acceso a través de un índice (cuando usamos un objeto como si fuera un arreglo); éstas a su vez diferencian los casos de lectura y asignación para cada una de ellas, así como expresiones de invocación a función que no corrresponden a métodos del objeto global y métodos asociados al sistema de objetos.

Existe en la interfaz pública de aplicación del espacio de nombres Microsoft.JScript una clase llamada LateBinding cuyo objetivo es el almacenar los componentes involucrados en una expresión que requiere de ligado lo más tarde posible, que vienen a ser el lado izquierdo y derecho de las expresiones arriba mencionadas, donde el lado izquierdo es el objeto sobre el cual se intentará realizar la operación y el lado derecho es el nombre del miembro (cuando se está usando el operador punto) ó el índice o cadena (cuando es un acceso estilo arreglo); además esta clase exporta un conjunto de métodos para llevar a cabo este tipo de evaluación, el listado 4.2.1 muestra las firmas de los métodos que dan el soporte para la evaluación lo más tarde posible.

Listado 4.2.1 Soporte de ejecución de la evaluación lo más tarde posible.

```
public object Call (object [] arguments, bool construct,
                    bool brackets, VsaEngine engine)

public static object CallValue (object thisObj,
                                object val,
                                object [] arguments,
                                bool construct,
                                bool brackets,
                                VsaEngine engine)

public object GetNonMissingValue ()

public static void SetIndexedPropertyValueStatic (
                             object obj,
                             object [] arguments,
                             object value)

public void SetValue (object value)
```

Los criterios para la generación de código de ligado lo más tarde posible son los siguentes:

- Para el acceso a un campo, tenemos una expresión binaria en la cual ambos elementos—el lado izquierdo y derecho—son identificadores tenemos la posibilidad de verificar que se trate de un acceso a una propiedad estática de un objeto que pertenece al sistema de objetos de ECMAScript; en dicho caso se genera código específico de lectura o escritura a dicha propiedad. Si alguno de los elementos no es un identificador se genera código de ligado lo más tarde posible, que en el caso de lectura involucra a GetNonMissingValue y de escritura a SetValue.
- Para la invocación de método, verificamos que el método a invocarse pertenezca al objeto global o sea método de un objeto del sistema de objetos de ECMAScript y se genera código explicito para realizar la invocación; en caso contrario se genera código de ligado lo más tarde posible que involucra a Call y CallValue.
- Para el acceso mediante índice, siempre se genera código de ligado los más tarde posible, que involucra a SetIndexedPropertyValueStatic y SetValue.

Los métodos Call y CallValue se encargan del acceso a una propiedad que luego es usado como aplicación de función o acceso a la propiedad mediante un índice. La diferencia entre estos dos métodos es que al último se le pasa explícitamente el valor de this y el objeto con el cual se realizará la aplicación. GetNonMissingValue se encarga del acceso para lectura de alguna propiedad así como SetValue realiza la asignación, en todas ellas se verifica que se cumplan todos los demás requisitos necesarios para realizar la operación; por ejemplo, que exista la propiedad referenciada o en su defecto que extienda al objeto involucrado con una nueva propiedad nombrada así e iniciarla a su valor por omisión.

4.3. Implementación de funciones

Como se mencionó en la sección 3.2.4 podemos ver las funciones de ECMAScript como funciones declaradas estáticamente o como tipo de datos. Es por esto que en la fase de generación de código tenemos diferencias también. Durante la descripción de las funciones en la sección antes mencionada no se hizo mención explícita a un conjunto de características que repercuten en la forma en que se implementan este tipo de funciones; éstas son:

1. ECMAScript es un lenguaje con alcance léxico, es decir, cuando se tienen regiones anidadas, las declaraciones más profundas toman precedencia sobre las que se encuentran en regiones exteriores.
2. ECMAScript es un lenguaje de programación con ligado estático, es decir, que la relación entre la referencia de una variable y su declaración es una propiedad estática.

Además de un conjunto de propiedades conocidas en la literatura de lenguajes de programación:

- El valor de una expresión depende solamente de los valores asociados a las variables que ocurren libres dentro de la expresión.
- El contexto en el cual se encuentra dicha expresión debe proveer esos valores.

4.3.1. Declaración de funciones

El proceso de compilación involucra varias fases: reconocimiento léxico, reconocimiento gramatical, análisis semántico y generación; en esta subsección describiremos los aspectos relacionados con los dos últimos. En el primero se asocian las referencias de variables a sus declaraciones y en el segundo se crea la representación en CIL del código fuente ECMAScript.

- En la fase de asociación de referencias, cuando nos encontramos en un bloque de enunciados, realizamos dos pasadas sobre el árbol de sintaxis abstracta:

  1. Por cada enunciado de declaración de variable o de declaración de función en el bloque, agregamos su identificador a una tabla; si el enunciado es de cualquier otra forma lo ignoramos en esta fase.
  2. Para cualquier otro elemento resolvemos las posibles referencias que pueda hacer. Si es una declaración de función, se crea una nueva capa de alcance que perdura hasta el final del cuerpo de la función, ya que al llegar éste se elimina la última capa de alcance creada.

  Para este propósito usamos una tabla de símbolos como la que Appel [4] presenta y que consiste de una clase que asocia una referencia con su declaración[5] y una tabla de identificación que brinda un mecanismo de creación y eliminación de ambientes de alcance según las reglas del alcance léxico.

  Esto nos permite implementar la semántica de alcance léxico y ligado estático que mencionamos arriba.

- La generación de código se realiza también en un orden especial:

  1. Se crean los campos de la clase que representa nuestro bloque o si estamos dentro de una función creamos una variable local.
  2. Se crean los campos y métodos que representan una declaración de función, así como la construcción de la *cerradura*.
  3. Por último se genera el código del resto de enunciados y expresiones en el bloque.

[5] Del término en inglés *binder*.

Una de las partes más interesantes es cómo se construye el ambiente cerrado. De acuerdo con Friedman [7] cuando se aplica una función, su cuerpo se evalúa en un ambiente que liga los parámetros formales de la función a los argumentos de la aplicación. Las variables que ocurren libres en la función también deben obedecer la regla del alcance léxico. Eso requiere que retengan el ligado que estaba actuando en el momento en que la función se creó. Para que una función retenga el ligado que sus variables libres tenían en el momento que fue creada, debe ser un paquete cerrado, independiente del ambiente en el cual se usa. A tal paquete se le conoce como *cerradura*[6]. Para que sea autocontenida, una *cerradura* debe contener el cuerpo de la función, las lista de parámetros formales y el ligado de las variables libres.

La mayoría de toda esa información se obtiene durante el proceso de análisis sintáctico; esa información consiste de:

- Nombre local de la función.
- Nombre completo de la función, que en caso de funciones anidadas incluye como prefijo los nombres de las funciones que lo contienen concatenadas y separados por puntos.
- Un arreglo con los nombres de los parámetros formales.
- Un arreglo de JSLocalField's formado por los parámetros formales y cualquier variable local o función declarada dentro de la función.
- Una representación como cadena de la función, que incluye todos y cada uno de los carácteres que aparecieron en la declaración.

Todo esto con el fin de construir la cerradura a través del método JScriptFunctionDeclaration de la clase FunctionDeclaration, que luego es almacenado en el campo o variable local de tipo ScriptFunction que se genera, además del método mismo, del lado del CIL.

4.3.2. Funciones como tipo de datos

En la sección 3.2.4 se mostró cómo las funciones en ECMAScript pueden pasarse como argumentos de otras funciones, aparecer como valores de regreso y ser almacenadas en estructuras de datos, es decir, cumplen la definición de objeto de primera clase dada por Friedman [7]. Debido a las características requeridas de las funciones en ECMAScript y que no se tienen en el CLI directamente, se implementó de la siguiente manera:

En 4.3.1 se dijo que se crea un campo y un método cada vez que encontramos una declaración de función. De gran importancia es que el campo sea de tipo ScriptFunction del cual es subclase Closure, que es el tipo del objeto que se le asigna al campo cuando se construye la cerradura. Cada vez que se quiere realizar una invocación se utiliza el método creado y cuando se quiere pasar como parámetro utilizamos el campo.

[6]Del término en inglés *closure*.

Veamos por ejemplo una invocación del método Opera, que se muestra en el listado 4.3.1.

Listado 4.3.1 Funciones como parámetros.

```
Opera (Suma, 2, 3);
```

Esta invocación se codifica en el código CIL como se muestra en el listado 4.3.2, donde se aprecia como Suma es usado desde el campo mediante la instrucción ldsfld[7] y Opera desde el método mediante la instrucción call.

Listado 4.3.2 Código CIL del paso de una función como parámetro.

```
callvirt instance object class IActivationObject::
                                        GetDefaultThisObject()
ldarg.0
ldfld   class VsaEngine ScriptObject::engine
ldsfld   class ScriptFunction 'JScript␣0'::'Suma'
ldc.r8 2
box Double
ldc.r8 3
box Double
call object 'JScript␣0'::Opera(object, VsaEngine,
                                    object, object, object)
```

Para los casos de funciones como valores de regreso de funciones y asignación de un método a una estructura de datos, usamos el campo estático también.

Este mecanismo nos permite simular las funciones de primera clase de ECMAScript en el CLI.

Pasamos ahora a revisar algunos de los algoritmos usados para la implementación de ECMAScript en el CLI.

[7]Del término en inglés *load static field*.

Capítulo 5

Algoritmos utilizados en el compilador

El escribir un compilador es una tarea divertida e interesante ya que involucra el uso conjunto de estructuras de datos y algoritmos que se estudian durante la licenciatura pero con propósitos mas específicos como podría ser la evaluación más eficiente de ciertas expresiones, el minimizar el uso de los recursos de memoria, el reconocimiento sintáctico, entre otros. Es por ello que este capítulo se dedica a hacer un recuento de estos algoritmos, en cuanto a que afectan el desempeño del compilador o los tiempos de ejecución de un programa dado.

5.1. Análisis amortizado

Muchos de los algoritmos usados durante el proceso de compilación involucran el uso de estructuras de datos que crecen y decrecen bajo ciertos criterios. Un análisis tradicional indicaría que la complejidad en el peor de los casos de la realización de n operaciones es de orden cuadrático; sin embargo esa cota es demasiado holgada; existe una técnica llamada análisis amortizado que nos permite encontrar cotas mucho mas precisas, además de ayudar a comprender el costo general de todas las operaciones que se realizan sobre una estructura dada.

La manera intuitiva de ver el análisis amortizado es la siguiente: el tiempo requerido para realizar un conjunto de operaciones se promedia entre todas las operaciones hechas, para tomar en cuenta de mejor manera los estados por los que pasa esa estructura. El análisis amortizado se usa para mostrar que el costo promedio de una operación es bajo, si se hace el promedio sobre el conjunto de operaciones, aunque una operación en la sucesión de operaciones llegara a ser costosa. El análisis de peor caso asume que la estructura siempre se encuentra en un estado tal que cada operación es lo más costosa posible, pero esto no es cierto, ya que la estructura pasa por diversos estados. Una de sus diferencias con respecto al análisis del caso promedio es que no involucra el uso de probabilidad;

el análisis amortizado garatiza el desempeño promedio de cada operación en el peor de los casos.

Existen varios métodos para hacer este tipo de análisis, algunos de los más conocidos son:

- **Análisis de agregación**. Con este método se muestra que para cualquier número entero n, una sucesión de n operaciones toma $T(n)$ de tiempo total en el peor de los casos. En el peor de los casos, el costo promedio, o *costo amortizado* por operación es $T(n)/n$, aunque hay que mencionar que este costo se aplica a cada una de las operaciones, aún y cuando existan diferentes tipos de operaciones en la sucesión.

- **Método por conteo**. En este método se asignan costos diferentes a operaciones distintas, algunas con mayor o menor costo de lo que realmente tienen. La cantidad que cuesta una operación se llama *costo amortizado.* Cuando el costo amortizado de una operación excede su costo real la diferencia se asigna a otro objeto específico como *crédito.* Ese crédito se usa más adelante para pagar el costo de las operaciones cuyo costo amortizado es menor que su costo real.

- **Método de potenciales**. En este método en lugar de representar el trabajo prepagado como crédito asignado a objetos específicos dentro de la estructura de datos, se representa como *energía potencial* que puede ser liberada para pagar operaciones futuras. El potencial está asociado a la estructura de datos completa, no a objetos específicos dentro de la estructura de datos. A grandes rasgos lo que se tiene es una función que mapea una estructura de datos D_i a un número real $\Phi(D_i)$ que es el potencial asociado con la estructura de datos D_i. El costo amortizado ca_i de la i-ésima operación con respecto a la función potencial Φ se define como $ca_i = c_i + \Phi(D_i) - \Phi(D_i - 1)$ pues refleja los cambios de potencial en la estructura de datos.

5.1.1. Tablas dinámicas

Las tablas o arreglos dinámicos son de particular importancia para los algoritmos que presentaré más adelante, ya que algunos de ellos involucran el uso de este tipo de estructura de datos. Aquí sólo menciono un resultado cuya demostración se puede encontrar en [5] sección 17.4.2:

> Dada una tabla dinámica con operaciones de inserción y borrado que duplica su tamaño cuando todos los lugares han sido ocupados y reduce su tamaño a la mitad cuando menos de 1/4 de la tabla está siendo ocupada; el costo amortizado de cada operación se encuentra limitada por arriba por una constante y el tiempo que se toma para cualquier sucesión de n operaciones es de $O(n)$.

5.2. Algoritmos para análisis sintáctico

El análisis sintáctico es la etapa inicial del proceso de compilación. Esta fase se subdivide en tres etapas que son: análisis léxico, reconocimiento sintáctico y la construcción de una representación de la estructura del programa. El análisis léxico se encarga de transformar el código fuente del programa en un flujo de átomos, tales como identificadores, literales, operadores, palabras reservadas y símbolos de puntuación, además de eliminar los comentarios y espacios que tuviera el programa. La base teórica que sustenta al análisis léxico es la teoría de los autómatas finitos y lenguajes regulares. La siguiente etapa—el reconocimiento sintáctico—trabajá con el flujo de átomos y trata a cada uno de ellos como símbolos terminales y se encarga de determinar la estructura del programa creando unidades sintácticas mayores como son enunciados, sucesiones de enunciados, expresiones, declaraciones, entre otras; en caso de violar las reglas sintácticas del lenguaje se reporta un error; en ausencia de errores se procede a construir la representación de la estructura del programa. Dicha representación es típicamente un árbol de sintaxis abstracta que posteriormente es usado por las siguientes fases del compilador.

Se suele dividir en dos al conjunto de estrategias para realizar el reconocimiento sintáctico que son: *abajo-hacia-arriba* y *arriba-hacia-abajo*. Ellas se caracterizan por la manera en que construyen el árbol de sintaxis abstracta correspondiente al flujo de entrada. El primero de ellos examina los símbolos terminales del flujo de entrada en orden de izquierda a derecha y reconstruye el árbol a partir de los nodos terminales hacia el nodo raíz. El segundo hace lo mismo con el flujo de entrada, pero reconstruye el árbol de sintaxis a partir de la raíz hacia abajo, hasta llegar a símbolos terminales.

En particular en este trabajo se utilizó una técnica arriba-hacia-abajo muy conocida y usada: descenso recursivo. Watt [11] describe un analizador sintáctico de descenso recursivo para una gramática G como un conjunto de métodos *reconoceN*, uno por cada símbolo no terminal N de G. La tarea de cada uno de estos métodos es reconocer únicamente frases N. En conjunto, reconocen el lenguaje completo.

La sintaxis de nuestro lenguaje viene especificada mediante una gramática libre de contexto en ECMA-262; a partir de dicha gramática, los programadores de Rhino construyeron un analizador sintáctico escrito en Java, se tradujo dicho analizador a C# y se construyó una representación intermedia distinta para nuestros propósitos.

Debido a que la gramática de ECMAScript contiene reglas que hacen imposible decidir cuál regla aplicar en el momento de continuar el análisis sintáctico únicamente identificando el siguiente símbolo no terminal, es necesario tener un mecanismo que nos permita ver uno o más elementos que siguen al elemento actual en el flujo de átomos y de ser necesario tener la posibilidad de regresar y restablecer los átomos leídos como si no se hubiese realizado dicha operación; este mecanismo es conocido como *lookahead*. Un caso ejemplo de esto sería la posibilidad de encontrarnos con una declaración de función y un enunciado de expresión (que se puede reducir a una expresión de función), en cuyo

caso estaríamos frente a la posibilidad de construir dos árboles de sintaxis abstracta distintos para la misma construcción gramatical, lo cual implicaría tener problemas de ambigüedad. Para ello es necesario verificar que el enunciado de expresión no inicie con { o la palabra reservada function, para decidir aplicar la regla de declaración de función de manera no ambigua. El algoritmo se presenta en el listado 5.2.1.

El algoritmo de reconocimiento sintáctico funciona de la siguiente manera:

- Nos mantenemos en un ciclo hasta que nos encontremos con el fin de archivo.
- Leemos un átomo; si es el átomo de inicio de una función intentamos reconocer una función[1] y, en caso contrario devolvemos el átomo e intentamos reconocer algún enunciado válido; en ambos casos—si no hubo algún error de sintaxis—devolvemos la representación de la unidad sintáctica reconocida y la agregamos a un bloque que representará al programa completo, y continuamos la iteración.

Es importante mencionar cuales métodos son aquellos que consumen la mayor parte del tiempo de cómputo: GetToken, Function y Statement.

- GetToken se encarga de obtener el átomo que continúa, pertenece a la clase TokenStream y trabaja de la siguiente manera:
  - Lee de manera secuencial los carácteres de un archivo hasta que alguna de sus reglas de construcción de átomos esté completa.

  Básicamente es un autómata finito con Lookahead de un átomo, en el cual están codificadas las reglas para construir los átomos para identificadores, palabras reservadas, literales numéricas, cadenas y expresiones regulares, además de eliminar comentarios, espacios en blanco e implementar la semántica de la inserción automática de finalizador de enunciado. La complejidad en el peor caso es $O(n)$ donde n es la longitud del átomo que se reconoce.

- Function reconoce y construye la representación de una declaración de función, pertenece a la clase Parser y trabaja así:
  - Discernimos entre el ser una función anónima o no, invocando al método MatchToken con Token.NAME como parámetro. MatchToken regresa verdadero o falso dependiendo de si el número del átomo que recibe como parámetro es igual al que regresa una invocación a GetToken. Su complejidad es la misma que la de GetToken.
  - Aseguramos que los elementos obligatorios estén en su lugar; en caso contrario reportamos un error. MustMatchToken se encarga de eso, recibe el número del átomo obligatorio, obtiene el siguiente átomo con GetToken, los compara y reporta el error en caso de no ser existosa. Su complejidad es la misma que la de GetToken.

[1] Que puede ser una expresión de función o una declaración de función

Listado 5.2.1 Método Parse de la clase Parser.

```
AST Parse ()
{
  current_script_or_fn = new ScriptBlock ();
  ok = true;
  try {
    while (true) {
      ts.allow_reg_exp = true;
      int tt = ts.GetToken ();
      ts.allow_reg_exp = false;
      if (tt <= Token.EOF)
        break;
      AST n;
      if (tt == Token.FUNCTION) {
        try {
          n = Function (current_script_or_fn,
                     FunctionType.Statement);
        } catch (ParserException e) {
          ok = false;
          break;
        }
      } else {
        ts.UnGetToken (tt);
        n = Statement (current_script_or_fn);
      }
      current_script_or_fn.Add (n);
    }
  } catch (StackOverflowException ex) {
      throw new Exception (
                "Error: too deep parser recursion.");
  }
  if (!ok)
    return null;
  return current_script_or_fn;
}
```

- Obtenemos todos los parámetros de la función, lo cual tiene complejidad $O(n)$ donde n es el número de parámetros formales.
- Reconocemos al cuerpo de la función con ParseFunctionBody. Esta función es idéntica a Parse, excepto por los siguientes cambios:
  - No obtenemos el siguiente átomo sino que simplemente lo leemos; esto tiene como consecuencia que no tenemos que regresarlo al flujo de átomos en caso de que no vayamos a reconocer a otra función anidada.
  - El ciclo se detiene cuando encontramos una llave que cierra, es decir, reconocemos a todos los elementos que se encuentran dentro del cuerpo de la función.

  La entrada para este algoritmo es un programa que tiene n enunciados a nivel global y su complejidad en el peor caso depende de manera lineal de n ya que, dependiendo del tipo de unidad sintáctica, podemos tener que para reconocer alguna de ellas sea trivial y sea $O(1)$ (como podría ser reconocer una literal numérica) pero también puede ocurrir que la unidad sintáctica a reconocer posee en su estructura un bloque de enunciados, el cual depende a su vez de manera lineal del número de enunciados en su bloque, lo que nos genera una estructura que inicia como un arreglo de longitud n en el cual están contenidos los enunciados globales iniciales, pero de cada unos de ellos puede colgar otro arreglo con k_i enunciados y así recursivamente; esto indica que el algoritmo que reconoce esa estructura depende de n y además debe considerar el tiempo que le toma reconocer las unidades anidadas; sin embargo, el tiempo que le toma reconocer cada una de las k_i unidades se expresa como la suma de la complejidad de cada uno de sus elementos y como el peor caso de cada elemento es lineal el reconocimiento de cualquiera es lineal. De donde Function invierte Kn en procesar todos los elementos - con K $Maximo(TiempoDeProcesamiento(k_i))$ - de donde es $O(n)$.

- Statement reconoce y construye la representación de cualquiera de los enunciados válidos pertenecientes al lenguaje. Opera de manera similar a Function: obtiene el átomo que continúa y realiza una comparación exahustiva de las posibles unidades sintácticas que podría estar iniciando, dependiendo de la unidad sintáctica que cumple, se intenta reconocer a profundidad todos los elementos que la conforman. Su peor caso es similar al de Function, que es aquel en que la unidad sintáctica que está siendo reconocida contiene un bloque y tiene que reconocerse de manera recursiva.

Procesamos uno por uno cada elemento global a profundidad, con lo cual tenemos Kn tiempo de procesamiento (con K siendo el máximo de los tiempos invertidos en procesar a los elementos k_i), por lo cual, Parse es $O(n)$.

5.2.1. Representaciones intermedias

De acuerdo con Muchnick [9] podemos clasificar en tres grupos a las representaciones intermedias: de alto, medio y bajo nivel. La representación de alto nivel se usa en las etapas iniciales del proceso de compilación y luego se transforma en formas de más bajo nivel. El ejemplo mas común es el del árbol de sintaxis abstracta, el cual hace explícita la estructura del programa y solamente posee la información necesaria para reconstruir el código fuente a partir de ella. La representación de nivel medio está diseñada para reflejar el rango de características en un conjunto de lenguajes de programación, pero de manera independiente del lenguaje y además están diseñados de tal manera que sirva de base para la generación eficiente de código para una o más arquitecturas. Las representaciones de bajo nivel corresponden generalmente a un mapeo uno a uno con algún conjunto de instrucciones de máquina y por lo tanto son dependientes de la arquitectura.

En nuestro caso utilizamos una mezcla de árbol de sintaxis abstracta que posteriormente, durante el proceso de análisis semántico, reúne la información necesaria para generar el código adecuado y hacer el ligado de las variables con su declaración, entre otras operaciones. En esta versión del compilador no se realizan optimizaciones del estilo de reemplazo de constantes, eliminación de código muerto, expansión de ciclos, por eso es que nos es suficiente esta representación por el momento; en un futuro para poder implementar al menos las optimizaciones antes mencionadas será necesario cambiar la representación intermedia; cabe mencionar que no hay resultados que muestren qué tipo de optimizaciones conviene hacer a nivel de CIL o si es mejor implementarlas en el jitter.

La representación que se utilizó consiste de una jerarquía con la clase abstracta AST como raíz. Ésta indica que toda clase que derive de ella tiene que implementar dos métodos:

- Resolve que realiza el análisis semántico sobre el enunciado actual.
- Emit que agrega el código CIL adecuado para el enunciado en turno al flujo de código que será el programa ejecutable.

Algunos ejemplos de las clases que derivan de la clase raíz AST son ArrayLiteral, Binary, Block, Break, Call, Continue, Eval, ForIn, If, Relational, ScriptBlock, Throw, Try, VariableDeclaration, With, entre otras. Todas ellas representan a una unidad sintáctica o semántica específica que permite encapsular de manera adecuada las operaciones que se tienen que realizar en cada caso.

La representación intermedia se construye al mismo tiempo que el analizador sintáctico realiza el proceso de validación, de tal manera que una vez que ese proceso finaliza de forma exitosa tenemos como resultado un árbol que representa el programa con el que iniciamos y es ésa la estructura de datos sobre la que se realiza el análisis semántico y la generación de código.

5.3. Algoritmos para análisis semántico

En forma general se dice que esta fase del proceso de compilación se caracteriza por asociar las presencias de variables con sus declaraciones, realizar la verificación de tipos y la transformación de la representación intermedia. Wilhelm [12] señala que un número considerable de las propiedades necesarias de programas no se pueden describir con una gramática libre de contexto, sino que tienen que describirse a través de predicados, que son llamados *condiciones de contexto*. Estas incluyen la propiedades relacionadas con las declaraciones y la consistencia de tipos. Ambas dependen de las reglas de alcance y visibilidad del lenguaje de programación. En concreto, la tarea del análisis semántico es verificar que se cumplan dichas *condiciones de contexto* y se realiza en dos fases: la primera realiza la identificación de símbolos y además verifica que se cumplan las propiedades relacionadas con las declaraciones, lo cual involucra las reglas de alcance y visibilidad del lenguaje de programación; la segunda realiza la verificación de tipos.

5.3.1. Identificación de símbolos

La identificación de símbolos se lleva a cabo mediante el uso de una tabla de símbolos. Una propiedad interesante de ECMAScript es que permite un número ilimitado de declaraciones del mismo identificador y al encontrarse con una ocurrencia de un símbolo en un contexto como expresión se toma el valor de la última declaración hasta ese punto; esto es así para variables así como para funciones. La tabla de símbolos que se usó está basada en la que Appel [4] propone; sin embargo se agregaron verificaciones y funciones para implementar la semántica específica de ECMAScript. La tabla de símbolos tiene una estructura de bloques análoga a la del programa, pero no es el producto del proceso de identificación sino que es usada para llevarlo acabo. De acuerdo con Wilhelm [12], el resultado del proceso de identificación es que para todo nodo de una ocurrencia no declaratoria de un identificador, se guarda alguna de las siguientes opciones:

- Un apuntador al nodo de la declaración,
- La dirección de la entrada en la tabla de símbolos de la ocurrencia de definición, o
- La información declarativa de la presencia de definición

Se optó por la opción (1) ya que en la tabla de símbolos guardamos el símbolo y le asociamos el apuntador a la declaración que lo define; esto nos permite compartir la declaración entre todos las presencias no declarativas que son iguales en el árbol.

El analizador sintáctico nos devuelve la representación intermedia del programa y el analizador semántico realiza su trabajo sobre esa estructura de datos, que en realidad es un ScriptBlock, que a su vez depende de un Block. El método Resolve de la clase Block es el que se muestra en el listado 5.3.1.

Listado 5.3.1 Método Resolve de la clase Block.

```
internal override bool Resolve (IdentificationTable context)
{
  AST e;
  bool no_effect;
  bool r = true;
  int i, n = elems.Count;
  if (parent == null || parent is FunctionDeclaration)
    no_effect = true;
  else
    no_effect = false;
  for (i = 0; i < n; i++) {
    e = (AST) elems [i];
    if (e is VariableStatement)
      (e as VariableStatement).PopulateContext (context);
    else if (e is FunctionDeclaration) {
      string name = ((FunctionDeclaration) e).func_obj.name;
      AST binding = (AST) context.Get (
                              Symbol.CreateSymbol (name));
      if (binding == null) {
        ocurrences.Add (name, i);
        context.Enter (
               Symbol.CreateSymbol (
                    ((FunctionDeclaration) e).func_obj.name),
                    new FunctionDeclaration ());
      } else {
        Console.WriteLine (
          "warning: JS1111: '{0}' has already been defined.",
          name);
        if (!(binding is FunctionDeclaration))
          throw new Exception (
          "error JS5040: '" +
          ((VariableDeclaration) binding).id +
          "' it's read only.");
        int k = (int) ocurrences [name];
        elems.RemoveAt (k);
        if (k < i)
          i -= 1;
        ocurrences [name] = i;
      }
    }
  }
  n = elems.Count;
  for (i = 0; i < n; i++) {
    e = (AST) elems [i];
    if (e is Exp)
      r &= ((Exp) e).Resolve (context, no_effect);
    else
      r &= e.Resolve (context);
  }
  return r;
}
```

Antes de explicar la idea de este algoritmo quiero mencionar dos características de ECMAScript:

1. Permite que se usen símbolos que aún no han sido declarados.
2. Permite tener expresiones a nivel global.
3. Permite la declaración de múltiples variables o funciones con el mismo nombre.

La primera de ellas quiere decir que necesariamente tenemos que visitar cuando menos en dos ocasiones el árbol durante la fase de análisis semántico, porque si no estaríamos ante la posibilidad de detener el proceso indicando un error por la aparición de una referencia a un símbolo que es declarado posteriormente, lo que sería erróneo hacer. Por ejemplo, el programa del listado 5.3.2 sería clasificado como inválido y estaríamos violando la regla semántica antes mencionada, sin embargo, el programa se compila sin errores y su resultado en tiempo de ejecución es 7.

Listado 5.3.2 Uso de variables aún no declaradas en ECMAScript.

```
x = 3;
y = 4;
print (x + y);
var x, y;
```

La segunda cláusula nos dice que podemos tener cualquier expresión que se evalúe a nivel global y después el posible valor que regresa sea descartado. Es sumamente importante mantener la integridad de la pila de evaluación en el CLI y una de las reglas más importantes es la de terminar con la pila de evaluación vacía cuando intentamos regresar de una función; es por ello que tenemos que poder discernir en que momento se tiene que realizar un operación OpCodes.Pop para poder regresar de las funciones.

Para la tercer cláusula es importante mencionar la semántica de cada uno de los casos:

- Cuando en un programa se tienen dos o más declaraciones de variable con el mismo identificador, lo que ocurre es que se crea un solo campo; en caso de tener iniciadores se evalúan en el orden en que aparecen; eso permite que la variable vaya tomando distintos valores conforme avanza el flujo del programa. Por ejemplo, si tuviéramos el programa que se muestra en el listado 5.3.3; en el momento de ejecutarlo, obtenemos lo que se muestra en el listado 5.3.4 que implementa la semántica adecuada.

Listado 5.3.3 Declaración múltiple de una misma variable.

```
var x = 1;
print ("x␣=␣", x);

x = 1000;
print (‘‘x = ‘‘, x);

var x = 2;
print (‘‘x = ‘‘, x);
```

Listado 5.3.4 Resultado de la declaración múltiple de una misma variable.

```
x = 1
x = 1000
x = 2
```

El método PopulateContext de la clase VariableStatement durante el proceso de análisis semántico verifica que en el bloque actual no se agregan declaraciones de variable repetidas, pero sólo tiene que ser en el bloque actual ya que también debemos permitir poder definir variables con el mismo nombre que estén en bloques distintos. Esa verificación es $O(1)$ en el caso promedio u $O(n)$ en el peor caso, ya que se realiza la búsqueda sobre la tabla de dispersión del bloque actual.

- Caso contrario es el de las declaraciones múltiples de función con el mismo identificador; en este caso debemos descartar todas las declaraciones menos la última, a diferencia de las declaraciones de variables donde tenemos que ligar con la declaración inmediata anterior más cercana. Si tuviéramos el programa como el del listado 5.3.5, el resultado es desplegar una advertencia la cual expresa que se está redefiniendo la función y obtenemos como resultado al evaluarlo lo que se muestra en el listado 5.3.6.

Listado 5.3.5 Declaración múltiple de la misma función.

```
function f ()
{
  print (‘‘primera f’’);
}

f ();

function f ()
{
  print (‘‘segunda f’’);
}

f ();
```

Listado 5.3.6 Resultado de declaración múltiple de una misma función.

```
segunda f
segunda f
```

Con esto en mente analicemos el método Resolve de la clase Block. La idea principal de este algoritmo es poblar la tabla de símbolos con los identificadores de las declaraciones de variables y funciones, manejar de manera adecuada las declaraciones repetidas y manejar de manera adecuada las expresiones globales. Inicialmente determinamos si nos encontramos con una expresión global, después recorremos cada una de las unidades sintácticas que conforman al bloque y tratamos de manera especial a las declaraciones. Si estamos frente a un enunciado de declaración de variables invocamos a PopulateContext de la clase VariableStatement que inserta en la tabla el identificador de la declaración, a menos que exista una declaración previa con el mismo identificador en el bloque de alcance actual; InCurrentScope de la clase IdentificationTable realiza la verificación. Si nos encontramos con una declaración de función se realiza lo siguiente:

1. Buscamos en el bloque de alcance actual si existe una declaración previa. Dicha búsqueda se realiza sobre una tabla de dispersión simple, lo cual nos brinda una complejidad de $O(n)$, con n el máximo número de elementos en la tabla en el peor caso, pero de $O(1)$ en el caso promedio.

2. La semántica de las declaraciones repetidas de función indica que toda presencia de invocación de una función toma la última de las declaraciones en caso de haber declaraciones múltiples de función con el mismo

identificador. Si la función no está previamente definida se guarda el índice en el que apareció y se agrega su símbolo a la tabla; en caso contrario lanzamos la advertencia; si la declaración repetida no es declaración de función se lanza un error, mientras que en el otro caso obtenemos el índice de la presencia anterior, eliminamos el elemento en ese índice y guardamos la nueva posición de la última declaración. Esto nos permite ir eliminando las declaraciones que nunca serán usadas y poder mantener la posición de la última declaración que es a la cual se ligarán todas las presencias de invocación de ese símbolo en el bloque. Encontrar el índice de la última presencia también se realiza sobre una tabla simple de dispersión.

3. Finalmente se verifica si la expresión es global en cuyo caso necesitamos generar una instrucción de OpCodes.Pop para vaciar la pila de ejecución. Esta verificación es de orden constante ya que simplemente verificamos que la unidad sintáctica derive de la clase abstracta Exp y se le pasa como argumento un valor booleano que indica si su valor tendrá efecto en un momento posterior; dicho valor se obtiene a través del padre de la unidad sintáctica en cuestión.

Con todas las operaciones desglosadas podemos observar que tenemos una suma de términos de orden lineal ($\Sigma_{i=0}^{n} k_i$) donde la complejidad de cada una de las k_i es $O(n)$, por lo que la complejidad de Resolve de la clase Block es $O(n)$, ya que resulta ser la suma de complejidades de $O(n)$.

5.3.2. Análisis semántico del ligado lo más tarde posible

En la sección 3.2.3 ya se explicó a qué nos referimos con este término y en 4.2 se mencionó cómo se realiza la evaluación de este tipo de expresiones. En esta sección explicaré cómo se decide cuál expresión necesita de este tipo de evaluación. Debido a que las expresiones que involucran ligado lo más tarde posible son aquéllas donde se quiere realizar una invocación, acceso a propiedad y aplicación de método, todas esas construcciones del lenguaje se mapean a una expresión binaria donde el método Resolve de la clase Binary es donde se realiza todo el proceso que determina cuándo se necesita la evaluación lo más tarde posible. Este método se presenta en el listado 5.3.7.

La manera en que se resuelven los accesos a propiedades es la siguiente:

1. Se tiene una interfaz IAccesible, la cual obliga a sus implementadores a tener una definición del método ResolveFieldAccess.

2. Dicho método hace uso de los servicios de introspección de Mono para detectar cuándo podemos evitar la evaluación lo más tarde posible.

La necesidad de poder evitar la evaluación lo más tarde posible tiene dos razones: la primera es por eficiencia ya que el acceso de manera directa es menos costoso y la segunda es que se deben poder realizar dichas operaciones de manera directa en los casos en que se use un método estático de alguno de los objetos del

Listado 5.3.7 Método Resolve de la clase Binary.

```
internal override bool Resolve (IdentificationTable ctx)
{
  bool r = true;

  if (left != null)
    r &= left.Resolve (ctx);

  if (right != null)
    if (op == JSToken.AccessField &&
        right is IAccesible) {
      r &= ((IAccesible) right).ResolveFieldAccess (left);
      if (!r)
        late_bind = true;
    } else
        r &= right.Resolve (ctx);
  return r;
}
```

sistema de objetos de ECMAScript, que además en ocasiones poseen una firma fuertemente tipificada; como consecuencia se tiene que solamente cuando se usa el operador *punto* o de acceso a propiedad (sea campo o método) es cuando puede optimizarse, lo cual nos lleva a enfocarnos en el método ResolveFieldAccess de la clase Identifier, que se presenta en el listado 5.3.8. De las cuestiones importantes

Listado 5.3.8 Método ResolveFieldAccess de la clase Binary.

```
public bool ResolveFieldAccess (AST parent)
{
  if (parent is Identifier) {
    Identifier p = parent as Identifier;
    return is_static_property (p.name.Value, name.Value);
  }
  //
  // Return false so late binding will take place
  //
  return false;
}
```

que requieren mención es que ResolveFieldAccess tiene como valor de regreso verdadero si y sólo si el objeto objetivo y la propiedad que está siendo accedida pertenece al sistema de objetos de ECMAScript y es un método estático de dicho objeto, en cuyo caso no necesitamos de evaluación lo más tarde posible.

El método is_static_property maneja este tipo de casos, como se muestra en el listado 5.3.9

Listado 5.3.9 Método is_static_property de la clase Identifier.

```
//
// If obj_name is a native object, search in its
// constructor for prop_name, otherwise return
// false letting late binding taking place
//
bool is_static_property (string objName, string propName)
{
  bool native_obj =SemanticAnalyser.is_js_object(objName);
  if (!native_obj)
    return false;

  bool contained;
  contained = SemanticAnalyser.object_contains (
  SemanticAnalyser.map_to_ctr (objName), propName);
  if (!contained)
    throw new Exception (
      "error JS0438: No support of this property");
  return true;
}
```

En concordancia con el paradigma orientado a objetos basado en prototipos de ECMAScript, en esta implementación los constructores de los objetos son quienes poseen los miembros que serán compartidos, en particular los métodos estáticos; es por ello que realizamos el mapeo (map_to_ctr) del nombre al constructor adecuado. Los únicos constructores que poseen miembros estáticos son: Date, Math, Number y String; sobre el constructor adecuado realizamos la búsqueda del miembro al que se desea acceder y es aquí donde usamos instrospección, SemanticAnalyser.contains; ese método recibe el tipo objetivo y el nombre del miembro y se encarga de comprobar que dicho miembro pertenece o no al objeto.

La complejidad en el peor de los casos de Binary.Resolve depende de:

- La suma entre el tiempo que se tarde en resolver left y el tiempo que se tarde en resolver rigth (que se puede resolver de la manera normal o mediante ResolveFieldAccess).
- La complejidad de ResolveFieldAccess depende de la complejidad del método is_static_property que a su vez depende de la suma de la complejidad entre is_js_object y object_contains.
  - is_js_object invoca a contains con el objeto global como parámetro y a su vez contains invoca a Type.GetMembers e itera sobre el arreglo

que devuelve para saber si la propiedad con el nombre especificado sí se encuentra.

- object_contains también invoca a contains pero con el constructor del objeto adecuado como parametro.
    - Type.GetMembers es un filtro que obtiene un arreglo con los elementos que cumplen con las características especificadas en sus parámetros, en cuyo caso tiene que revisar todos y cada uno de los miembros para poder encontrar cuáles sí cumplen las condiciones; es $O(n)$ con respecto al número de miembros que contenga el tipo especificado.

Al final todo se reduce a la suma de términos lineales, de donde Binary.Resolve es $O(n)$.

5.4. Algoritmos para generación de código

Esta fase de la compilación es la encargada de transformar los programas escritos en cierto lenguaje de programación a código objeto, por lo que es dependiente del lenguaje y la máquina objetivo en la cual se ejecutará (ya sea abstracta o real). Es por esto que es más difícil poder sacar conclusiones generales que apliquen a cualquier lenguaje y máquina en cuestión. En general se dice que hay tres subproblemas en la generación de código:

1. **Selección de código**. Este problema consiste en decidir qué sucesión de instrucciones será el código objeto de las unidades sintácticas del lenguaje de programación que se está compilando.

2. **Alojamiento de almacenamiento**. Este problema consiste en decidir cuál es la dirección de almacenamiento de cada variable en el programa fuente.

3. **Alojamiento de registros**. Si la máquina objetivo tiene registros, es muy importante usar dichos registros para almacenar resultados intermedios durante la evaluación de expresiones; sin embargo, se presentan muchas complicaciones al tratar de resolver este problema.

En nuestro caso, dichos puntos se atacaron de la siguiente manera:

1. La selección de código se generó inductivamente sobre las unidades sintácticas de ECMAScript; esto nos dió un algoritmo muy similar al patrón *visitador* en el cual se recorre el árbol de sintaxis abstracta anotado, generando el código de cada construcción de manera recursiva con cada unidad sintáctica encargada de traducirse de la manera adecuada.

2. Como la máquina objetivo es el CLI tenemos dos ventajas: el alojamiento de las direcciones de almacenamiento las maneja el generador de código por

omisión (ILGenerator); como el modelo de ejecución del CLI es una máquina con pila de evaluación no tenemos que preocuparnos por el alojamiento de registros de manera eficiente ya que el JITter de Mono se encarga de ello en una etapa posterior, en la etapa de ejecución del programa objeto generado.

Por tal razón toda esta sección está dedicada a mostrar algunos de los algoritmos para realizar la traducción a código CIL. Un método común y que Watt [11] explica, consiste en especificar la selección de código inductivamente sobre las unidades sintácticas del lenguaje usando *funciones de código* y *plantillas de código*. El código objeto de cada unidad sintáctica es una sucesión de instrucciones a las cuales se traducirá. El código objeto está en el lenguaje CIL^*. Para cada unidad sintáctica de clase P en la sintaxis abstracta del lenguaje se introduce una función f_p que traduce cada frase en la clase P a código objeto:

$$f_p : P \rightarrow CIL^*.$$

Se define la función de código f_p a través de un número de plantillas de código, con al menos una plantilla por cada forma de unidad sintáctica en la clase P. Si alguna de las formas dentro de clase P contiene subunidades Q y R entonces su plantilla de código se define en términos de las subunidades.

$$f_p[\ldots Q \ldots R \ldots] =$$

$$\vdots$$

$$f_Q Q$$

$$\vdots$$

$$f_R R$$

donde f_Q y f_R son funciones de código apropiadas para las subunidades Q y R.

5.4.1. Generación de código para declaración de funciones

El algoritmo que realiza este mapeo se encuentra en la función create_closure, la cual es invocada en el método Emit de la clase Block, antes de que el cuerpo de la función se genere, con lo que garantizamos que cualquier referencia recursiva será válida, al igual que cualquier referencia a ella como variable libre dentro del cuerpo de otra función donde sea válido ligar dicha variable con la declaración de función involucrada.

1. **Construimos el nombre de la función**. Si es una función no anidada su nombre es el de la declaración, en el otro caso su nombre está formado por la concatenación de los nombres de las funciones que la contienen separados por puntos.

2. Obtenemos un constructor de método.

Listado 5.4.1 Método create_closure de la clase FunctionDeclaration.

```
internal void create_closure (EmitContext ec)
{
  string name = func_obj.name;
  string full_name;
  TypeBuilder type = ec.type_builder;
  ILGenerator ig = ec.ig;

  if (prefix == String.Empty)
    full_name = name;
  else
    full_name = prefix + "." + name;

  MethodBuilder method_builder = type.DefineMethod (
                                 full_name,
                                 func_obj.attr,
                                 HandleReturnType,
                                 func_obj.params_types ());
  MethodBuilder tmp =(MethodBuilder)TypeManager.Get(name);

  if (tmp == null)
    TypeManager.Add (name, method_builder);
  else
    TypeManager.Set (name, method_builder);

  set_custom_attr (method_builder);
  this.ig = method_builder.GetILGenerator ();

  if (parent == null ||
      parent.GetType () == typeof (ScriptBlock))
    type.DefineField (name, typeof (ScriptFunction),
                      FieldAttributes.Public |
                      FieldAttributes.Static);
  else {
    local_func = ig.DeclareLocal (
                      typeof (ScriptFunction));
    TypeManager.Add (name, local_func);
  }
  build_closure (ec, full_name);
}
```

3. Si la función ya está declarada nada más asociamos el nuevo constructor de método con el identificador que ya se encuentra en el TypeManager.

4. Si es una función no anidada creamos un campo estático de tipo ScriptFunction; de otro modo creamos una variable local del mismo tipo; en ambos casos se agregan al TypeManager.

5. Construimos la cerradura (*closure*) de la función en el contexto de generación de código actual.

Este algoritmo recibe como entrada un nodo del árbol de sintaxis abstracta que representa un declaración de función y obtenemos como resultado todo el código CIL que nos permitirá invocar dicha función.

Durante el paso 1, el único trabajo a realizar consiste en concatenar el prefijo de la función (el cual fue construido al mismo tiempo que se construyó nuestro árbol de sintaxis abstracta) con el nombre local, en caso de que nuestra función sea anidada; la complejidad de realizar la concatenación de dos cadenas en el peor de los casos es de orden $O(k + l)$, donde k es la longitud del prefijo y l la longitud del nombre actual.

En el paso 2 obtenemos un constructor de método[2]. Para esto, el constructor de tipo[3] en el peor de los casos tiene que crear un nuevo arreglo de longitud el doble del número de métodos máximo que puede almacenar actualmente el constructor de tipo, copiar el contenido del arreglo anterior al nuevo y agregar el nuevo método. De donde la complejidad de esto es de $O(m)$ con m el número máximo de métodos que se pueden almacenar actualmente.

En el paso 3 formamos el nuevo contexto de generación de código; esto nos toma $O(1)$, ya que simplemente construimos el nuevo contexto con el constructor de tipo, el constructor de módulo y el generador de código CIL como parámetros.

En el paso 4 construimos un campo estático o una variable local según fuera el caso necesario para esto en el caso de la definición del campo estático; de igual manera que en el caso del constructor de método, el peor caso se da cuando necesitamos crear un nuevo arreglo (que contiene a los campos definidos) de longitud doble del actual, se copian los campos anteriores y agregamos el campo estático que estamos definiendo; esto es de $O(n)$ donde n es el número máximo de campos estáticos actual. Lo mismo para la definición de la variable local, que es de $O(p)$ donde p es el número máximo de variables locales actual.

En el paso 5 construimos la cerradura de la función, lo que involucra varias llamadas a subrutinas auxiliares. La primera de ellas consiste en generar el código correspondiente a los parámetros que recibe la función. Básicamente lo que hacemos es guardar los nombres de cada uno de los parámetros dentro de un arreglo; esto es de orden $O(q)$ donde q es el número de parámetros que recibe nuestra función. Luego generamos el código para las variables locales declaradas dentro del cuerpo de la función. Lo que hacemos es generar código que construye campos locales que en tiempo de ejecución pueden ser accedidos y hacemos esto

[2]Del término en inglés MethodBuilder.

[3]Del término en inglés TypeBuilder.

para cada una de las variables locales declaradas en la función, esta subrutina es de $O(r)$ con r el número de variables locales declaradas en la función. Al final asignamos la cerradura al campo estático creado en el paso 4.

En el paso 6 generamos el código de cada uno de los enunciados que pertenecen al cuerpo de la función, exceptuando las declaraciones de variables o funciones.

5.4.2. Generación de código para expresión de funciones

Una expresión de función es sintácticamente igual a una declaración de función, exceptuando el caso donde una función de expresión es anónima. Semánticamente difieren aún más, ya que una declaración de función solamente puede aparecer en un contexto global, ya sea dentro o fuera de otra declaración de función o expresión de función, mientras que una expresión de función puede aparecer del lado derecho de una asignación – lo cual nos permite asignar funciones a variables o campos de objetos – o como una aplicación de función. El caso particular en que la función es anónima es especial ya que aunque en el programa fuente esa función no tiene un idenficador con el cual nos podemos referir a ella, internamente debemos ser capaces de identificarla; para ello el analizador semántico se encarga de llevar un registro de todas la funciones anónimas que aparecen en el programa y les asigna un nombre único formado por el prefijo “anonymous ” y un número de serie que nunca se repite. Todo el proceso de construcción del prefijo en el caso de tratarse de casos anidados y de la construcción de la cerradura se realiza de manera similar a como se hace en la declaración de funciones.

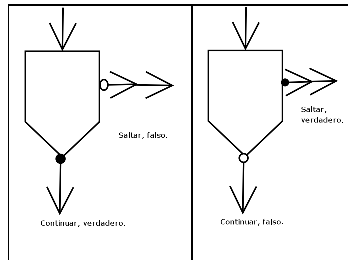
5.4.3. Generación de código para expresiones booleanas

Las expresiones booleanas ocurren en dos contextos distintos en los lenguajes de programación. El caso más común es aquel en el cual las expresiones son evaluadas para afectar el control del flujo. Todas las expresiones booleanas que se presentan en los enunciados condicionales pertenecen a esta categoría. La otra posibilidad es aquella en la que se evalúa una expresión para asignarla o pasarla como parámetro a un subprograma.

Una de las maneras de expresar la evaluación de dichas expresiones booleanas es a través de la evaluación en corto circuito. En nuestro caso, ECMA-262 especifica en la sección 11.11, de manera precisa que la manera de evaluación de los operadores binarios lógicos en ECMAScript se realiza de esta manera. Esta forma de evaluación se implementa usando código de salto. El código de salto evalúa las expresiones incrementalmente y termina la ejecución tan pronto como el resultado es conocido.

Se produce código de salto mediante un recorrido recursivo del árbol de la expresión en la sintaxis abstracta. Hay dos operaciones fundamentales que se deben brindar:

- Producir código que evalúe expresiones booleanas y que condicionalmente



Cuadro 5.1: Primitivas de control de flujo, fall_true y fall_false.

pase el control a una de dos posibles etiquetas.

- Evaluación de expresiones booleanas con la semántica correcta, donde el resultado es un valor booleano en el tope de la pila de evaluación.

Para lograr esto, usamos los métodos recursivos fall_true y fall_false de la clase CodeGenerator, que nos permiten realizar las operaciones fundamentales antes mencionadas.

Existen dos circunstancias distintas en las cuales queremos generar saltos condicionales. La primera es aquella en la cual queremos que el control de flujo brinque hacia una etiqueta conocida si alguna expresión booleana se evalúa a verdadero. Un ejemplo común de esta situación es cuando queremos generar código para un ciclo cuya expresión condicional se evalúa posteriormente a la evaluación del cuerpo del ciclo. Por lo tanto, queremos generar código que salta al inicio del ciclo si la expresión se evalúa a verdadero. Si la expresión se evalúa a falso, queremos que el flujo de ejecución pase al siguiente enunciado del programa.

De igual manera tenemos situaciones donde queremos que el control de flujo brinque hacia alguna etiqueta conocida si la expresión booleana se evalúa a falso. Un ejemplo conocido es el de los ciclos con expresión condicional *a posteriori* pero donde se continúa en el ciclo hasta que la condición se cumpla (*repeat ... until (b)*). En estos casos queremos que el control de flujo continúe si la expresión se evalúa a verdadero, pero que brinque hacia el inicio del ciclo si la expresión es falsa.

De acuerdo con Gough [8] todos los saltos condicionales se pueden expresar como instancias de los dos casos anteriores y cuya representación gráfica se presenta en el cuadro 5.1.

Nuestros dos métodos reciben como parámetros una expresión a evaluarse y una etiqueta a la cual saltar; en el caso de fall_true si la expresión es falsa y en el de fall_false si la expresión se evalúa a verdadero. Estos dos métodos se invocan recursivamente uno al otro en el curso de la evaluación de una expresión booleana.

En casi todos los lenguajes existen construcciones sintácticas explícitas para denotar una expresión condicional tal como:

$$exp_1 \ ? \ exp_2 \ : \ exp_3$$

En esta expresión deseamos que el control *caiga* a la evaluación de exp_2 en el caso de que exp_1 sea verdadero y que *brinque* a la evaluación de exp_3 en el caso falso. Entonces, lo que tenemos son dos etiquetas, una a la cual saltaremos si la expresión se evalúa a falso y la otra es el punto de unión al final de la expresión. La implementación de dicha expresión está dada en la clase Conditional y el método Emit de ella es el que se presenta en el listado 5.4.2.

Listado 5.4.2 Método Emit de la clase Conditional.

```
internal override void Emit (EmitContext ec)
{
  ILGenerator ig = ec.ig;

  Label false_label = ig.DefineLabel ();
  Label merge_label = ig.DefineLabel ();

  CodeGenerator.fall_true (ec, cond_exp, false_label);

  if (true_exp != null)
    true_exp.Emit (ec);

  ig.Emit (OpCodes.Br, merge_label);
  ig.MarkLabel (false_label);

  if (false_exp != null)
    false_exp.Emit (ec);

  ig.MarkLabel (merge_label);

  if (no_effect)
    ig.Emit (OpCodes.Pop);
}
```

Lo que se hace es definir dos etiquetas, en una iniciará la evaluación de exp_3 y la otra será el punto de unión donde termina la evaluación de la expresión completa; después invocamos el método fall_true con exp_1 y la etiqueta que apuntará al inicio de exp_3 como parámetros.

fall_true, que se muestra en el listado 5.4.3, lo que hace es verificar qué tipo de expresión le fue pasada como parámetro; esto es necesario ya que en ECMAScript existe el operador comma[4], en cuya presencia se genera el código de evaluación de todas las expresiones involucradas y después es necesario checar el tipo de la última de las expresiones del encadanamiento; de otro modo analizamos el tipo del parámetro directamente. Tenemos un caso canónico y dos casos especiales;

[4]El operador comma es un operador para encadenar la evaluación de varias expresiones, un ejemplo sería $x + 1, w < z, true$.

Listado 5.4.3 Método fall_true de la clase CodeGenerator.

```
internal static void fall_true (EmitContext ec, AST ast,
                                Label lbl)
{
  Type type = ast.GetType ();

  if (type == typeof (Expression)) {
    Expression exp = ast as Expression;
    exp.Emit (ec);
    AST last_exp = (AST) exp.exprs [exp.exprs.Count - 1];

    if (last_exp is Binary)
      ft_binary_recursion (ec, last_exp, lbl);
    else if (last_exp is Equality)
      ft_emit_equality (ec, last_exp, lbl);
  } else if (type == typeof (Binary))
    ft_binary_recursion (ec, ast, lbl);
  else
    emit_default_case (ec, ast, OpCodes.Brfalse, lbl);
}
```

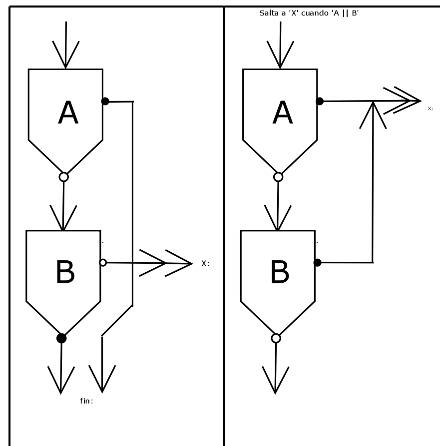
el caso canónico lo único que hace es transformar la expresión condicional a un valor booleano y se salta—haciendo una verificación de que la condición sea falsa—a la etiqueta deseada; los casos especiales son para los tipos Equality y Binary. Para el primer tipo se invoca el método ft_emit_equality en el cual si el operador es de igualdad saltamos a la etiqueta si la expresión condicional es falsa y si el operador es de desigualdad saltamos a la etiqueta si la condición es verdadera. Para el caso de una condición de tipo Binary se invoca el método ft_binary_recursion que separa los casos para los demás operadores lógicos (donde se implementa la evaluación en *corto circuito*) y de relación.

Listado 5.4.4 Método fall_false de la clase CodeGenerator.

```
internal static void fall_false (EmitContext ec, AST ast,
                                 Label lbl)
{
  Type type = ast.GetType ();

  if (type == typeof (Expression)) {
    Expression exp = ast as Expression;

    if (exp.Size > 1)
      exp.Emit (ec);

    AST last_exp = (AST) exp.exprs [exp.exprs.Count - 1];

    if (last_exp is Relational)
      ff_emit_relational (ec, last_exp, lbl);
    else if (last_exp is Binary)
      ff_binary_recursion (ec, last_exp, lbl);
    else if (last_exp is Identifier ||
             last_exp is BooleanLiteral)
      emit_default_case (ec, last_exp, OpCodes.Brtrue,
                         lbl);
    else if (last_exp is Equality)
      ff_emit_equality_cond (ec, last_exp, lbl);
    else
      throw new Exception ("uknown type: " +
                     last_exp.GetType ().ToString ());
  } else if (type == typeof (Binary))
      ff_binary_recursion (ec, ast, lbl);
    else
      emit_default_case (ec, ast, OpCodes.Brtrue, lbl);
}
```

El método fall_false, que se encuentra en el listado 5.4.4, hace de cierta manera lo mismo que fall_true pero de manera dual, salta verificando que la condición sea verdadera, y además maneja de manera específica los casos en que la condición o la última de las expresiones de la condición sean de tipo Relational, Identifier, BooleanLiteral para generar código adecuado para cada caso. En el caso de una condición de tipo relacional se utilizan las instrucciónes OpCodes.Blt, OpCodes.Bgt, OpCodes.Ble y OpCodes.Bge según sea el operador relacional en cuestión. En el caso de un identificador o una variable booleana, se genera la instrucción de conversión a tipo booleano sólo en caso de ser necesario y se salta a la etiqueta deseada.

De manera sumamente importante aparecen los patrones con que se imple-



Cuadro 5.2: Implementación de la disyunción mediante las primitivas fall_true y fall_false.

mentan los operadores de la disyunción y la conjunción. Las siguientes figuras muestran como estos dos métodos descomponen expresiones de la forma A *op* B cuando *op* es disyunción o conjunción lógica.

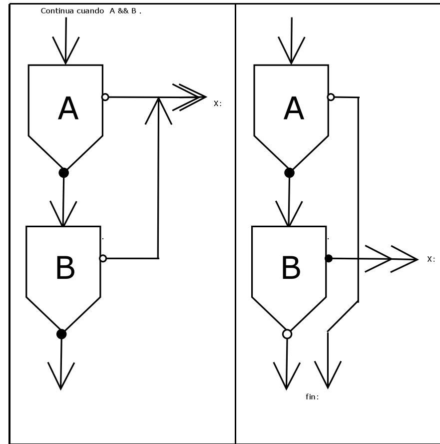
La implementación de estos operadores se encuentra contenida en los métodos ft_binary_recursion y ff_binary_recursion, en los cuales se separa cada uno de ellos y se realizan llamadas recursivas fall_true y fall_false según sea necesario dependiendo del operador.

En el caso de ft_binary_recursion, que se encuentra en el listado 5.4.5, como se indica en los cuadros 5.2 y 5.3, tenemos los siguientes casos:

- Para la disyunción, empezamos evaluando A; si A devuelve el valor de falso evaluamos el segundo término, B. Por lo que empezamos invocando fall_false con la nueva etiqueta ftLb—que marca el fin de la evaluación de la expresión completa—como parámetro. Si el segundo término regresa verdadero *caemos* a la términación de la evaluación de la expresión y sólo en caso de que ambos términos se evalúen a falso saltaremos hacia la etiqueta lbl donde cargaremos un valor de falso como resultado de la evaluación completa.

- Para la conjunción, si A se evalúa a falso debemos saltar a la etiqueta lbl donde cargamos el valor de falso como el de la evaluación de la expresión completa. Si A se evalúa a verdadero debemos continuar la evaluación del siguiente término, B. Cuando B se evalúa, si el resultado es falso de nueva cuante saltamos a lbl, de lo contrario permitimos que la evaluación continúe con el siguiente enunciado del programa. De lo anterior, dos llamadas recursivas a fall_true hacen ese trabajo, pasándole como parámetro la misma etiqueta.

Para ff_binary_recursion, que se encuentra en el listado 5.4.6, de nueva cuenta



Cuadro 5.3: Implementación de la conjunción mediante las primitivas fall_true y fall_false.

Listado 5.4.5 Método ft_binary_recursion de la clase CodeGenerator.

```
static void ft_binary_recursion (EmitContext ec, AST ast,
                                  Label lbl)
{
  ILGenerator ig = ec.ig;
  Binary b = ast as Binary;

  switch (b.op) {
  case JSToken.LogicalOr:
    Label ftLb = ig.DefineLabel ();
    fall_false (ec, b.left, ftLb);
    fall_true (ec, b.right, lbl);
    ig.MarkLabel (ftLb);
    break;

  case JSToken.LogicalAnd:
    fall_true (ec, b.left, lbl);
    fall_true (ec, b.right, lbl);
    break;
  }
}
```

Listado 5.4.6 Método ff_binary_recursion de la clase CodeGenerator.

```
static void ff_binary_recursion (EmitContext ec, AST ast,
                                 Label lbl)
{
  ILGenerator ig = ec.ig;
  Binary b = ast as Binary;

  switch (b.op) {
  case JSToken.LogicalOr:
    fall_false (ec, b.left, lbl);
    fall_false (ec, b.right, lbl);
    break;

  case JSToken.LogicalAnd:
    Label ftLb = ig.DefineLabel ();
    fall_true (ec, b.left, ftLb);
    fall_false (ec, b.right, lbl);
    ig.MarkLabel (ftLb);
    break;
  }
}
```

tenemos los siguientes casos:

- Para el caso de la disyunción, si la expresión A se evalúa a verdadero, inmediatamente se salta a la etiqueta de unión o terminación de la evaluación de la expresión completa y se coloca el valor de verdadero como el resultado de la evaluación completa de la expresión. En caso contrario, el control prosigue hacia la evaluación de B, que de manera recursiva se procesa de la misma manera que A.

- Para el caso de la conjunción, empezamos evaluando A; solamente si A se evalúa a verdadero continuamos la evaluación del siguiente miembro, B. Por lo que iniciamos con una invocación a fall_true con la etiqueta ftLb como punto de salida, donde falso será el valor de regreso de evaluación de toda la expresión. Si A se evalúa a verdadero continuamos la evaluación de B; si B se evalúa a falso *caemos* a la etiqueta donde falso es el resultado; en caso contrario saltamos a la etiqueta lbl a la cual saltamos solamente si ambos términos se evaluan a verdadero.

De esta manera queda implemetada la evaluación de *corto circuito*. El modelo es muy sencillo y uniforme lo que hace muy clara la implementación. Una desventaja es que no siempre se genera el código más eficiente posible, sin embargo la claridad y correctez en la implementación lo justifican, además de que para

generar el código óptimo tenemos que utilizar técnicas de análisis de control de flujo, que para este trabajo no se tiene contemplado implementar.

5.4.4. Generación de código del ligado lo más tarde posible

En la sección 5.3.2 vimos cómo se distingue entre cuáles expresiones binarias necesitan de la evaluación lo más tarde posible y las que no; sin embargo el proceso de generar el código adecuado también es interesante. En esta ocasión se maneja cuidadosamente caso por caso el tipo de la expresión a evaluarse, sobre todo porque se pueden hacer algunas mejoras al código generado si se realiza de esta manera.

La implementación del método Emit de la clase Binary se encuentra en el listado 5.4.7.

La idea intuitiva de este algoritmo consiste en que la generación de código se especializa de acuerdo con el tipo de expresión que tenemos, si tenemos una evaluación de una expresión binaria booleana (LogicalAnd y LogicalOr) generamos código de salto (emit_jumping_code); si el operador binario es un corchete izquierdo entonces generamos código de acceso a variables u objetos como si fuesen arreglos (emit_array_access); cuando tenemos un acceso mediante el operador *punto* o de acceso de campo, verificamos si se necesita evaluación lo más tarde posible en cuyo caso emit_late_binding es invocada o si podemos realizar el acceso directamente llamamos a emit_access, para los demás operadores realizamos la evaluación tomando en cuenta la precedencia y dirección a la cual asocian.

Los métodos que queremos enfatizar son:

- emit_jumping_code, se encarga de generar el código adecuado para la evaluación de las expresiones lógicas. Su análisis se explica en la sección 5.4.3.

- emit_array_access, que se encuentra en el listado 5.4.9, implementa toda la lógica de generación de código asociada con el acceso a variables u objetos como arreglos. En la sección 4.2 dijimos que métodos de la clase LateBinding están involucrados para realizar este tipo de evaluación así como los criterios que determinan los argumentos que se le proveen en su uso concreto. Debido a que en ECMAScript no tenemos anotaciones de tipo, cualquier acceso a una variable tiene que realizarse de esta manera, no importando si es un objeto ordinario o un objeto arreglo con lo cual uno podría pensar que el acceso podría realizarse mediante una indirección y desplazamiento explícito. Tomemos un ejemplo concreto de acceso a propiedad mediante un índice y a través del operador punto en el listado 5.4.8.

  Debemos notar que:

  1. La variable x - a la cual se le asigna una instancia del objeto interconstruido Object - se le agregan las propiedades "hola" y "2" al vuelo; si x no hubiese sido iniciada como instancia del objeto interconstruido Object se produciría un error. En principio ese objeto estaba vacío,

Listado 5.4.7 Método Emit de la clase Binary.

```
internal override void Emit (EmitContext ec)
{
  ILGenerator ig = ec.ig;

  if (op == JSToken.LogicalAnd ||
      op == JSToken.LogicalOr)
    emit_jumping_code (ec);
  else if (op == JSToken.LeftBracket) {
    if (!assign)
      get_default_this (ig);
    if (left != null)
      left.Emit (ec);
    emit_array_access (ec);
  } else if (op == JSToken.AccessField) {
    if (late_bind)
      emit_late_binding (ec);
    else
      emit_access (left, right, ec);
  } else {
    emit_operator (ig);

    if (left != null)
      left.Emit (ec);

    if (right != null)
      right.Emit (ec);
    emit_op_eval (ig);
  }
  if (no_effect)
    ig.Emit (OpCodes.Pop);
}
```

Listado 5.4.8 Creación de nuevas propiedades a través de asignación indexada.

```
var x = new Object ();
x [``hola''] = 1;
x [2] = ``hey'';

print (``x ["hola"] = ``, x [``hola'']);
print (``x [2] = ``, x [2]);
print (``x.hola = ``, x.hola);
print (``x.no_existo = ``, x.no_existo);
```

después de las asignaciones podemos acceder a esas propiedades como si fuera un arreglo o mediante el operador punto; aquí está en acción el concepto de objetos dinámicos del cual hablamos en la sección 3.2.5, dedicada a las características de los objetos en ECMAScript.

2. Debemos poder distinguir entre accesos de lectura y de escritura.
3. En caso de tener un acceso de lectura a una propiedad inexistente, no se genera un error en tiempo de compilación como tampoco en tiempo de ejecución sino que se regresa el valor undefined.

La manera en que se implementan estas características se presenta en el listado 5.4.9.

Para lograr (1) hacemos uso del método SetIndexedPropertyValueStatic, cuya firma se muestra en el listado 5.4.10, perteneciente a la clase LateBinding; la estrategia es construir todos los parámetros que esta función necesita, donde:

- obj es el objeto al que se le agregarán las propiedades.
- arguments son los argumentos; donde se encuentra el nombre de la propiedad nueva que se quiere agregar.
- value es el valor que se le asigna a la nueva propiedad.

Para nuestro caso obj viene a ser x, arguments contiene la expresión que está entre corchetes y value es la expresión que se encuentra del lado derecho del operador de asignación. El código CIL generado es el que se muestra en el listado 5.4.11.

Todas las operaciones que se realizan en emit_array_access son $O(1)$ excepto por GetMethod y GetField que pertenecen a System.Reflection y cuya complejidad es $O(n)$ en el peor de los casos, donde n es el número de miembros del tipo, en este caso LateBinding.

Para lograr (2) simplemente checamos si nos encontramos en un contexto de asignación; esto se lleva a cabo en el método Resolve del objeto Assign, que se contruye como parte del proceso de construcción del árbol de sintaxis abstracta, y que cuando invoca el proceso de análisis semántico sobre la expresión situada del lado izquierdo de la asignación le pasa una bandera (assign) con valor de verdadero, indicando que estamos en un contexto de asignación y que una instrucción de guardar (store) y no de cargar (load) tiene que ser generada.

Para la lectura usamos el método CallValue de LateBinding, cuya firma es la que se muestra en el listado 5.4.12, donde:

- thisObj es el valor de this en ese contexto.
- val es el objeto del cual se quiere obtener el valor, en nuestro ejemplo sería x.

Listado 5.4.9 Método emit_array_access de la clase Binary.

```
internal void emit_array_access (EmitContext ec)
{
  ILGenerator ig = ec.ig;
  ig.Emit (OpCodes.Ldc_I4_1);
  ig.Emit (OpCodes.Newarr, typeof (object));
  ig.Emit (OpCodes.Dup);
  ig.Emit (OpCodes.Ldc_I4_0);
  if (right != null)
    right.Emit (ec);
  ig.Emit (OpCodes.Stelem_Ref);

  if (assign) {
    if (right_side != null)
      right_side.Emit (ec);
    ig.Emit (OpCodes.Call,
             typeof (LateBinding).GetMethod (
                 "SetIndexedPropertyValueStatic"));
  } else {
    ig.Emit (OpCodes.Ldc_I4_0);
    ig.Emit (OpCodes.Ldc_I4_1);
    if (InFunction)
      ig.Emit (OpCodes.Ldarg_1);
    else {
      ig.Emit (OpCodes.Ldarg_0);
      ig.Emit (OpCodes.Ldfld,
         typeof (ScriptObject).GetField ("engine"));
    }
  ig.Emit (OpCodes.Call,
     typeof (LateBinding).GetMethod ("CallValue"));
  }
}
```

Listado 5.4.10 SetIndexedPropertyValueStatic.

```
public static void SetIndexedPropertyValueStatic (
        object obj, object [] arguments, object value)
```

Listado 5.4.11 Código CIL generado para un caso de la evaluación lo más tarde posible.

```
ldsfld  object 'JScript_0'::x
ldc.i4.1
newarr [mscorlib]System.Object
dup
ldc.i4.0
ldstr "hola"
stelem.ref
ldc.r8 1
box [mscorlib]System.Double
call void class
    [Microsoft.JScript]Microsoft.JScript.LateBinding::
SetIndexedPropertyValueStatic(object, object[], object)
```

Listado 5.4.12 Método CallValue de la clase LateBinding.

```
public static object CallValue (object thisObj,
                                object val,
                                object [] arguments,
                                bool construct,
                                bool brackets,
                                VsaEngine engine)
```

  - arguments contiene cualquier posible argumento para realizar el acceso, es decir lo que se encuentra dentro de los corchetes. En nuestro ejemplo simplista sería la cadena "hola".
  - construct es una variable booleana que indica si estamos dentro de una expresión donde está involucrado algún constructor. En nuestro ejemplo es falso, y no tiene nada que ver con constructores.
  - brackets es otra variable boolean que nos indica si estamos frente a un acceso a través de índice o identificador. En nuestro ejemplo es verdadero pues precisamente es la operación que queremos realizar.
  - VsaEngine es el motor de ejecución actual y controla la cadena de objetos de activación. Toda ejecución tiene asociada a ella un motor de ejecución que nos brinda los valores de this, etc.

- emit_late_binding se hace cargo de la generación de código que tiene que evaluarse de manera retardada. Para realizar la evaluación hay un conjunto de instrucciones canónicas que siempre se repiten; esas instrucciones están encapsuladas en el método init_late_binding; luego se procede a generar el código para obtener o asignar el valor deseado.

  El código de emit_late_binding se encuentra en el listado 5.4.13.

Listado 5.4.13 Método emit_late_binding de la clase Binary.

```
void emit_late_binding (EmitContext ec)
{
  LocalBuilder local_lb = init_late_binding (ec);
  emit_late_get_or_set (ec, local_lb);
}
```

El método emit_late_get_or_set, que se encuentra en el listado 5.4.14, usa a los métodos auxiliares SetValue y GetNonMissingValue de la clase LateBinding para obtener la propiedad deseada.

Listado 5.4.14 Método emit_late_get_or_set de la clase Binary.

```
void emit_late_get_or_set (EmitContext ec,
                           LocalBuilder local)
{
  ILGenerator ig = ec.ig;

  ig.Emit (OpCodes.Ldloc, local);
  ig.Emit (OpCodes.Dup);

  left.Emit (ec);

  CodeGenerator.load_engine (InFunction, ec.ig);

  ig.Emit (OpCodes.Call,
           typeof (Convert).GetMethod ("ToObject"));

  Type lb_type = typeof (LateBinding);
  ig.Emit (OpCodes.Stfld, lb_type.GetField ("obj"));

  if (assign) {
    right_side.Emit (ec);
    ig.Emit (OpCodes.Call,
             lb_type.GetMethod ("SetValue"));
  } else
    ig.Emit (OpCodes.Call,
             lb_type.GetMethod ("GetNonMissingValue"));
}
```

Cuando tenemos una asignación necesitamos saber cual es el valor que vamos a asignar; por tal razón right, el lado derecho de la asignación tiene que generarse antes de generar la invocación a SetValue; por otro lado, si simplemente queremos obtener el valor de la propiedad en cuestión, pero asegurando que al menos un valor se tiene que regresar, generamos la invocación a GetNonMissingValue que en caso de no existir dicha propiedad regresa un valor no definido. Los métodos en los que se invierte mayor tiempo de cómputo son GetMethod y GetField, que ya dijimos que son $O(n)$ con n el número de miembros del tipo que está realizando la invocación.

- emit_access realiza generación de código específico para el acceso directo a campos estáticos de los objetos interconstruidos. Este método hace uso extenso de los servicios de introspección, pues una vez cumplida la condición de que se puede tipificar fuertemente la expresión binaria en cuestión, entonces accedemos a sus miembros para poder utilizarlos directamente.

Con esos métodos como base fundamental, se abre el espectro del tipo de

programas que se pueden escribir en ECMAScript, brindando así características híbridas de programación orientada a objetos y funcional. Mostraremos algunos ejemplos de programas que son válidos en ECMAScript y que muestran estas capacidades. Los ejemplos aquí presentados fueron tomados del libro de Flanagan [6].

1. Iniciaremos con el uso de funciones anidadas, con el programa que se muestra en el listado 5.4.15.

Listado 5.4.15 Funciones anidadas en ECMAScript.

```
function Hypotenuse (a,b)
{
  function Square (x)
  {
    return x * x;
  }

  return Math.sqrt (Square (a) + Square (b));
}

print (Hypotenuse (1, 1))
```

2. También tenemos la posibilidad de asignar funciones a variables. Ver listado 5.4.16.

Listado 5.4.16 Asignación de expresiones de función.

```
var f = function (x) { return x * x; }
print ("f (10) = ", f (10));

var my_factorial = function fact (x) {
                     if (x == 1)
                       return 1;
                     else
                       return x * fact (x - 1);
                   };
print ("my_factorial (4) = ", my_factorial (4));
```

3. Podemos efectuar aplicaciones de función con funciones definidas en el momento de la aplicación, en el mismo sentido que las funciones anónimas de Scheme[5]. Ver listado 5.4.17.

[5]Nos estamos refiriendo a las *lambda*'s de Scheme.

Listado 5.4.17 Aplicación de función definida al vuelo.

```
var tensquared = (function (x) { return x * x; }) (10);
print ("tensquared = ", tensquared);
```

4. Podemos tener alias a funciones. Ver listado 5.4.18.

Listado 5.4.18 Asignación de funciónes a variables.

```
function Square (x)
{
  return x * x;
}

var a = Square (4);
print ("a = ", a);

var b = Square;
var c = b (5);
print ("c = ", c);
```

5. Podemos asociar funciones a propiedades de objetos. Ver listado 5.4.19.

Listado 5.4.19 Asignación de funciónes a propiedades de objetos.

```
var o = new Object;
o.Square = new Function ("x", "return x * x;");

var y = o.Square (16);
print ("y = ", y);
```

6. Podemos guardar funciones en estructuras de datos explícitamente; por ejemplo, se podría tener un arreglo de funciones y posteriormente aplicar alguna de ellas. Ver listado 5.4.20.

Listado 5.4.20 Arreglo de funciones y aplicación de función mediante acceso indexado del arreglo.

```
var a = new Array (3);

a [0] = function (x) { return x * x; }
a [1] = 20;
a [2] = a [0] (a [1]);

print ("a [2] = ", a [2]);
```

7. Podemos tener como valor de regreso de una función, otra función. Ver listado 5.4.21.

Listado 5.4.21 Funciones como valor de regreso.

```
function f ()
{
  function g (x)
  {
    return x * x;
  }
  return g;
}

print (f ()(10));
```

8. Y finalmente, un ejemplo que muestra todas estas características juntas, es decir, el uso de las funciones como datos. Ver listado 5.4.22

5.4.5. Generación de código de declaración de variables y funciones repetidas

La generación de código de declaración de variables y funciones tiene que cuidar que no se generen campos y funciones en más de una ocasión aún y cuando hay varias declaraciones en el código fuente. Tiene que realizarse de manera manual ya que el generador de código CIL (ILGenerator no considera inválida la operación de incluir en el flujo de código campos y funciones con el mismo identificador; sin embargo el comportamiento de un programa con dichas características no es determinístico, lo cual lo convierte en un programa completamente inútil. Los métodos EmitDecl de la clase VariableDeclaration y variable_defined_in_current_scope de la clase CodeGenerator se encargan de realizar ese trabajo.

Listado 5.4.22 Revisión del uso de funciones como datos.

```
function Add (x, y)
{
  return x + y;
}

function Substract (x, y)
{
  return x - y;
}

function Multiply (x, y)
{
  return x * y;
}

function Divide (x, y)
{
  return x / y;
}

function Operate (operator, operand1, operand2)
{
  return operator (operand1, operand2);
}

var i = Operate (Add,
                 Operate (Add, 2, 3),
                 Operate (Multiply, 4, 5));
print ("i = ", i);

var operators = new Object ();
operators ["Add"] = function (x, y){return x + y;};
operators ["Substract"] = function (x, y){return x - y;};
operators ["Multiply"] = function (x, y){return x * y;};
operators ["Divide"] = function (x, y){return x / y;};

function Operate2 (op_name, operand1, operand2)
{
  if (operators [op_name] == null)
    return "unknown operator";
  else
    return operators [op_name] (operand1, operand2);
}

var j = Operate2 ("Add", "hello",
                  Operate2 ("Add", " ", "world"));
print ("j = ", j);
```

Listado 5.4.23 Método EmitDecl de la clase VariableDeclaration.

```
internal void EmitDecl (EmitContext ec)
{
  object var;

  if ((var = CodeGenerator.variable_defined_in_current_scope (id))
        != null) {
    Type t = var.GetType ();
    if (t == typeof (FieldBuilder))
      field_info = (FieldBuilder) var;
    else if (t == typeof (LocalBuilder))
      local_builder = (LocalBuilder) var;
    return;
  }

  ILGenerator ig = ec.ig;

  if (parent == null ||
      (parent.GetType () != typeof (FunctionDeclaration)
      && parent.GetType () != typeof (FunctionExpression))) {
    FieldBuilder field_builder;
    TypeBuilder type_builder  = ec.type_builder;
    field_builder = type_builder.DefineField (id, this.type,
                        FieldAttributes.Public |
                        FieldAttributes.Static);
    TypeManager.Add (id, field_builder);
    field_info = field_builder;
  } else {
    local_builder = ig.DeclareLocal (type);
    TypeManager.Add (id, local_builder);
  }
}
```

Lo que EmitDecl, que se encuentra en el listado 5.4.23, hace es verificar que en el bloque actual no esté definida en el flujo de CIL una variable con ese nombre, en cuyo caso simplemente asocia la variable ya definida con su variable constructor[6]; durante el proceso de generación de código se mantiene una tabla de dispersión en la clase TypeManager con todos los campos, métodos y variables locales que se van generando; esto es necesario debido a la limitante en la interfaz de programación de introspección (System.Reflection) de no poder obtener los métodos definidos dinámicamente en variables de tipo TypeBuilder; por un lado podría parecer un costo alto; sin embargo debemos considerar que con respecto a eficiencia en las búsquedas es un beneficio, ya que no tenemos que invocar métodos de la interfaz de programación System.Reflection que son más costosas que un acceso a la tabla de dispersión simple. variable_defined_in_current_scope simplemente realiza la búsqueda en la tabla que TypeManager mantiene para decidir, lo cual también es $O(1)$ en el caso promedio y $O(n)$ en el peor caso.

Con esto damos por terminada la revisión de los algoritmos empleados en el compilador para lograr implementar las características dinámicas de ECMAScript en el CLI.

El compilador de ECMAScript forma parte de la distribución oficial de Mono; el código fuente y paquetes pueden obtenerse en la página del proyecto en http://mono-project.com/Downloads.

[6]En el sentido de constructor de campo (FieldBuilder) o constructor de variable local (LocalBuilder).

Capítulo 6

Conclusiones

Durante el desarrollo de la computación uno de sus principales objetivos a sido el de construir abstracciones y herramientas que permitan expresarlas de manera mucho mas clara y concisa; es por ello que se han inventado distintos lenguajes de programación como medio para alcanzar dicha meta, en un principio se usaba lenguaje ensamblador, luego surgió Fortran seguido de Lisp y la lista continúa creciendo, sin embargo es muy importante mencionar de que manera son ejecutados dichos lenguajes, muchos de ellos eran traducidos a lenguaje de máquina directamente, otros se interpretaban y en tiempos recientes son traducidos a representaciones intermedias que son posteriormente traducidas a lenguaje de máquina, este es el caso de ECMAScript/CIL. La mayoría de las implementaciones de ECMAScript eran interpretes, por lo cual el desempeño de programas escritos en este lenguaje no era el óptimo, sin embargo con este nuevo enfoque se mejoró esa característica, en particular con programas y pruebas sencillas comparado con Rhino (intérprete escrito en Java) el código generado por mjs, al ser ejecutado lo superó en velocidad de ejecución. En este trabajo se construyó un compilador que genera código CIL el cual puede ser ejecutado por máquinas virtuales o compiladores al vuelo, no importando que sean propietarias o libres como es el caso de Mono, Portable .NET y Microsoft .NET Platform. Una de las cosas más ilustrativas de este trabajo fue el poder dar soporte para funciones de primer orden, principalmente porque los lenguajes mas usados dentro de la plataforma .NET como C#, VB .NET y C++ no las ofrecen, y demuestra que dichas características pueden ofrecerse de manera natural y eficiente en un ambiente de ejecución diseñado para lenguajes de programación con características principalmente orientadas a objetos basado en clases y poco preocupadas por aquellas que pertenecen principalmente a la programación funcional y basada en prototipos.

Este trabajo es el principio de una serie de desarrollos que pueden realizarse, en particular para avalar la correctez y tener un sustento al realizar cambios en el compilador sería muy benéfico poseer un conjunto de pruebas hechas por terceros, de particular interés es el lograr que el conjunto de pruebas de Mozilla/Rhino y la de Microsoft/Rotor fueran compiladas exitosamente por mjs. De

gran importancia es el implementar el soporte para la ejecución de los programas generados, para este trabajo se usó principalmente el soporte de ejecución de Microsoft/Rotor, sin embargo para brindar una alternativa completamente libre es necesario implementarlo del lado de Mono. De gran valor sería el implementar la interoperabilidad ECMAScript/.NET en un grado mayor, como puede ser el brindar acceso a la biblioteca de clases de .NET a través de características como importar y exportar espacios de nombre, proveer anotaciones de tipo en el lenguaje, definición explícita de clases e interfaces, herencia, modificadores de visibilidad entre algunos otros. Un posible candidato para extender relevantemente la semántica del lenguaje es el agregar soporte para continuations a él, Rhino ya posée dicho soporte y ha resultado de gran beneficio para los desarrolladores de aplicaciones web en lo que respecta al manejo del flujo de datos. Es por todo esto que me atrevo a decir que varias ventanas para seguir explorando continúan abiertas.

Bibliografía

[1] *ECMAScript Language Specification.* ECMA International, third edition, 1999.

[2] *C# language specification.* ECMA International, second edition, 2002.

[3] *Common Language Infrastructure (CLI).* ECMA International, second edition, 2002.

[4] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge University Press, second edition, 2002.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press, second edition, 2001.

[6] David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly and Associates, Inc, fourth edition, 2002.

[7] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press, second edition, 2001.

[8] John K. Gough. *Compiling for the .NET common language runtime (CLR).* Prentice Hall PTR, first edition, 2002.

[9] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, first edition, 1997.

[10] Benjamin C. Pierce. *Types and programming languages.* MIT Press, first edition, 2002.

[11] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java.* Pearson Education Limited, first edition, 2000.

[12] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* Addison-Wesley Publishing Company Ltd., first edition, 1996.