PROYECTO LAMBDA:
HACIA UN FRAMEWORK DE IMPLEMENTACIÓN Y EJECUCIÓN DE MÁQUINAS ABSTRACTAS GENERADAS A PARTIR DE UN LENGUAJE DE PROPÓSITO ESPECÍFICO

Juan David Ospina Lozano

UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
Departamento de Ingeniería de Sistemas y Computación
Bogotá, Colombia

Julio de 2009

Nota de Aceptación

Revisado y aprobado por:

Silvia Takahashi, Ph.D.
Profesor Asociado
Asesor de Tesis

Rafael Gómez Díaz, M.S.
Profesor Asociado
Jurado Interno

Luis Daniel Benavides Navarro, Ph.D.
Profesor Universidad EAN
Jurado Externo

Bogotá, Julio 28 de 2009

PROYECTO LAMBDA:
HACIA UN FRAMEWORK DE IMPLEMENTACIÓN Y EJECUCIÓN DE MÁQUINAS ABSTRACTAS GENERADAS A PARTIR DE UN LENGUAJE DE PROPÓSITO ESPECÍFICO

Presentado por

JUAN DAVID OSPINA LOZANO

Tesis de grado presentada como requisito parcial
para optar al título de:

Magíster en Ingeniería de Sistemas y Computación

Asesor
Silvia Takahashi, Ph.D
Profesor Asociado

UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
Departamento de Ingeniería de Sistemas y Computación
Bogotá, Colombia
Julio de 2009

A mis Padres.

Agradecimientos

Agradezco en primer lugar a Dios por permitirme nacer en el seno de mi hogar.

Agradezco a mis padres su invaluable cariño, apoyo y orientación durante toda mi vida.

Agradezco a mi hermana su compañía y permanentes palabras de aliento, también a mi familia la motivación recibida.

Agradezco a mi asesora de tesis, la profesora Silvia Takahashi, por la confianza permanente que ha depositado en mi, y por su invaluable asesoría en este trabajo.

Agradezco a los profesores Rafael Gómez y Daniel Benavides su participación como jurados de esta investigación.

Agradezco a la École des Mines de Nantes (Francia) la calurosa acogida y la orientación recibida durante mi pasantía. En particular, agradezco su apoyo a los profesores Jean-Claude Royer y Annya Réquilé.

Agradezco a la Universidad de los Andes, y en particular a los profesores del departamento de Ingeniería de Sistemas y Computación, toda la formación que recibí a lo largo de mis estudios de pregrado y maestría. En particular agradezco a los profesores Germán Bravo, Jorge Villalobos, Rubby Casallas y Juan Diego Jiménez su confianza.

Agradezco a todos mis amigos su presencia y apoyo... y los "coffee breaks".

Muchas gracias a todos.

Juan David Ospina

Bogotá, Julio 10 de 2009

Índice general

1. Introducción 10

2. Planteamiento del Problema 12

3. Objetivos 13
- 3.1. Objetivo General 13
- 3.2. Objetivos Específicos 13

4. Estado del Arte 15
- 4.1. Máquinas Abstractas 15
  - 4.1.1. Estructura de la Compilación 16
  - 4.1.2. Diseño de una Máquina Abstracta 17
  - 4.1.3. Estructuras de una Máquina Abstracta 17
  - 4.1.4. Conjunto de Instrucciones de una Máquina Abstracta 20
- 4.2. Lenguajes de Programación y Máquinas Abstractas 21
  - 4.2.1. Lenguajes Imperativos 21
  - 4.2.2. Lenguajes Funcionales 26
  - 4.2.3. Lenguajes Lógicos 29
- 4.3. Lenguajes de Propósito Específico 32
  - 4.3.1. Propuesta de Spinellis 33

5. Propuesta de Solución 36
- 5.1. Diseño del Lenguaje de Propósito Específico AMDL 37
- 5.2. Esquema de Implementación de AMDL 40
- 5.3. IDE de Máquinas Abstractas 41
  - 5.3.1. ECLIPSE 43

5.4. LAMBDA: Framework de Máquinas Abstractas 43

6. Implementación de la Solución **47**
6.1. Traductor Dirigido por Sintaxis 47
6.2. IDE de Máquinas Abstractas en ECLIPSE 49

7. Casos de Estudio **52**
7.1. Lenguajes Imperativos: Máquina P 52
7.2. Lenguajes Funcionales: Máquina CAM 53
7.3. Lenguajes Lógicos: Máquina WAM 53

8. Contribuciones y Trabajo Futuro **58**
8.1. Contribuciones 58
8.2. Trabajo Futuro 59

9. Conclusiones **60**

Bibliografía **61**

Índice de Tablas

4.1. *Conjunto de instrucciones de la Máquina P* 26
4.2. *Conjunto de instrucciones de la Máquina CAM* 30
4.3. *Conjunto de instrucciones de la Máquina WAM* 32

Índice de Figuras

1.1. *Ejecución de programas en una máquina virtual* 11

4.1. *Esquema de traducción de una máquina virtual* 16
4.2. *Diseño estructural de una máquina abstracta* 17
4.3. *Marco de pila típico en una máquina abstracta* 19
4.4. *Administración dinámica de memoria para la pila y el heap en una máquina abstracta* 19
4.5. *Implementación de subrutinas en los lenguajes imperativos* 23

5.1. *Gramática BNF de AMDL. Parte 1* 38
5.2. *Gramática BNF de AMDL. Parte 2* 39
5.3. *Gramática BNF de AMDL. Parte 3* 40
5.4. *Esquema de traducción dirigida por sintaxis de AMDL* 41
5.5. *Implementación en Java de una máquina abstracta desarrollada por Takahashi, S. [25]* 42
5.6. *Arquitectura de plug-ins de ECLIPSE [17]* 44

6.1. *Implementación de la traducción dirigida por sintaxis de AMDL en ANTLR* 48
6.2. *Generación del traductor en ANTLR* 49
6.3. *Ambiente de desarrollo de AMDL en ECLIPSE* 50
6.4. *Funcionalidades del ambiente de desarrollo de AMDL* 50
6.5. *Generación de la implementación de la máquina abstracta en AMDL* 51
6.6. *Máquina virtual generada a partir de una especificación en AMDL* 51

7.1. *Especificación en AMDL de la Máquina P* 54
7.2. *Especificación en AMDL de la Máquina CAM* 55
7.3. *Especificación en AMDL de la Máquina WAM* 57

CAPÍTULO 1

Introducción

Los lenguajes de programación han sido uno de los pilares fundamentales de las ciencias de la computación en toda su historia. Su objetivo principal es brindar una notación para expresar soluciones a problemas susceptibles de ser resueltos en un computador. Todo el software que usamos en nuestra vida diaria fue desarrollado en algún lenguaje de programación. Por tal razón, el diseño y la implementación de lenguajes es un área de estudio de gran importancia actualmente, y lo ha sido también en las últimas décadas.

La etapa de diseño de un lenguaje se centra principalmente en definir la estructura y el significado del mismo, es decir, su sintaxis y semántica. Adicionalmente, en la etapa de diseño de un lenguaje se especifican detalles como los tipos de datos a ser implementados, la estructura de flujo de control, etc. El objetivo principal de un diseñador de lenguajes debe ser mantener un equilibrio entre una simplicidad sintáctica y una semántica poderosa.

Sin embargo, el diseñador de lenguajes afronta una tarea adicional: definir las características del ambiente de ejecución del lenguaje diseñado, es decir, de su implementación. Uno de los retos ineludibles que se presenta al diseñar un lenguaje es siempre tener en mente la dificultad de implementar cierta sintaxis y semántica en un ambiente de ejecución dado. La combinación sintaxis-semántica de un lenguaje siempre se ve afectada por requerimientos no funcionales como: recursos limitados de hardware, desempeño, distribución, etc.

Tradicionalmente, han existido dos grandes aproximaciones para la implementación de lenguajes de programación: compilación e interpretación.[6] En la compilación, un programa en un lenguaje es traducido a una representación del mismo en otro lenguaje, generalmente de más bajo nivel. Si esta representación generada es ejecutable, entonces el usuario puede proceder a ejecutarla. En cambio, un intérprete ejecuta un programa de entrada instrucción

por instrucción. Al ir ejecutando el programa, el intérprete solicita entradas (si las hay) y produce salidas.

Cada una de estas aproximaciones tiene sus ventajas y desventajas. Un intérprete es dos o tres órdenes de magnitud más lento que un compilador[1], debido a que todas las "decisiones"tienen que ser tomadas en tiempo de ejecución. Sin embargo, el uso de un intérprete tiene sus ventajas como: reporte efectivo de errores, portabilidad, etc.

Usar la compilación no es posible cuando el lenguaje que se desea implementar desarrolla acciones en tiempo de ejecución dependiendo de los datos de entrada. Esto es conocido como *late binding.*[22] Lenguajes como LISP y Prolog usan esta aproximación, por lo tanto, son lenguajes interpretados. Sin embargo, existe otra aproximación de procesamiento de lenguajes que combina las ventajas de la compilación y la interpretación: las máquinas abstractas.

En el enfoque de las máquinas abstractas, un programa fuente es traducido por un compilador a una representación intermedia. Luego, el programa traducido es interpretado, es decir, ejecutado instrucción por instrucción por la implementación de la máquina abstracta: la máquina virtual. Esta aproximación tiene la ventaja, entre muchas otras, que un programa traducido a la representación intermedia puede ser ejecutado en otro computador a través de una red. Esta es la aproximación usada por la máquina virtual de Java[16], la cual ha puesto vigente la opción de las máquinas abstractas para implementar lenguajes de programación. El esquema de ejecución de un programa en una máquina virtual es mostrado en la Figura 1.1.

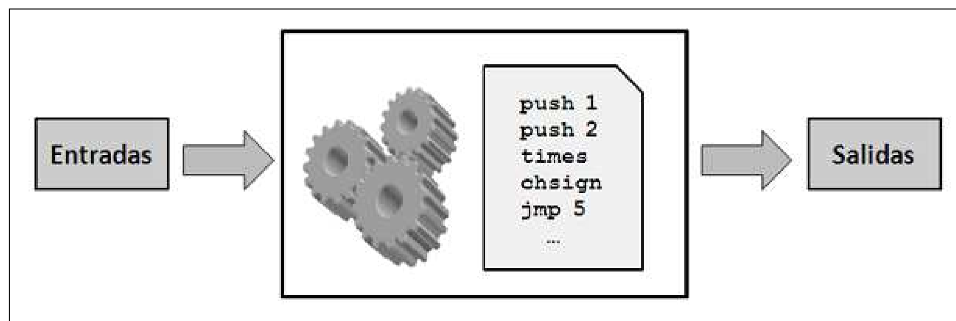

Figura 1.1: *Ejecución de programas en una máquina virtual*

CAPÍTULO 2

Planteamiento del Problema

El enfoque de máquinas abstractas para procesar lenguajes tiene unas ventajas muy importantes, sin embargo, cuando un diseñador de un lenguaje decide implementarlo en una máquina abstracta, se encuentra con un panorama sombrío: debe diseñar la máquina, debe implementar sus estructuras y, además, desarrollar un conjunto de instrucciones que permitan manipular esas estructuras. Usualmente, esta labor se desarrolla en un lenguaje de propósito general con el inconveniente de tener que desarrollar todo desde cero.

Por tal motivo, una de las posibles soluciones a este problema es proporcionarle al diseñador un *framework* de especificación y ejecución de máquinas abstractas que permita un ágil desarrollo de prototipos de máquinas. Esto con el fin de poder experimentar con todas las prestaciones que puede ofrecer una máquina abstracta en la tarea de servir de ambiente de ejecución para un lenguaje de programación. La especificación de una máquina abstracta se hará por medio de un lenguaje de propósito específico (*Domain Specific Language* - DSL). Una vez validada esta especificación, ésta será procesada por el *framework* para dar lugar a una implementación que puede ser usada por el diseñador de lenguajes como punto de partida para procesar su lenguaje.

La idea de desarrollar un *framework* de estas especificaciones surge como continuación del proyecto final del curso de Diseño de Lenguajes propuesto por la profesora Silvia Takahashi [25] en la Maestría en Ingeniería de Sistemas y Computación de la Universidad de los Andes en el año 2006.

CAPÍTULO 3

Objetivos

3.1. Objetivo General

Desarrollar la especificación, e implementar un prototipo, de un *framework* que permita la implementación y ejecución de máquinas abstractas especificadas a partir de un lenguaje de propósito específico. Este *framework* se denominará LAMBDA (*Languages and Abstract Machines, building and definition application*).

3.2. Objetivos Específicos

1. Diseñar un lenguaje de propósito específico que permita modelar una máquina abstracta en términos de su estructura e instrucciones asociadas. Este lenguaje se denominará AMDL (*Abstract Machine Definition Language*).

2. Implementar un ambiente de desarrollo integrado (*Integrated Development Environment* - IDE) que sirva como apoyo para la especificación y ejecución de máquinas abstractas desarrolladas a partir de AMDL. Este ambiente de desarrollo será implementado como una extensión de la plataforma ECLIPSE [17].

3. Desarrollar un generador de máquinas abstractas que permita crear una implementación de una máquina específica en Java a partir de una especificación en AMDL. Este generador será desarrollado por medio de un compilador de compiladores, e integrado por medio de un *plug-in* a la plataforma ECLIPSE.

4. Identificar las posibles extensiones que pueden desarrollarse en el lenguaje AMDL y en el *framework* LAMBDA para que, a futuro, sea posible el desarrollo de máquinas

abstractas que incluyan funcionalidades como verificación y optimización de código, recolección de basura, etc.

CAPÍTULO 4

Estado del Arte

4.1. Máquinas Abstractas

En las ciencias de la computación, el término máquina abstracta es usado para categorizar modelos de computación teóricos como las máquinas de estados finitos, máquinas de Turing, etc.

Adicionalmente, una máquina abstracta también se define como un modelo de ambiente de ejecución para ayudar a cerrar la brecha entre un lenguaje de alto nivel y una arquitectura de hardware. Es en una implementación de una máquina abstracta donde los programas escritos en un lenguaje de alto nivel pueden ser ejecutados.[4] En adelante, el término "máquina abstracta"se referirá a esta última definición, por ser la pertinente para esta investigación.

La aproximación de las máquinas abstractas ha permitido el diseño de ambientes de ejecución para lenguajes que, por su naturaleza, no se adaptan a arquitecturas de hardware tipo von Neumann, o cuya implementación es muy compleja. En estos casos, la máquina abstracta modela, y a la vez simplifica, las estructuras e instrucciones de esa arquitectura de hardware no convencional, permitiendo enfocar esfuerzos en asuntos como el diseño del lenguaje y no en aspectos de implementación de bajo nivel.

Algunos autores identifican a una máquina abstracta como "máquina virtual"y viceversa, es decir, no hacen ninguna distinción entre estos dos términos.[4] Sin embargo, es posible ver la máquina abstracta como una especificación de un ambiente de ejecución, y la máquina virtual como la implementación en software de dicha especificación. Esta relación se puede comparar con la existente entre un algoritmo y un programa, siendo el primero

la especificación de una solución, y el segundo, la implementación del algoritmo en un lenguaje particular. En esta investigación, se especificará lo más posible cuando se habla de una máquina abstracta o de una máquina virtual.

Adicionalmente, una máquina virtual también se define como la simulación en software de una o más capas de un sistema operacional, en particular, múltiples máquinas virtuales permiten la ejecución de diversos sistemas operativos en una misma plataforma de hardware.[15]

Las máquinas abstractas no deben ser confundidas con los intérpretes. Una máquina abstracta no es un intérprete en el sentido estricto del término, sino que extiende esta definición para tratar problemas como la distribución, portabilidad, seguridad, interoperabilidad, versatilidad en plataformas, etc.

4.1.1. Estructura de la Compilación

Las máquinas abstractas contienen un conjunto de estructuras que permiten la ejecución de un determinado programa. Sin embargo, la implementación de la máquina abstracta, es decir, la máquina virtual, no ejecuta directamente el programa escrito en un lenguaje de programación de alto nivel. Concretamente, la máquina ejecuta una *representación* del programa original, escrito en términos del conjunto de instrucciones de la máquina abstracta. Esta representación intermedia es construida a partir de la traducción, o compilación, de las instrucciones de alto nivel a las instrucciones particulares de la máquina abstracta. Ver figura 4.1.

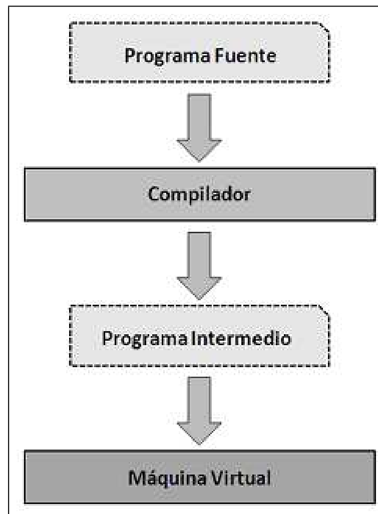

Figura 4.1: *Esquema de traducción de una máquina virtual*

Las máquinas abstractas son usadas como un lenguaje intermedio para la compilación. Como resultado, la implementación de un lenguaje de programación de alto nivel se divide en dos fases: La implementación del compilador y la implementación de la máquina abstracta, es decir, la máquina virtual.[4]

4.1.2. Diseño de una Máquina Abstracta

El diseño de una máquina abstracta debe satisfacer los requerimientos de ejecución de un lenguaje, o familia de lenguajes de alto nivel. Esto ocurre al implementar las instrucciones del lenguaje relacionadas con tipos de datos, procedimientos, parámetros, flujo de control, etc.

Una máquina abstracta generalmente es desarrollada por medio de una estrategia de ensayo y error. Concretamente, se hace un estudio de las estructuras del lenguaje a ser procesado por la máquina y a la par de este proceso, se va diseñando la máquina abstracta a la medida de las necesidades del lenguaje. Adicionalmente, se debe identificar las instrucciones que deben administrar las estructuras de la máquina. Estas instrucciones están reunidas en el llamado *instruction set* o conjunto de instrucciones de la máquina.

4.1.3. Estructuras de una Máquina Abstracta

En general, las máquinas abstractas se componen de una o más estructuras que, conjuntamente, dan soporte al procesamiento del lenguaje. Ver figura 4.2.

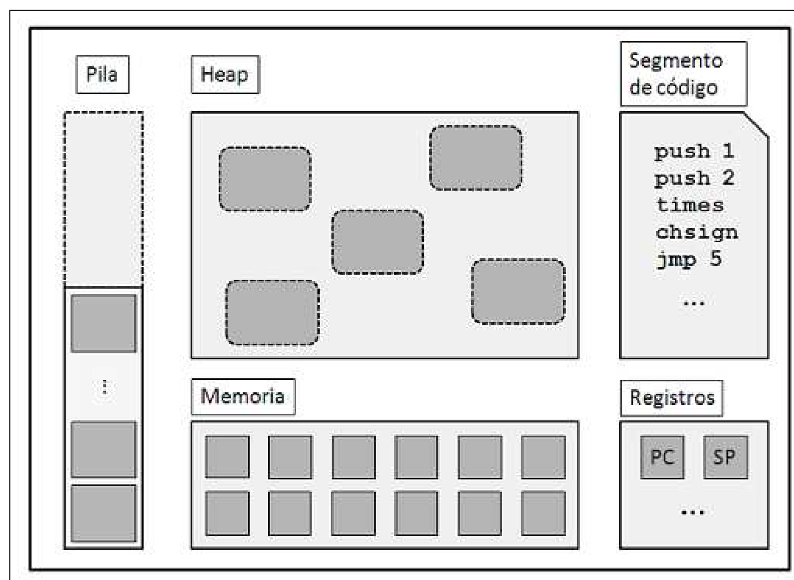

Figura 4.2: *Diseño estructural de una máquina abstracta*

Un diseñador de máquinas abstractas puede construir una máquina a partir de las siguientes estructuras básicas:

- Segmento de Código
- Pila
- *Heap*
- Memoria
- Registros

A continuación, se describirá en detalle cada una de las estructuras anteriores.

Segmento de Código

Es una estructura de almacenamiento que contiene una secuencia de instrucciones a ser ejecutadas por la máquina abstracta. Estas instrucciones pertenecen al *instruction set* de la máquina. En general, el segmento de código tiene un tamaño fijo que es establecido en la etapa de compilación a partir del número de instrucciones que componen el programa a ser procesado. Esta zona de memoria es de solo lectura, esto con el objetivo de prevenir que el crecimiento de la memoria correspondiente a la pila o al *heap* pueda sobrescribir la información del código ejecutable.

Pila

Es una estructura LIFO (*Last in - First out*), es decir, el último elemento en entrar a la estructura es el primero en salir de ella. Las dos operaciones fundamentales de una pila son *push* y *pop*. La operación *push* ingresa un elemento a la pila ubicándolo en el tope, y la operación *pop* retira de la pila el elemento que se encuentre en el tope de la misma.

En una máquina abstracta para procesar lenguajes, una pila almacena elementos de un solo tipo. Puede haber múltiples pilas si así se requiere: pilas de evaluación (para evaluar expresiones), pilas de ejecución, etc. En una pila de ejecución el elemento fundamental que se almacena es llamado Marco de Pila (o simplemente marco). Algunos autores lo denominan también registro de activación (*activation record*) [1].

Un marco de pila es creado y almacenado en la pila principalmente cuando es iniciado un llamado a una subrutina. Cada instancia de una subrutina creada en tiempo de ejecución tiene su propio marco. Cada marco está compuesto de la siguiente información relativa a la subrutina: enlace dinámico (o dirección de retorno de la subrutina), parámetros, variables locales, e información temporal creada por la subrutina. Ver figura 4.3.

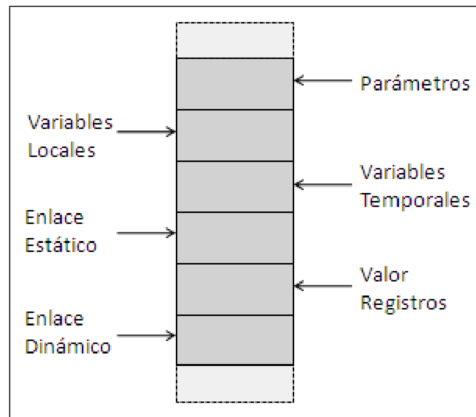

Figura 4.3: *Marco de pila típico en una máquina abstracta*

Heap

Es una región de almacenamiento en la cual se almacenan elementos que pueden ser creados, modificados o eliminados en tiempo de ejecución. Esta región en memoria es necesaria cuando el lenguaje desea implementar estructuras de datos cuyo tamaño puede cambiar como resultado de una operación del lenguaje. Esencialmente estructuras dinámicas como conjuntos, listas, árboles, grafos, etc.

En los lenguajes donde el manejo de la memoria dinámica no se delega en el usuario del lenguaje, se debe implementar un mecanismo que administre el ciclo de vida de los elementos del *heap*. Concretamente, se debe implementar mecanismos de administración de memoria como el recolector de basura.[22]

En general, las implementaciones del *heap* y de la pila de una máquina abstracta tienen un tamaño que puede disminuir o aumentar de acuerdo a las necesidad de las estructuras, siempre y cuando no se sobrepase la memoria libre de la máquina reservada para tal fin. Ver figura 4.4.

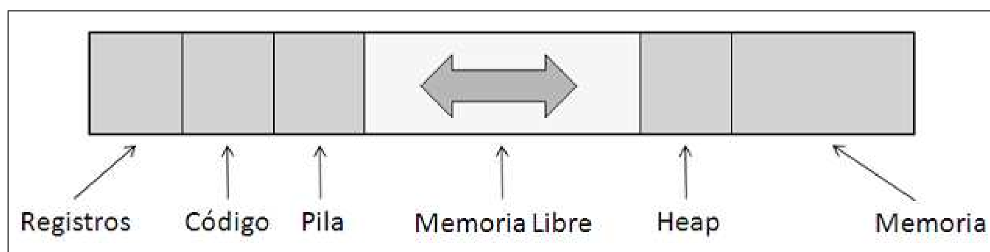

Figura 4.4: *Administración dinámica de memoria para la pila y el heap en una máquina abstracta*

El término *heap* aplicado a esta investigación no debe ser confundido con la implementación de una cola de prioridad basada en árboles.

Memoria

Es una región de almacenamiento de datos, sin destinación específica, donde es posible guardar información que no se desee almacenar en las otras estructuras de la máquina (pila, *heap* o registros) por motivos de implementación. En particular, las variables globales a un programa se almacenan usualmente en la memoria.

Registros

Los registros son elementos especiales de almacenamiento, de tamaño reducido, que pueden ser accedidos muy rápidamente. Usualmente, el número de registros en una máquina abstracta, así como en una arquitectura de hardware, es limitado. En una máquina abstracta usualmente existen dos registros de propósito específico llamados apuntador de pila (SP o *stack pointer*), y contador de programa (PC o *program counter*). El registro SP se encarga de mantener una referencia al marco presente en el tope de la pila. En algunas máquinas abstractas de más bajo nivel, el registro SP apunta a la primera posición libre en la pila. El registro PC almacena la referencia a la siguiente instrucción del segmento de código a ser ejecutada por la máquina abstracta.

4.1.4. Conjunto de Instrucciones de una Máquina Abstracta

El conjunto de instrucciones de una máquina abstracta contiene instrucciones de relativo bajo nivel que implementan un lenguaje de alto nivel de manera parcial o total. Este conjunto de instrucciones es totalmente independiente de la arquitectura del sistema operativo donde se vaya a ejecutar la implementación de una máquina abstracta.

Al ser una máquina abstracta una representación de alto nivel de una arquitectura de hardware, el conjunto de instrucciones de una máquina abstracta guarda cierto parecido al de una arquitectura de hardware típica. En general, como se verá más adelante, la mayoría de máquinas abstractas centran su estructura en una pila. Por tanto, el conjunto de instrucciones de una máquina abstracta es orientado a la pila (*stack-oriented*). El conjunto de instrucciones de una máquina abstracta orientada a la pila contiene las siguientes clases de instrucciones:

- Operaciones de lectura y escritura de valores constantes y variables sobre las estructuras de la máquina.
- Operaciones aritméticas: suma, resta, multiplicación, división, módulo, etc.
- Operaciones lógicas: *and*, *or*, *xor*, etc.
- Operaciones de conversión de tipos de datos.

- Operaciones condicionales y no condicionales de flujo de control.
- Operaciones de invocación de subrutinas y retorno de valores.

Al diseñar un conjunto de instrucciones para una máquina abstracta, se debe procurar que éste quede lo más reducido posible, es decir, incluir el menor número de instrucciones. Esto con el fin de que la ejecución de un programa sea lo más eficiente posible.

4.2. Lenguajes de Programación y Máquinas Abstractas

A continuación, se estudian los principales paradigmas de programación, a saber: imperativo, funcional y lógico en términos de las estructuras de sus lenguajes y de sus ambientes de ejecución. Concretamente, para cada paradigma, se estudian sus principales características y, adicionalmente, las máquinas abstractas existentes que dan soporte a la ejecución de sus lenguajes asociados.

4.2.1. Lenguajes Imperativos

Los lenguajes imperativos han sido predominantes en la historia reciente de la computación. El modelo imperativo de programación surgió a partir del trabajo de matemáticos como Alan Turing y Stephen Kleene. A pesar de que estos científicos de la computación trabajaron de manera independiente, lograron derivar formalmente el concepto de algoritmo. Esta definición se fundamentó en conceptos como los autómatas, manipulación simbólica, combinatoria, etc.

El modelo formal subyacente a los lenguajes imperativos es la máquina de Turing. Una máquina de Turing hace cálculos de una manera imperativa al cambiar los valores de las celdas de la cinta, tal como funcionan los lenguajes imperativos al cambiar el valor de las variables.[5]

En un lenguaje imperativo, el enfoque de programación se encuentra en cómo el computador debe realizar alguna tarea. En estos lenguajes, el modelo de computación es la modificación de variables. La asignación es el principal medio para realizar estos cambios. Una asignación toma dos argumentos: un valor, y una referencia en memoria de la variable a la cual se le va a actualizar su valor.

Todo algoritmo imperativo puede ser expresado por medio de la secuenciación de declaraciones, condicionales y ciclos. El flujo de control de los lenguajes imperativos se basa enteramente en el concepto de secuenciación debido a que existen declaraciones, por ejemplo asignaciones, que deben ser ejecutadas una tras otra. La secuenciación es el principal medio de controlar la ejecución de las asignaciones. Cuando una declaración sigue a otra, la primera es ejecutada antes que la segunda.[22]

Los lenguajes imperativos, al estar basados fundamentalmente en el concepto de asignación de variables, son muy susceptibles a los efectos de borde. Al cambiar el valor de una variable, es afectado el valor de las evaluaciones de las expresiones que involucren esa variable. De esta manera, la computación del programa es afectada, llevando a efectos de borde.[6]

A continuación, se listan los principales conceptos presentes en los lenguajes imperativos. Cada uno de estos conceptos debe ser representado en estructuras y secuencias de instrucciones de la máquina abstracta.[1][22]

Variables

Son contenedores de datos cuyos contenidos (valores) pueden ser cambiados durante la ejecución de un programa. El valor de una variable es cambiado por medio de la ejecución de declaraciones (*statements*) tales como las asignaciones.

Las variables en un programa son identificadas por medio de nombres, así mismo, los procedimientos, constantes y variables, son identificados por nombres.

En general, las variables pueden ser recopiladas en estructuras de datos tales como arreglos y registros. Los valores actuales de las variables de un programa conforman el estado del mismo en ese momento.

Expresiones

Son declaraciones formadas a partir de constantes, variables y operadores las cuales son evaluadas en tiempo de ejecución. El valor de una expresión, en general es dependiente del estado del programa, es decir, del valor de las variables involucradas en la expresión.

Especificación Explícita del Flujo de Control

En los lenguajes imperativos, el flujo de control permite al programador, por medio de instrucciones de alto nivel, especificar el orden de ejecución de un conjunto de instrucciones. Estas instrucciones de flujo de control implementan los conceptos de condición e iteración, fundamentales en los lenguajes de este paradigma.

Adicionalmente, el flujo de control se puede alterar por medio de llamados a subrutinas, únicamente si el lenguaje implementa este concepto. En ciertos lenguajes imperativos, el flujo de control se puede alterar incondicionalmente por medio de la instrucción *goto*.

Detalles de Implementación

Los lenguajes imperativos que implementan el concepto de subrutinas, necesariamente tienen un esquema de manejo de memoria basado en una pila de ejecución, también llamada “pila de ambientes”. Cuando el lenguaje implementa recursión en las subrutinas, la creación y el llamado de cada subrutina da lugar a nuevos marcos de ejecución en la pila. Cuando la subrutina invocada finaliza su ejecución, los espacios en la pila de la subrutina son liberados.

Adicionalmente, los lenguajes imperativos no son capaces de crear funciones en tiempo de ejecución (funciones de primera clase o *first-class functions*). Esto indica que el concepto de subrutina debe implementarse por medio de un mecanismo basado en una pila y no por medio de un *heap*. Ver figura 4.5.

En general, los lenguajes imperativos son estáticamente tipados (*statically typed*). Esto implica que no es posible incluir procedimientos dentro de procedimientos, adicionalmente, los lenguajes puramente imperativos no implementan tipos de dato que puedan cambiar su tamaño en tiempo de ejecución. Estos hechos tienen como consecuencia de diseño descartar la presencia de un *heap* en la máquina abstracta de un lenguaje puramente imperativo para el manejo de subrutinas. Sin embargo, si el lenguaje implementa tipos de dato estructurados que pueden cambiar en tiempo de ejecución, como las listas, la presencia de un *heap* es imperativa.

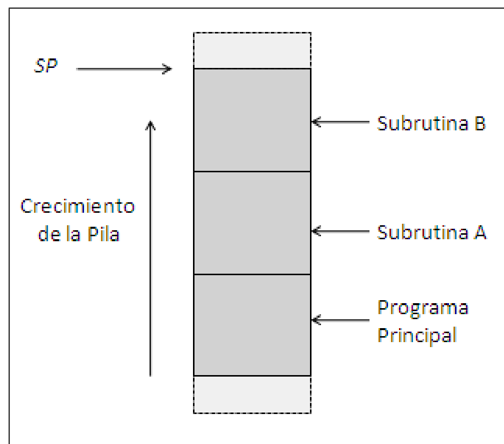

Figura 4.5: *Implementación de subrutinas en los lenguajes imperativos*

Pascal y la Máquina P

La máquina P es el enfoque de procesamiento de Pascal más popular que involucra interpretación. Se desarrolló en la década de 1970 para lograr que Pascal fuera más portable y, adicionalmente, implementar un depurador de bajo nivel. La estructura de la máquina P está compuesta de una zona de memoria llamada *data store* y una zona de registros. [2][5]

La zona de memoria, o *data store*, está dividida de la siguiente manera:

- Pila
- *Heap*
- Zona de memoria para constantes

- Segmento de código

La zona de registros contiene 4 registros, a saber:

- **PC** (*Program counter*). Referencia a la instrucción actual siendo ejecutada.
- **SP** (*Stack pointer*). Referencia al tope de la pila.
- **MP** (*Mark stack pointer*). Referencia a la base del marco que se encuentra en el tope de la pila.
- **NP** (*New pointer*). Referencia al inicio del espacio libre hacia donde puede crecer el *heap*.

La mayoría de las operaciones de la máquina P involucran a la pila, por ejemplo, llamado a subrutinas, paso de parámetros, etc. En la pila, se almacenan marcos o ambientes de ejecución. Un marco es creado y ubicado en la pila cada vez que una subrutina es invocada. Una llamada a subrutina en Pascal implica la administración de información que se encuentra en la pila como parámetros, direcciones de retorno, etc. El primer marco, o marco principal, es creado automáticamente cuando se ejecuta el programa principal.

La información que es susceptible de cambiar en tiempo de ejecución es ubicada en el *heap*. Para ello, existen instrucciones de alto nivel en el lenguaje Pascal que delegan en el programador la administración de la memoria dinámica. Estas instrucciones son: *mark* (reservar memoria) y *release* (liberar memoria).

Estructura de un Marco de Ejecución en la Máquina P

A continuación, se describe en detalle cada una de las estructuras que componen el marco de ejecución de la máquina P.[5]

- El apuntador MP es usado para ajustar el valor del apuntador SP cuando el marco del tope de la pila es eliminado. Adicionalmente, es usado como puerta de entrada para acceder a las variables locales del marco en cuestión. El apuntador SP además es usado para referenciar el punto donde se debe ubicar el nuevo marco.
- La zona de parámetros es usada para almacenar los parámetros transferidos a la subrutina asociada al marco. Los parámetros pueden almacenarse en forma de valores o de direcciones de memoria (absolutas o relativas) dependiendo si la modalidad de paso de parámetros que se implementó fue por valor o por referencia.
- El valor de la función es el área encargada de almacenar el valor de retorno de la subrutina si esta es una función.

- El enlace estático es usado como referencia para acceder a variables localizadas en el marco de ejecución que textualmente rodea al marco en cuestión. Un marco de ejecución A rodea *textualmente* a otro marco B cuando A corresponde a una subrutina que invocó a la subrutina B.
- El enlace dinámico en este caso es el valor previo del apuntador MP. Este valor se guarda con el objetivo de mantener consistente la estructura de la pila cuando una llamada a subrutina es finalizada y, en consecuencia, el marco de ejecución correspondiente es eliminado.
- La dirección de retorno es usada para devolver el control a la subrutina que originó la creación del marco.

Conjunto de Instrucciones de la Máquina P

El conjunto de instrucciones de la máquina P contiene instrucciones típicas para la administración de las estructuras de la máquina además de operaciones que permiten alterar el flujo de control. Del conjunto de instrucciones de la máquina P hay dos que llaman la atención: *MST* y *CUP*. Estas son las instrucciones encargadas de construir e invocar marcos de ejecución.

La ejecución de la instrucción MST permite además la creación de los enlaces estáticos y dinámicos en un marco de ejecución. Específicamente, esta instrucción actualiza el valor del enlace estático del marco con la dirección del marco que textualmente rodea al mismo. El enlace dinámico es asignado al valor actual del MP de la pila. Adicionalmente, el valor del SP se actualiza.

La instrucción CUP (*call user procedure*) actualiza el valor de la dirección de retorno y, acto seguido, ejecuta el llamado a subrutina.

A continuación, en la tabla 4.1 se destaca las instrucciones más importantes del conjunto de instrucciones de la máquina P. El total de instrucciones puede ser consultado en la referencia [5].

En la tabla, es posible ver para cada instrucción, cómo se altera la estructura principal de la máquina, es decir, la pila. Concretamente, se indica el estado previo que debe tener la pila para poder ejecutar la instrucción y el estado final de la misma después de ejecutada la instrucción. La notación *(A,B)* indica que el elemento que se encuentra en el tope de la pila es A y el que está inmediatamente debajo de A en la estructura es el elemento B. La letra x indica que el tipo de dato del elemento involucrado en la operación puede ser entero, real, booleano o caracter.

Instrucción	*Estado de la Pila*		*Descripción*
	Antes	*Después*	
ABI	entero	entero	Valor absoluto de un entero
ABR	real	real	Valor absoluto de un real
ADI	(entero,entero)	entero	Suma de dos enteros
ADR	(real,real)	real	Suma de dos reales
CHR	entero	caracter	Convierte un entero en un caracter
CSP	N/A	N/A	Llamado a procedimiento estándar (libreria)
CUP	N/A	N/A	Llamado a procedimiento definido por el usuario
DEC	x	x	Decremento
DVI	(entero,entero)	entero	División entera
DVR	(real,real)	real	División real
EQU	(x,x)	booleano	Comparación de igualdad de dos valores
FJP	booleano		Salto si la condición es falsa
GEQ	(x,x)	booleano	Comparación de mayor o igual de dos valores
INC	x	x	Incremento
LCA		dirección	Cargar dirección de una constante de la zona de memoria
LEQ	(x,x)	booleano	Comparación de menor o igual de dos valores
LES	(x,x)	booleano	Comparación de menor de dos valores
LOD		x	Cargar valor de una dirección dada
MOD	(entero,entero)	entero	Modulo
MPI	(entero,entero)	entero	Multiplicación entera
MPR	(real,real)	real	Multiplicación real
MST	N/A	N/A	Mark Stack
NEQ	(x,x)	booleano	Comparación de no igualdad de dos valores
NOT	booleano	booleano	Negación booleana
RET	N/A	N/A	Retorno de un marco
SBI	(entero,entero)	entero	Resta de dos enteros
SBR	(real,real)	real	Resta de dos reales
STP	N/A	N/A	Detención de la máquina
UJP	N/A	N/A	Salto incondicional

Tabla 4.1: *Conjunto de instrucciones de la Máquina P*

4.2.2. Lenguajes Funcionales

Los lenguajes funcionales emplean un modelo computacional basado en la definición recursiva de funciones. Este modelo está inspirado en el llamado cálculo lambda, desarrollado por Alonzo Church a principios de la década de 1930. Concretamente, la programación funcional define las salidas de un programa como una función matemática de las entradas.[27]

El pionero de los lenguajes funcionales fue LISP a principios del año 1960.[18] LISP ha sido un lenguaje ampliamente reconocido por su capacidad de manipular datos simbólicos. Únicamente a finales de la década de 1970, surgieron más lenguajes basados en el paradigma funcional. Últimamente, los lenguajes funcionales han sido utilizados no solo para aplicaciones científicas o académicas, sino comerciales también.

En un lenguaje funcional, las funciones son la estructura fundamental de cualquier programa. Un programa escrito en un lenguaje funcional consiste de la declaración de una función principal definida en términos de funciones o expresiones más simples. El principio de ejecución de los lenguajes funcionales es la reducción. Concretamente, su ejecución consiste en la evaluación de la función principal hasta que el resultado es obtenido. Una expresión particular "invoca"otras expresiones por medio de la aplicación de funciones, sin embargo, no existe una definición explicita en términos de instrucciones de flujo de control.[27]

En un programa escrito en un lenguaje puramente funcional, no hay efectos de borde, ya que el valor resultante de la evaluación de una función, únicamente afecta al contexto que la rodea, es decir, la función "padre". Por este hecho, las funciones en un lenguaje funcional son referencialmente transparentes.[22]

Miranda, y su sucesor, Haskell son enteramente libres de efectos de borde por ser puramente funcionales, en cambio, en lenguajes como Scheme (dialecto de LISP) que implementan conceptos de lenguajes procedimentales como las asignaciones e iteraciones, los efectos de borde son recurrentes.[27]

Los lenguajes funcionales proveen cierta funcionalidad para facilitar su uso, por ejemplo:

- Funciones de orden superior y funciones de primera clase.
- Polimorfismo.
- Listas y operadores sobre listas.
- Recursión.
- Retorno de tipos de datos estructurados en funciones (retornar arreglos por ejemplo).
- Recolección de basura.

Las listas son fundamentales en los lenguajes funcionales porque su definición es puramente recursiva, lo cual permite una fácil manipulación al operar sobre el primer elemento de la lista, y sobre el resto de manera recursiva. La recursión es fundamental en este tipo de lenguajes al ser el único mecanismo de implementar ejecución repetitiva. El cálculo o evaluación de funciones complejas, como operaciones sobre listas, puede requerir un alto uso de recursión.

En LISP en particular, un programa es una lista. Y puede extenderse a sí misma en tiempo de ejecución al construir una lista y ejecutarla. Esto es posible porque LISP retrasa casi toda la validación semántica hasta la etapa de ejecución.[18]

En los lenguajes funcionales, las listas pueden ser de dos tipos: homogéneas o heterogéneas. Por ejemplo, en ML, las listas son homogéneas, es decir, deben estar compuestas únicamente

de elementos de un solo tipo de dato, en cambio, en LISP, las listas son heterogéneas, lo cual implica que elementos de diferentes tipos pueden pertenecer a una misma lista, siempre y cuando la lista no quede inconsistente respecto a la correspondencia de tipos entre parámetro y definición de la función.

Los lenguajes funcionales dependen mucho del polimorfismo (debido a que, ciertos lenguajes, deben inferir los tipos de datos de los parámetros), hacen uso intensivo de listas, y de funciones de orden superior. Este tipo de funciones, exclusivas de los lenguajes funcionales, toman una función como argumento y retornan una función como resultado.

LISP y sus dialectos como Scheme son dinámicamente tipados, y por lo tanto polimórficos sintácticamente, mientras Miranda, Haskell, ML, etc., implementan el polimorfismo por medio de mecanismos de inferencia de tipos de datos. [18][13]

Detalles de Implementación

En los lenguajes funcionales, la implementación del concepto de funciones definidas en términos de funciones, el manejo recursivo de listas, etc., lleva a sugerir la presencia de una estructura de memoria dinámica en la máquina abstracta, concretamente, un *heap*.

En general, cuando se habla de la implementación de estructuras dinámicas, es decir, que pueden cambiar en tiempo de ejecución, se habla de un *heap*. En particular, los lenguajes funcionales al hacer un uso intensivo de listas, requieren un ambiente de ejecución que ofrezca una estructura en memoria que permita implementar las características intrínsecas de una lista, concretamente, el cambio de tamaño en memoria de sus componentes.

Adicionalmente, debido a que una función puede llegar a tener elementos locales a la misma que cambien su tamaño en tiempo de ejecución, el uso de un *heap* es necesario.

En los lenguajes funcionales, los elementos dinámicos son creados y almacenados en el *heap* de manera automática, sin mayor intervención del programador, ya que el lenguaje no proporciona funcionalidad para el manejo de referencias, sin embargo, en los lenguajes como LISP que no son puramente funcionales, esto sí es posible conllevando a posibles inconsistencias en la información debido a que un cierto elemento en el *heap* puede ser referenciado por más de una función.

Una ventaja de los lenguajes puramente funcionales es que las estructuras creadas ya sea en la pila, en el *heap* o en ambos, no son cíclicas. Los nuevos elementos creados referencian a los viejos, pero los viejos nunca cambian y, por lo tanto, no van a apuntar a los nuevos. Por tanto, en los lenguajes puramente funcionales, el problema de las referencias circulares para la recolección de basura es evitado.

ML y la Máquina CAM

La máquina CAM (*Categorical Abstract Machine*) es un ambiente de ejecución de lenguajes estrictamente funcionales cuya especificación se deriva de la máquina SECD.[13] Las

implementaciones de esta máquina abstracta han sido muy usadas especialmente para procesar el lenguaje ML. La máquina CAM permite la implementación de las características más importantes de un lenguaje funcional como funciones de orden superior, funciones de primera clase, polimorfismo, etc.

Como se vio en la sección anterior, en los lenguajes puramente funcionales, no existe el concepto de estructuras funcionales cíclicas. En otras palabras, toda función evaluada siempre tiene una referencia a la función que la creó, sin embargo, la referencia a la función "hija"por parte de la función "padre"no existe. Por lo tanto, la necesidad de eliminar marcos de ejecución de la pila es evitada. Esto lleva a que la pila en la máquina CAM no implemente operaciones de eliminación de valores del tope de la estructura. La pila en la máquina CAM se denomina pila de ambientes, o simplemente, *ambiente*. [3]

La estructura de la máquina CAM está compuesta básicamente de una zona de memoria conformada por las siguientes estructuras:

- Pila
- Memoria auxiliar
- Segmento de código

Conjunto de Instrucciones de la Máquina CAM

Debido a las pocas estructuras que contiene la máquina CAM, su conjunto de instrucciones es relativamente pequeño. Existen instrucciones básicas para evaluar nuevas funciones, así como para retornar valores. Adicionalmente, se encuentran las instrucciones típicas para alterar el flujo de control e instrucciones comunes sobre la pila y la máquina como PUSH y STOP.

A continuación, en la tabla 4.2 se lista el conjunto de instrucciones de la máquina CAM.

4.2.3. Lenguajes Lógicos

Los lenguajes lógicos son ampliamente usados para desarrollar especificaciones formales que involucren, por ejemplo, comprobación de teoremas. Estos lenguajes están inspirados en la lógica de predicados. Su modelo computacional es hallar valores que satisfagan ciertas relaciones especificadas usando una búsqueda a través de reglas lógicas. La programación lógica formaliza la búsqueda de valores que hacen a cierta proposición verdadera.[27]

Los lenguajes lógicos no se preocupan del tema del flujo de control. El programador simplemente especifica un conjunto de reglas de inferencia, y la implementación del lenguaje debe hallar un orden en el cual se apliquen esas reglas para lograr deducir valores que satisfagan cierta propiedad definida. En pocas palabras, los lenguajes lógicos le permiten

Instrucción	*Descripción*
ACC n	Accede al marco número n de la pila de ambientes
QUOTE	Ejecuta la operación actual del segmento de código
PUSH	Empila un marco en la pila de ambientes
SWAP	Intercambia dos marcos
RETURN	Retorna el valor de una función que ha sido evaluada
APP	Salto indirecto a la función padre del marco en el tope de la pila de ambientes
FST n	Acceso a la primera parte de un determinado marco de ejecución
SND n	Acceso a la segunda parte de un determinado marco de ejecución
REST n	Acceso al valor restante de un determinado marco de ejecución
CUR	Construye una clausura
STOP	Detiene la ejecución de la máquina
CALL	Llamado a función
GOTOFALSE	Salto si una condición determinada es falsa
SWITCH	Salto con varias opciones según una condición
GOTO	Salto incondicional

Tabla 4.2: *Conjunto de instrucciones de la Máquina CAM*

al desarrollador definir una colección de axiomas a partir de los cuales se pueden probar teoremas.[22]

La programación lógica es ampliamente usada en sistemas expertos, diseño de circuitos digitales, y, por ejemplo, en el estudio formal de la semántica de un lenguaje, en particular, la semántica axiomática.

Prolog es el lenguaje lógico más conocido. Este y otros lenguajes lógicos se basan conceptualmente en el cálculo de predicados de primer orden, o lógica de primer orden. Los lenguajes lógicos implementan el concepto de resolución, es decir, el sistema intenta descubrir, para un conjunto particular de entradas, la demostración de un teorema a partir de una colección de axiomas y reglas de inferencia. Usualmente, los lenguajes lógicos implementan el concepto de *backtracking* para desarrollar la resolución. En la mayoría de lenguajes lógicos, incluido Prolog, la notación de las reglas de inferencia es la utilizada en las Clausulas de Horn.[27]

Los lenguajes lógicos implementan los conceptos de predicado, término, resolución y unificación. Un programa lógico calcula a través de la resolución de declaraciones lógicas dirigidas por la habilidad del lenguaje de unificar variables y términos. Además, estos lenguajes implementan conceptos de lenguajes de otros paradigmas como los tipos de dato, variables, constantes, expresiones, listas y los predicados de orden superior.[27]

Detalles de Implementación

Prolog y la Máquina WAM

La máquina abstracta más conocida para implementar lenguajes lógicos es la *Warren Abstract Machine* (WAM).[15] Esta máquina ofrece instrucciones especiales para implementar los conceptos de unificación y resolución, además instrucciones de flujo de control para implementar *backtracking*.

La máquina WAM especifica cuatro estructuras principales: una pila, un *heap*, y dos zonas de memoria especiales llamadas *Trail* y *Push-down list* (PDL).

La pila contiene los ambientes de ejecución creados a partir del procesamiento del programa. En la mayoría de implementaciones de la máquina WAM, se crea un ambiente de ejecución por cada regla de inferencia presente en el programa. Un ambiente de ejecución contiene las variables que son locales a la regla de inferencia y puede tener también almacenados algunos valores de registros de propósito específico de la máquina.

El *heap* se encarga de almacenar las estructuras de datos dinámicas que implementa el lenguaje, concretamente, las listas. Adicionalmente, es posible almacenar en el *heap* variables cuyo tamaño puede cambiar en tiempo de ejecución. La mayoría de las implementaciones de la máquina WAM implementan un sistema de recolección de basura automático para los elementos en el *heap* que dejaron de ser referenciados.

En la zona de memoria llamada *trail*, la máquina lleva la referencia del árbol de resolución de un determinado teorema compuesto de las reglas de inferencia involucradas. Cuando una regla de inferencia no conduce a obtener una prueba para el teorema, entonces el programa debe intentar el mismo procedimiento con otra regla de inferencia. Es en este punto donde se utiliza un algoritmo de *backtracking* para buscar una regla de inferencia alternativa en el árbol de resolución de un teorema.

Finalmente, en la zona de memoria llamada *push-down list* se implementa el mecanismo de unificación del lenguaje lógico. Específicamente, es en esta zona donde se efectúan las operaciones lógicas entre las reglas de inferencia presentes en la zona *trail*. Una vez el árbol de resolución de la zona *trail* es calculado, se efectúan las operaciones lógicas (*and*, *or*, etc) entre las reglas de inferencia para así lograr determinar si fue posible demostrar un teorema o no.

La máquina WAM tiene dos conjuntos de registros: los principales y los auxiliares. Los registros principales permiten la administración de las estructuras de la máquina. Los registros auxiliares sirven como variables temporales para el paso de información, específicamente referencias, entre las estructuras.

El número de registros principales de la máquina WAM son cinco, a saber:

- **P** (*Program pointer*). Referencia a la instrucción actual siendo ejecutada.

- **E** (*Environment pointer*). Referencia al último ambiente ubicado en la pila.
- **B** (*Choice point pointer*). Referencia al último *choice point* generado.
- **TR** (*Top of Trail*). Referencia al tope de la zona *Trail*.
- **H** (*Top of Heap*). Referencia a la última zona ocupada del *heap*.

Conjunto de Instrucciones de la Máquina WAM

En la tabla 4.3, se listan las instrucciones más representativas de la máquina WAM. El conjunto de instrucciones completo puede ser consultado en la referencia [26].

Instrucción	*Descripción*
get_variable *dir1 dir2*	Ubica el valor de la variable de la referencia *dir1* en la referencia *dir2*
get_value *dir reg*	Ubica el valor de la variable de la referencia *dir* en el registro *reg*
get_constant *dir1 dir2*	Ubica el valor de la constante de la referencia *dir1* en la variable o registro *dir2*
get_structure *dir1 dir2*	Cambia el valor de la referencia de una estructura de *dir1* a *dir2*
get_list *dir1*	Cambia el valor de la referencia de una lista de *dir1* a *dir2*
put_variable *var dir*	Almacena el valor de la variable *var* en la referencia *dir*
put_value *val dir*	Almacena el valor literal *val* en la referencia *dir*
put_constant *con dir*	Almacena el valor de la constante *con* en la referencia *dir*
put_nil *dir*	Almacena el valor nulo *nil* en la referencia *dir*
unify_variable *dir*	Unifica el valor de la variable de la referencia *dir*
unify_value *dir*	Unifica el valor de la referencia *dir*
unify_localvalue *val dir*	Unifica el valor local *val* de la referencia *dir*
unify_constant *dir*	Unifica el valor de la constante de la referencia *dir*
unify_nil	Unificación del valor nulo
proceed	Implementa la transferencia de control entre clausulas
execute *C*	Ejecuta una clausula *C*
call *C, N*	Ejecuta una clausula *C* con el valor *N*
allocate *N*	Reserva espacio en el *heap* de tamaño N
deallocate *N*	Libera espacio en el *heap* de tamaño N

Tabla 4.3: *Conjunto de instrucciones de la Máquina WAM*

4.3. Lenguajes de Propósito Específico

Los lenguajes de propósito específico (*Domain Specific Languages* - DSLs) siempre están orientados a un dominio particular, por lo tanto, un DSL permite capturar claramente la semántica de un dominio de aplicación determinado.

Este hecho permite que el usuario de un DSL (usualmente un experto del dominio de aplicación del DSL) pueda expresar la solución a un problema particular de manera más familiar y natural. En consecuencia, el tiempo de desarrollo de una solución programada

en un DSL es más reducido que si se usara un lenguaje de propósito general (*General Purpose Language* - GPL), permitiendo que el usuario del lenguaje focalice sus esfuerzos en desarrollar la solución del problema y no en aprender un lenguaje complejo.

4.3.1. Propuesta de Spinellis

Según Spinellis [23], la metodología de desarrollo de lenguajes de propósito específico (DSLs) difiere sustancialmente de la usada en el desarrollo de lenguajes de propósito general (GPLs), debido fundamentalmente, al tipo de problemas que pretende resolver cada uno de ellos.

Al implementar un GPL, es necesario desarrollar componentes necesarios para su procesamiento como: analizador léxico, sintáctico y semántico; optimizador, generador de código, etc. Sin embargo, en la implementación de lenguajes de propósito específico es posible, en algunos casos, obviar la utilización de los analizadores léxico, sintáctico y semántico. En lugar de estos, para un DSL, es posible procesar el lenguaje de entrada mediante traducción dirigida por sintaxis o expresiones regulares.

Sin embargo, para un diseñador de un DSL en ocasiones no es claro cómo se debe desarrollar su implementación. Esta situación llevó al surgimiento de ciertas estrategias que identifican y solucionan problemas recurrentes que se presentan en el desarrollo de DSLs para agilizar su desarrollo. Estas estrategias tradicionalmente se han denominado *patrones*.[8]

Gracias a los patrones, es posible divulgar entre los desarrolladores de software (no solo de DSLs) estas estrategias que permiten una mejor gestión de la experiencia acumulada y una transferencia eficaz de estos conocimientos.

En resumen, el propósito de la propuesta de Spinellis es proveer a los desarrolladores de DSLs, un conjunto de patrones que permitan desarrollar más rápida, certera y eficazmente lenguajes de propósito específico.

Patrones de Diseño en Lenguajes de Propósito Específico

Spinellis propone diversos patrones que ha identificado como útiles en el diseño de lenguajes de propósito específico. La descripción de estos patrones le proporciona al diseñador y desarrollador de un DSL, información específica acerca de, por ejemplo, qué criterios se deben tener en cuenta al escoger un patrón determinado, las consecuencias - buenas o malas - de la elección realizada, ejemplos reales de aplicación del patrón y las alternativas de implementación conocidas para el mismo.

Concretamente, se clasifican los patrones así:

- **Creacional:** Si el patrón involucra la creación de un DSL.

- **Estructural:** Si el patrón describe la estructura de un sistema que contiene o involucra de alguna manera un DSL.
- **Comportamiento:** Si el patrón describe interacciones entre DSLs.

A continuación se describirán brevemente los patrones más pertinentes para esta investigación propuestos por Spinellis:

Piggyback

Es un patrón estructural. Este patrón sugiere que el DSL a ser desarrollado permita en su sintaxis la inclusión de código de un lenguaje ya existente. Esto con el objetivo de dar soporte a elementos complejos de desarrollar desde ceros como: Manejo de excepciones, entrada-salida, recursión, etc. El patrón *piggyback* puede ser usado cuando el DSL desee ofrecer cierta funcionalidad implementada en un lenguaje ya existente, así se evita "reinventar la rueda".

Extensión del Lenguaje

Es un patrón creacional. Es usado para añadir nuevas características y funcionalidades a un lenguaje ya existente. Este patrón se diferencia del patrón *piggyback* en los roles que juegan los dos lenguajes involucrados en cada patrón: El patrón *piggyback* usa un lenguaje ya existente como modo de implementar un nuevo DSL, mientras que el patrón extensión del lenguaje es usado cuando, para crear un DSL, se extiende un lenguaje ya existente de tal forma que este conserve su sintaxis y semántica original.

La tarea de diseñar un DSL usando este patrón involucra la adición de nuevos elementos a un lenguaje ya existente, usualmente otro DSL. Estos elementos pueden incluir nuevos tipos de datos, elementos semánticos, azúcar sintáctico, etc. Usualmente, el DSL hereda toda la sintaxis y la semántica del lenguaje base, mientras que le adiciona sus propias extensiones.

Las implementaciones compiladas de este patrón, normalmente usan un preprocesador que transforma el DSL al lenguaje base.

Especialización del Lenguaje

Es un patrón creacional. Remueve características de un lenguaje base ya existente para crear un DSL. En ciertos casos, un DSL puede ser diseñado e implementado como un subconjunto de un lenguaje ya existente, usualmente un GPL. El diseño del DSL involucra la remoción de características sintácticas y semánticas del lenguaje original. Debido a que ciertas características del lenguaje base pueden ser inútiles para una aplicación dada, el diseño de un DSL siguiendo este patrón produce un lenguaje maduro que satisface los requerimientos planteados.

Transformación Fuente a Fuente

Es un patrón creacional. Permite la implementación eficiente de traductores de DSLs. Cuando el DSL no puede ser diseñado usando extensión del lenguaje, especialización del lenguaje, o *piggyback*, es posible usar las facilidades proporcionadas por las herramientas de un lenguaje ya existente usando la técnica de transformación fuente a fuente. El código fuente DSL es transformado a código fuente de un lenguaje existente usando un proceso de traducción cuyo nivel de complejidad puede variar de acuerdo a los lenguajes involucrados.

Cuando se usa este patrón, el lenguaje DSL puede hacer uso de la “infraestructura” de procesamiento de lenguajes que proporciona el lenguaje base. Esta puede incluir compiladores optimizados, depuradores, etc.

El uso de este patrón permite además hacer un seguimiento más fácil al proceso de compilación, ya que el código generado usualmente es más fácil de leer y entender por parte del programador. Además, este patrón puede ser implementado usando las técnicas tradicionales de análisis léxico, sintáctico, etc.

Representación en Estructura de Datos

Es un patrón creacional. Permite la especificación declarativa y relacionada a un dominio específico de ciertos datos complejos. Usualmente, algunas estructuras complicadas de todo tipo son más fácilmente representables usando un lenguaje en lugar de usar su representación subyacente. Diseñar un DSL para representar los datos es una atractiva solución al problema.

El DSL usualmente define una representación amigable, alternativa, pero equivalente a la estructura original de los datos representados. El compilador DSL puede entonces analizar sintácticamente la representación alternativa de los datos y convertirla en la representación original de los mismos.

La utilización de este patrón minimiza las posibilidades de inicializar estructuras de datos con datos errados o inconsistentes, ya que el compilador DSL puede efectuar una revisión sobre los datos al momento de efectuar la traducción del lenguaje original a la representación interna de los mismos.

CAPÍTULO 5

Propuesta de Solución

Una vez estudiadas las características de los paradigmas de programación más reconocidos y de sus requerimientos respecto a ejecución en máquinas abstractas, es posible proponer un lenguaje de propósito específico (DSL) que permita desarrollar una especificación de una máquina abstracta. Una vez diseñado, este lenguaje debe ser implementado con el objetivo de generar una máquina virtual a partir de una especificación, es decir, una implementación ejecutable de una máquina abstracta. Consecuentemente, podría pensarse en proveer un ambiente de desarrollo que le permita a un diseñador de máquinas abstractas desarrollar la especificación de una máquina particular y generar su implementación, todo en un mismo lugar.

Como se puede ver, la propuesta de solución puede ser desglosada en cuatro partes, a saber:

1. Diseñar un lenguaje de propósito específico que permita modelar las características comunes, en términos de estructura y de instrucciones, de las máquinas abstractas usadas para procesar lenguajes de diferentes paradigmas de programación.

2. Desarrollar un esquema de implementación para el lenguaje de máquinas abstractas que permita generar una máquina virtual.

3. Proporcionar un ambiente de desarrollo y ejecución de máquinas abstractas que permita al desarrollador modelar una máquina abstracta por medio del DSL propuesto y, adicionalmente, generar su implementación.

A continuación, se describirá en detalle cada uno de los componentes de la propuesta de solución.

5.1. Diseño del Lenguaje de Propósito Específico AMDL

Un lenguaje de propósito específico (DSL)[7] es un lenguaje de programación, usualmente de alto nivel, que permite la resolución de problemas de un dominio particular. Para esta investigación, se encontró conveniente desarrollar un DSL ya que permite especificar detalladamente una máquina abstracta reduciendo su complejidad subyacente y, además, evita la tarea de desarrollar la implementación desde cero. Este lenguaje de propósito específico se denominará AMDL por su sigla en inglés: *Abstract Machine Definition Language*.

La primera tarea a ejecutar cuando se decide implementar un lenguaje de programación es diseñar su sintaxis por medio de una gramática. Una gramática describe de manera natural la estructura jerárquica de las instrucciones de un lenguaje. El conjunto de cadenas que pueden ser derivadas empezando por el símbolo inicial, forman el lenguaje definido por la gramática. Para esta investigación, se desarrolló la sintaxis del lenguaje por medio de una gramática independiente del contexto usando notación BNF Extendida o EBNF (*Extended Backus-Naur Form*)[9].

Retomando la propuesta de Spinellis[23], es necesario identificar el o los patrones de desarrollo de DSLs que se deben aplicar según las características del DSL a desarrollar. Concretamente, se ha identificado en esta investigación que una máquina abstracta está compuesta fundamentalmente de elementos que pueden ser representados en estructuras de datos: pilas, *heaps*, memorias, etc. Adicionalmente, en virtud de la necesidad de brindar una implementación funcional de una máquina abstracta, se decidió aplicar el patrón de *Transformación fuente a fuente* de AMDL al lenguaje de programación Java[10]. Esto permitirá desarrollar máquinas virtuales cuya implementación sea familiar a gran parte de los desarrolladores gracias al uso masificado de Java.

La sintaxis del lenguaje AMDL permite la especificación de una máquina abstracta desde cero, o permite extender la definición de una máquina abstracta ya especificada por medio de la palabra clave *extends*. Adicionalmente, el *framework* LAMBDA proporciona la definición de instrucciones típicas para una máquina abstracta como instrucciones de flujo de control, instrucciones de administración de estructuras como la pila, etc. El diseñador de máquinas abstractas puede redefinir estas instrucciones por medio de la palabra clave *implements*.

El lenguaje AMDL permite especificar máquinas con una o más de las estructuras típicas identificadas en el estado del arte: pilas, *heaps*, memorias, registros, etc. Adicionalmente, por medio de AMDL es posible definir las instrucciones que se consideren necesarias para implementar cualquier lenguaje sobre la máquina abstracta, es decir, el diseñador de máquinas tiene la posibilidad de definir el conjunto de instrucciones de su máquina. Para la definición de instrucciones, el lenguaje proporciona instrucciones auxiliares como condicionales, ciclos, etc., que permiten un fácil desarrollo de las instrucciones requeridas.

A continuación se expone la sintaxis en formato EBNF del lenguaje AMDL. Ver figura 5.1.

```
AbstractMachine          ::= Header Structures Instructions

Header                   ::= define abstractMachine Identifier
                             ( extends Identifier )?
                             ( implements Identifier )?

Structures               ::= Structure +

Instructions             ::= Instruction *

Structure                ::= ( StackDeclaration
                             | HeapDeclaration
                             | RegisterDeclaration
                             | MemoryDeclaration
                             | CodeSegmentDeclaration
                             ) ;

StackDeclaration         ::= stack Identifier (Size)?

HeapDeclaration          ::= heap Identifier (Size)?

RegisterDeclaration      ::= register Identifier ( , Identifier )*

MemoryDeclaration        ::= memory Identifier ( size Size )? ( blockSize Size )?

CodeSegmentDeclaration   ::= codeSegment Identifier ( Size )?

Size                     ::= [ Expression ]

Instruction              ::= instruction Identifier ( FormalParameterList ? )
                             { InstructionBlock }

FormalParameterList      ::= FormalParameter ( , FormalParameter )*

FormalParameter          ::= Declaration

InstructionBlock         ::= ( Statement
                             | ConditionalConstruct
                             | IterativeConstruct
                             )*

ConditionalConstruct     ::= if ( Condition ) ( Statement | { InstructionBlock } )
                             ( ElseIfBlock )* ( ElseBlock )?

ElseIfBlock              ::= else if ( Condition )
                             ( Statement | { InstructionBlock } )

ElseBlock                ::= else ( Statement | { InstructionBlock } )
```

Figura 5.1: *Gramática BNF de AMDL. Parte 1*

```
IterativeConstruct      ::= WhileLoop | ForLoop

WhileLoop               ::= while ( Condition )
                            ( Statement | { InstructionBlock } )

ForLoop                 ::= for ( Initialization ; Condition ; Increment)
                            ( Statement | { InstructionBlock } )

Increment               ::= (Identifier = Identifier ArithmeticOperator Value)
                            | (Identifier AssignmentOperator (Value|Identifier))
                            | (Identifier PostfixPrefixOperator)
                            | (PostfixPrefixOperator Identifier)

Condition               ::= Expression

Statement               ::= Declaration | Assignment | InstructionInvocation ;

InstructionInvocation   ::= invoke Identifier ( ActualParameterList ? )

ActualParameterList     ::= ActualParameter ( , ActualParameter )*

ActualParameter         ::= Value | Identifier

Initialization          ::= Assignment

Assignment              ::= LeftHandSide = RightHandSide

LeftHandSide            ::= Declaration | Identifier | (Identifier Size)

RightHandSide           ::= (Identifier Size) | Expression

Expression              ::= OrExpression

OrExpression            ::= AndExpression ( || AndExpression )*

AndExpression           ::= EqualityExpression ( && EqualityExpression )*

EqualityExpression      ::= RelationalExpression
                            ( ( == | != ) RelationalExpression )*

RelationalExpression    ::= AdditiveExpression
                            (RelationalOperator AdditiveExpression )*

AdditiveExpression      ::= MultiplicativeExpression
                            ( ( + | - ) MultiplicativeExpression )*

MultiplicativeExpression::= UnaryExpression ( ( * | / |% ) UnaryExpression )*

UnaryExpression         ::= ( - )? ( Value | identifier ) ( ++ | -- )?
                            | ( Expression)
```

Figura 5.2: *Gramática BNF de AMDL. Parte 2*

```
Declaration            ::= Type Identifier

Value                  ::= Integer | Boolean | Char | Float

Type                   ::= int | boolean | char | float

RelationalOperator     ::= < | > | <= | >=

ArithmeticOperator     ::= + | - | * | / | %

PostfixPrefixOperator  ::= ++ | --

AssignmentOperator     ::= += | -= | *= | /= | %=

Identifier             ::= (a..z | A..Z )(a..z |A..Z | _ | 0..9 )* ;

Integer                ::= ( 1..9 )( 0..9 )*

Boolean                ::= true | false

Char                   ::= (a..z) | (A..Z) | (0..9) | ...

Float                  ::= ( - )? ( 0..9 )+ . ( 0..9 )+
```

Figura 5.3: *Gramática BNF de AMDL. Parte 3*

5.2. Esquema de Implementación de AMDL

Como se mencionó anteriormente, para la implementación de AMDL se aplicará el patrón *Transformación fuente a fuente*. Los lenguajes involucrados son AMDL como origen, y Java como destino. Concretamente, la implementación se desarrollará siguiendo un esquema de compilación denominado traducción dirigida por sintaxis[1]. Este esquema de compilación detalla cómo se debe realizar la traducción de un lenguaje de programación a otro teniendo en cuenta la estructura de las instrucciones del lenguaje origen. En el caso de AMDL, se traducirán las especificaciones de máquinas escritas en AMDL a una representación en código Java que implementa la máquina abstracta modelada (ver figura 5.4).

La implementación de la máquina abstracta extiende el modelo de implementación de una máquina abstracta desarrollado en Java por la profesora Silvia Takahashi como material para sus cursos de Teoría de Lenguajes y Diseño de Lenguajes en la Universidad de los Andes (ver figura 5.5). Este modelo incluye las estructuras básicas de una máquina abstracta a excepción del *heap* y los registros que se implementaron durante el desarrollo de esta investigación. En particular, esta implementación modela pilas de ambientes para implementar lenguajes imperativos. Además tiene integrado el visualizador de las estructuras de la máquina en Java Swing para poder ejecutar los programas paso a paso. Usando este

Figura 5.4: *Esquema de traducción dirigida por sintaxis de AMDL*

framework se implementaron a mano máquinas para un mini Pascal, un mini LISP y un lenguaje para manejar un robot estilo Karel [21] o Logo [12]. La interfaz lógica proporcionada es especialmente útil para la el lenguaje del robot en mención.

Para desarrollar el proceso de traducción dirigida por sintaxis, es necesario haber desarrollado la gramática independiente del contexto correspondiente al lenguaje de programación. Una vez se ha desarrollado esta gramática, es posible empezar a diseñar e implementar el proceso de traducción.

El proceso de traducción dirigida por sintaxis, se realiza en dos etapas, a saber:

1. **Procesamiento sintáctico o *parsing***. Esta labor consiste en procesar cierto programa de entrada y determinar si ese programa puede ser derivado completamente desde el símbolo inicial de la gramática. Si esto no es posible, el programa de entrada no es aceptado por la gramática.

2. **Generación de código**. Una vez el programa de entrada sea aceptado por la gramática, se procede a generar el código correspondiente. Esta labor consiste en asociar a cada producción de la gramática ciertos fragmentos de código que corresponden a la representación del programa de entrada en el lenguaje destino.

En la sección siguiente, se detallará el proceso de implementación del traductor dirigido por sintaxis para el lenguaje AMDL.

5.3. Ambiente de Desarrollo de Máquinas Abstractas

Actualmente, un programador de cualquier lenguaje tiene a su disposición gran cantidad de ambientes de desarrollo que le brindan herramientas de apoyo a su labor. Estas herramientas son fundamentales para su comodidad y productividad.

En general, un ambiente de desarrollo en general está compuesto por:

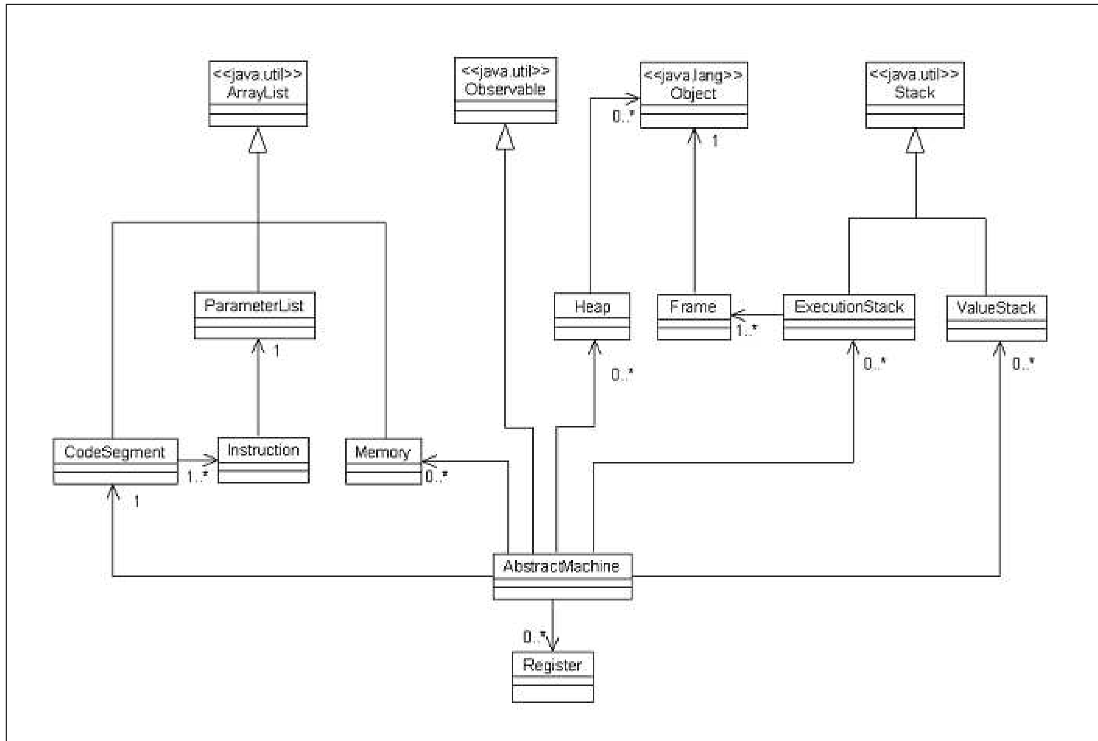

Figura 5.5: *Implementación en Java de una máquina abstracta desarrollada por Takahashi, S. [25]*

- Un editor de código fuente
- Un compilador o intérprete para el código.
- Un depurador
- Etc.

Así como un desarrollador de Java, C, Visual Basic, etc., dispone de un ambiente de desarrollo integrado que apoya su labor, un desarrollador de AMDL debería tener acceso a un ambiente de desarrollo de similares características.

El ambiente de desarrollo de AMDL debería permitirle al diseñador de máquinas abstractas desarrollar su especificación en el lenguaje AMDL, generar la máquina virtual correspondiente, y poder ejecutar la máquina virtual generada para procesar el lenguaje de su preferencia.

Sin embargo, construir un ambiente de desarrollo desde cero es una labor bastante ardua. El desarrollo de un editor, de un depurador, entre otros, puede requerir bastante esfuerzo de desarrollo que no está orientado a los objetivos principales de esta investigación.

Por tal motivo, para superar esta dificultad, se ha escogido extender un ambiente de desarrollo ya existente. El ambiente de desarrollo escogido ha sido ECLIPSE [17].

5.3.1. ECLIPSE

ECLIPSE[17] es un ambiente de desarrollo flexible multilenguaje y multiplataforma. Es uno de los ambientes de desarrollo más populares para el lenguaje Java, sin embargo, ECLIPSE puede ser usado para desarrollar software en diversos lenguajes, no solo Java.

ECLIPSE ofrece unas prestaciones muy interesantes al desarrollador de software, entre las más importantes están:

- **Editores enriquecidos**. Los editores de múltiples lenguajes disponibles en ECLIPSE tienen integrada la funcionalidad de proveer asistencia al desarrollador en caliente. Es decir, el editor es capaz de indicar información acerca de errores sintácticos, sugerencia de variables y métodos, etc., mientras se desarrolla la labor de codificación.
- **Múltiples perspectivas**. En ECLIPSE, existe el concepto de perspectiva. Una perspectiva es un conjunto de editores, barras de herramientas, etc., que proporcionan una funcionalidad orientada a una tarea determinada. Por ejemplo, existe la perspectiva Java, la perspectiva C++, la perspectiva de depuración de código, etc.
- Etc.

En sus orígenes, ECLIPSE era un software propietario, sin embargo, en 2001 se convirtió en una plataforma de código abierto. Este hecho permitió que ECLIPSE pueda ser complementado en funcionalidad por medio de extensiones llamadas *plug-ins*. En ECLIPSE, todo es un *plug-in* a excepción de un kernel reducido. ECLIPSE modela el concepto de *punto de extensión* y *extensión*. Un punto de extensión permite asociar a la plataforma una extensión, es decir, un *plug-in*. Ver figura 5.6

5.4. LAMBDA: Framework de Máquinas Abstractas

El *framework* LAMBDA debe, en un futuro, no solo limitarse a la especificación y generación de máquinas abstractas y el procesamiento básico de lenguajes en máquinas virtuales. Las máquinas virtuales modernas han manejado problemáticas que, en ambientes reales, pueden presentarse al decidir implementar un lenguaje con base en una máquina abstracta.

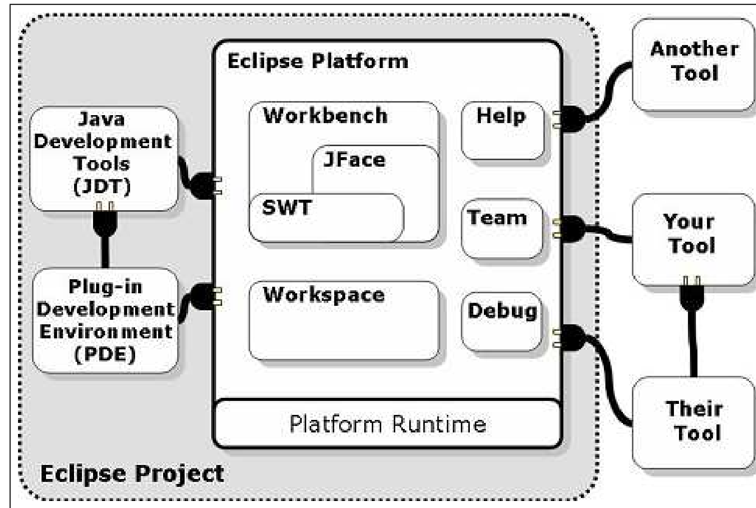

Figura 5.6: *Arquitectura de plug-ins de ECLIPSE [17]*

A continuación se enumeran los aspectos a implementar a futuro en el *framework* LAMBDA.

Seguridad/Verificación

Una de las inquietudes que surgen a partir de la aproximación de máquinas abstractas, y en general, de las diversas aproximaciones para procesar lenguajes (compilación, interpretación, etc) es cómo garantizar que no se pierde semántica al traducir un lenguaje fuente al instruction set de la máquina abstracta.

Trabajos previos como el efectuado por Stark et al. [24] sobre la JVM, proponen el desarrollo de un módulo verificador de *bytecode* cuya labor es garantizar, bajo ciertas condiciones, que determinado *bytecode* refleja completamente la semántica del código Java original. Para ello, formalizan de manera no ambigua, las especificaciones del lenguaje Java y de su máquina virtual, abarcando incluso futuras extensiones al lenguaje y a la JVM. Para lograr su objetivo, los autores formulan los modelos de la máquina virtual y de sus estructuras usando la notación de Máquinas de Estados Abstractas (*Abstract State Machines*) [24]. Esta notación es una especie de seudocódigo que permite trabajar sobre estructuras abstractas usando modelos matemáticos. Esto permite formalizar los componentes estructurales de una máquina, además de los conceptos del lenguaje como los objetos, métodos, clases, etc.

Para lograr implementar un esquema efectivo de verificación, es indispensable formalizar el lenguaje, la máquina virtual y el esquema de ejecución de un código generado sobre la máquina virtual. A partir de esta formalización, es posible verificar si cierto programa generado, que esté sintácticamente bien formado, no cumple con el modelo formal definido. Esta labor de verificación se hace teniendo en cuenta criterios como la verificación de tipos

de dato, la verificación de la compilación a partir de una precondición y una postcondición, etc.

En el caso de Java, una vez el programa es compilado a su representación correspondiente en *bytecode*, es un deber de la máquina virtual hacer una verificación de ese código antes de ser ejecutado. Para ello, es necesario hacer verificaciones relacionadas con seguridad de tipos de dato (operaciones no permitidas entre ciertos tipos de dato), seguridad de flujo de control (saltos definidos siempre a direcciones conocidas), seguridad de accesos a memoria (objetos con acceso únicamente a su memoria local), etc.

Otro problema relacionado con lo anterior es la seguridad del código ejecutado en una máquina virtual. Al tener la máquina virtual acceso casi total a los recursos de un sistema operativo, puede pasar que el código a ser ejecutado esté contaminado por código maligno. Este riesgo es latente en ambientes de programación orientados a Internet como Java. En toda máquina virtual orientada a un ambiente distribuido, se debe implementar mecanismos de verificación de código generado para garantizar que provenga del compilador sin modificación alguna. Adicionalmente, debe evitarse a toda costa la ejecución de código generado por una fuente desconocida. [10]

Just-in-time Compiling

Uno de los problemas de la interpretación es la falta de desempeño que se presenta al procesar un programa. La aproximación más usada en las máquinas virtuales para superar este problema es la compilación *just-in-time*.[1][16] Este esquema de compilación consiste en traducir a código de máquina nativo de la plataforma de *hardware* ciertas instrucciones del programa de entrada, y solamente dejar intactas las instrucciones relacionadas con operaciones de entrada-salida. De esta manera, el procesamiento que se hace en la máquina virtual es mínimo, así mejorando el desempeño y conservando la portabilidad del código.

Reflexión en Máquinas Virtuales

Una máquina virtual implementa el concepto de reflexión cuando tiene un mecanismo que le permite modificar su implementación en tiempo de ejecución. Esto tiene ventajas como: actualización y reconfiguración dinámica de la máquina virtual sin necesidad de detener su ejecución, adaptación estructural de la máquina virtual según el programa de entrada, etc. Para lograr estos objetivos, se debe diseñar e implementar un modelo genérico de una máquina virtual que sea capaz de modificar su configuración en tiempo de ejecución.[4]

Interoperabilidad entre Máquinas Virtuales

El problema de la interoperabilidad está presente siempre que se diseña un software orientado a ambientes distribuidos. Es posible pensar en desplegar máquinas virtuales situadas en diferentes nodos de una red, y lograr ejecutar un programa en la máquina virtual de manera distribuida, logrando un entorno de computación ubicuo. Sin embargo, para implementar la comunicación efectiva y el intercambio de información entre las máquinas, se

debe enfocar esfuerzos en definir claramente los protocolos de comunicación y el formato de los datos que se van a intercambiar entre las máquinas virtuales. Estos protocolos y formatos deben ser comunes para lograr una comunicación exitosa.[14]

Máquinas Virtuales de Tiempo Real

Cuando se desea implementar una máquina virtual en un sistema en tiempo real, los requerimientos exigidos a la máquina virtual cambian considerablemente. Fundamentalmente, el tiempo requerido para ejecutar un determinado programa debe ser bastante reducido. Una de las aproximaciones más usadas para incrementar el desempeño es implementar mecanismos de concurrencia en el procesamiento de un programa de entrada. Así mismo, la implementación de un conjunto de instrucciones de más bajo nivel asegura una disminución en el tiempo de ejecución de cada instrucción, incrementando el desempeño general. Estas aproximaciones tienen como objetivo no afectar el desempeño en la comunicación entre procesos, la cual es fundamental en esta clase de sistemas.[4]

Optimización de Código

En general, al diseñar un compilador, la principal preocupación es lograr traducir efectivamente un programa fuente a un programa destino. Sin embargo, puede que al traducir las instrucciones de un lenguaje de alto nivel a uno de más bajo nivel, como en el caso del conjunto de instrucciones de una máquina virtual, el programa generado incluya una gran cantidad de instrucciones susceptibles de ser remplazadas o eliminadas con el objetivo de ganar desempeño en tiempo de ejecución. Uno de los métodos más usados para lograr optimizar cierto bloque de código es el análisis de flujo de datos (*data-flow analysis*). Este análisis se basa en un conjunto de algoritmos cuya tarea es encontrar patrones en ciertos grupos de instrucciones que sean susceptibles de ser optimizadas. Un grupo de instrucciones puede ser optimizado cuando, por ejemplo, existen operaciones redundantes como exceso de asignaciones, exceso de accesos a estructuras de datos, etc.[11]

CAPÍTULO 6

Implementación de la Solución

Como se mencionó en la sección anterior, la implementación de la propuesta de solución consiste en desarrollar un mecanismo de procesamiento del lenguaje AMDL y, a continuación, integrarlo en un ambiente de desarrollo basado en ECLIPSE. A continuación, se desarrollará cada una de estas ideas.

6.1. Traductor Dirigido por Sintaxis

Para efectuar la labor del procesamiento del lenguaje AMDL, se propuso desarrollar un traductor dirigido por sintaxis[1]. Este traductor está organizado alrededor de la sintaxis del lenguaje, concretamente, de la gramática. La traducción dirigida por sintaxis consiste en adjuntar a cada producción de la gramática fragmentos de código usando un esquema de traducción.

La implementación de este traductor se desarrolló usando el generador de compiladores ANTLR [20]. Este software permite implementar compiladores e intérpretes generados a partir de la especificación de una gramática particular.

Concretamente, para el desarrollo de traductores dirigidos por sintaxis, ANTLR permite especificar en un archivo de entrada la gramática y, conjuntamente, el código asociado a cada producción de la gramática que va a ser generado. Este código a ser generado representa cada una de las estructuras de AMDL en el lenguaje destino, es decir, Java.

La especificación de la gramática y la traducción se desarrollan en un archivo con extensión **.g**. Ver figura 6.1.

```
ruleHeader
:
( ('define')('abstractMachine')(RULE_ID)
  {$am = factory.createAbstractMachine($RULE_ID.text);}
  (('extends') (RULE_ID))?
  {am.setExtends($RULE_ID.text);}
  (('implements') (RULE_ID))?
  {am.setImplements($RULE_ID.text);}
)
;
```

Figura 6.1: *Implementación de la traducción dirigida por sintaxis de AMDL en ANTLR*

Una vez el archivo es procesado por ANTLR, se genera una implementación del traductor en el lenguaje Java. Los archivos generados son los siguientes:

- Lexer.java
- Parser.java
- Tokens.java

Por medio de esta implementación, ya es posible generar archivos Java a partir de una especificación de una máquina abstracta escrita en lenguaje AMDL. Ver figura 6.2.

Uno de los objetivos de este trabajo es integrar este generador de máquinas abstractas a un ambiente de desarrollo, específicamente ECLIPSE. Para ello, se integrará un *plug-in* al ambiente de desarrollo para que el usuario pueda desarrollar su máquina abstracta con la comodidad que proporciona ECLIPSE.

Este hecho cambia ligeramente el esquema de implementación del lenguaje. Por las características de los puntos de extensión de ECLIPSE, fue difícil integrar ANTLR a la arquitectura de ECLIPSE. En la actualidad, existe un *plug-in* de ANTLR para ECLIPSE, sin embargo, no se encuentra actualizado a la última versión que fue la usada para la implementación de AMDL.

Por tal motivo, se buscó otra aproximación al problema de integrar ANTLR a ECLIPSE. Finalmente, se encontró un software que permitía realizar esta integración: *OpenArchitectureWare* [19].

OpenArchitectureWare es un *framework* que, entre otras funcionalidades, incorpora herramientas para desarrollar y procesar lenguajes de propósito específico siguiendo los principios de MDA (Model Driven Architecture). En particular, *OpenArchitectureWare* proporciona una herramienta llamada Xtext que permite generar un metamodelo a partir de la gramática de un lenguaje. Adicionalmente, permite generar un parser ANTLR integrado a un *plug-in* de ECLIPSE.

Gracias a este *framework*, fue posible integrar la implementación del traductor desarrollado en ANTLR a la plataforma ECLIPSE.

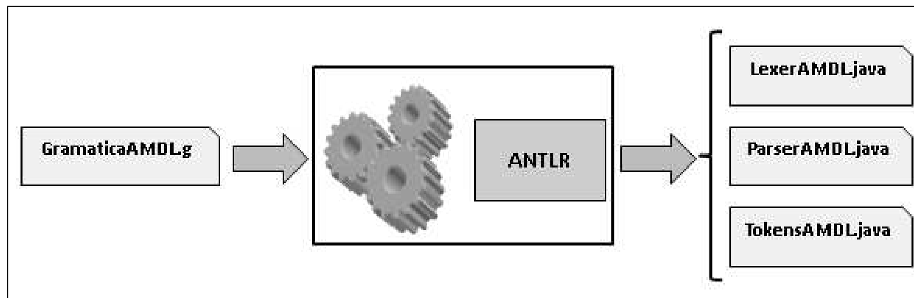

Figura 6.2: *Generación del traductor en ANTLR*

6.2. Ambiente de Desarrollo de Máquinas Abstractas en ECLIPSE

A partir del traductor del lenguaje AMDL integrado en ECLIPSE, el paso a seguir era desarrollar un ambiente de desarrollo de máquinas abstractas en ECLIPSE que permitiera especificar, generar, y ejecutar una máquina con todas las facilidades que proporciona ECLIPSE como ambiente de desarrollo.

Para esto, se desarrolló un *plug-in* para ECLIPSE que permitiera al desarrollador tener un ambiente de desarrollo específico de AMDL. Ver figura 6.3.

Este ambiente de desarrollo se diseñó para que implementara funcionalidades propias de ECLIPSE como coloreado de sintaxis, navegabilidad en el código, detección de errores en caliente, etc. Ver figura 6.4.

Una vez el desarrollador de AMDL haya desarrollado la especificación de su máquina en el editor de AMDL (ver figura 6.5), puede generar la implementación de la misma por medio del traductor dirigido por sintaxis desarrollado en ANTLR e integrado en ECLIPSE. La implementación generada será ubicada en el mismo espacio de trabajo del usuario en ECLIPSE, de tal manera que puede hacer uso de su máquina virtual al instante. Ver figura 6.6.

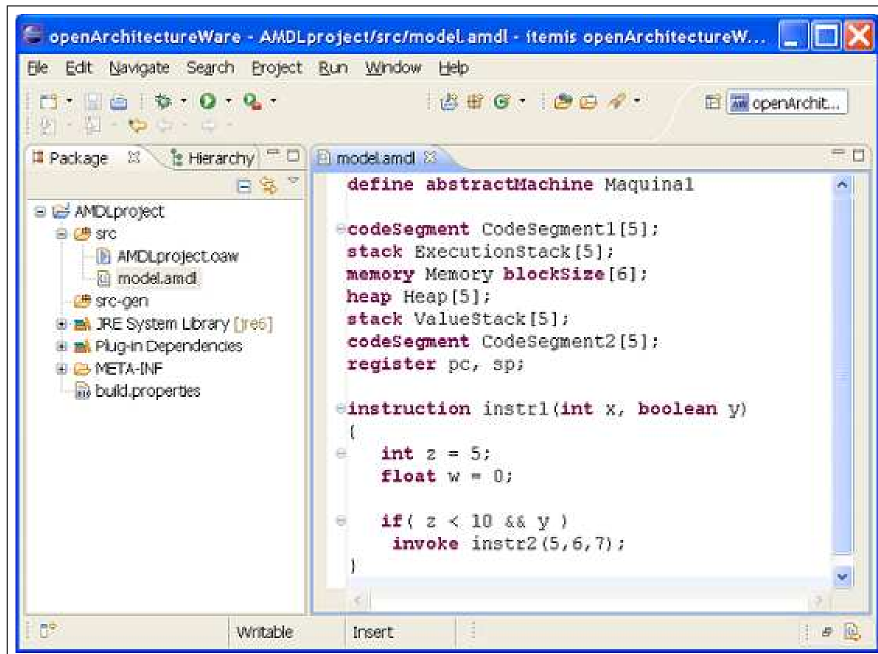

Figura 6.3: *Ambiente de desarrollo de AMDL en ECLIPSE*

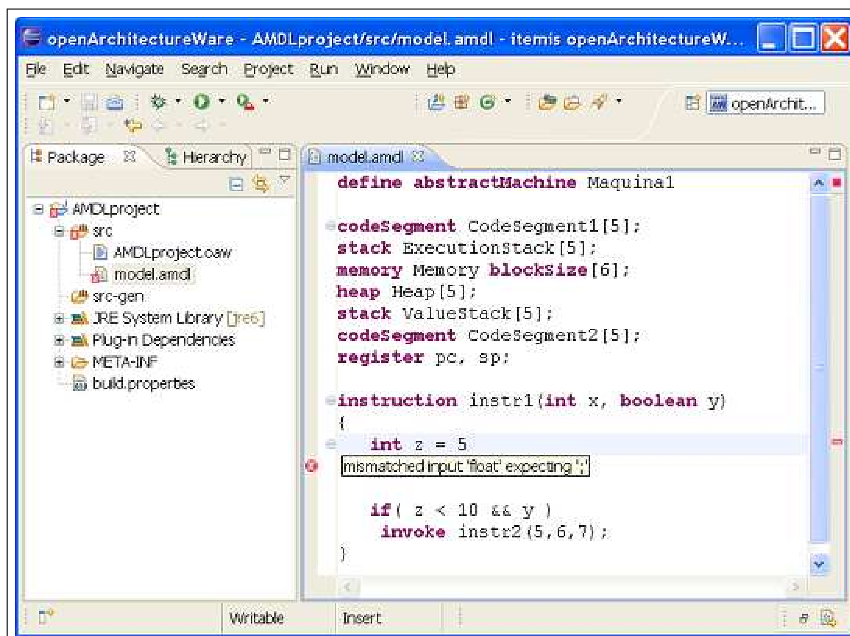

Figura 6.4: *Funcionalidades del ambiente de desarrollo de AMDL*

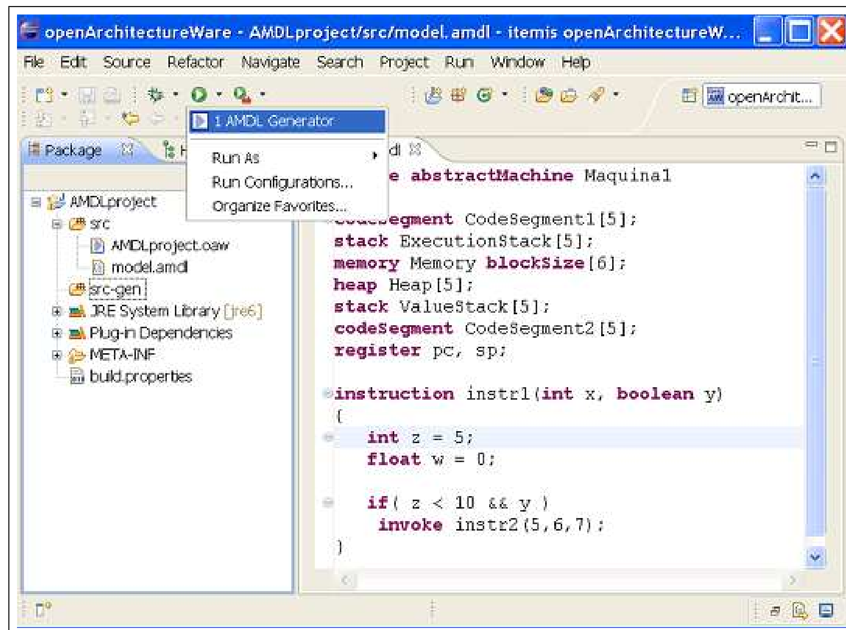

Figura 6.5: *Generación de la implementación de la máquina abstracta en AMDL*

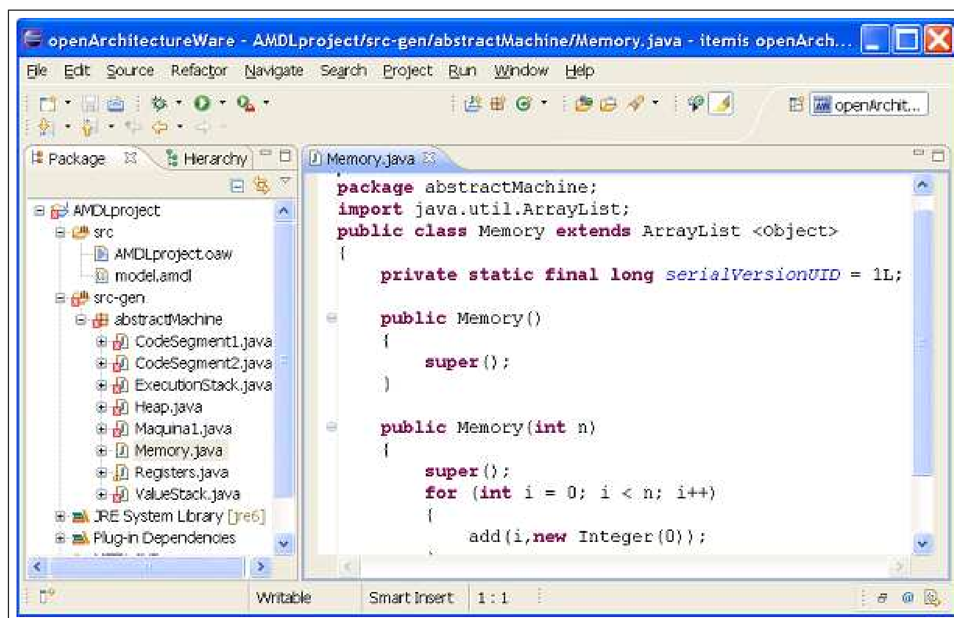

Figura 6.6: *Máquina virtual generada a partir de una especificación en AMDL*

CAPÍTULO 7

Casos de Estudio

A continuación, se ilustrará cómo el lenguaje AMDL permite modelar las características estructurales y la definición de instrucciones de una máquina abstracta.

Como caso de estudio, se usarán las tres máquinas referenciadas en el estado del arte pertenecientes a los paradigmas de programación imperativo, funcional y lógico.

7.1. Lenguajes Imperativos: Máquina P

Recapitulando, la máquina P estructuralmente está compuesta de una zona de memoria y de un conjunto de registros. La zona de memoria o *data store* está compuesta de las siguientes estructuras:

- Pila
- *Heap*
- Zona de constantes
- Segmento de código

El conjunto de registros está compuesto de la siguiente manera:

- **PC** (*Program counter*). Referencia a la instrucción actual siendo ejecutada.
- **SP** (*Stack pointer*). Referencia al tope de la pila.

- **MP** (*Mark stack pointer*). Referencia a la base del marco que se encuentra en el tope de la pila.
- **NP** (*New pointer*). Referencia al inicio del espacio libre hacia donde puede crecer el *heap*.

El conjunto de instrucciones de la máquina P está compuesto de 60 instrucciones, de las cuales, se identificaron las más importantes en la tabla 4.1. Como se mencionó anteriormente, dos de las instrucciones más importantes de la máquina P son MST y CUP. Estas dos instrucciones permiten crear e invocar marcos de ejecución.

En la figura 7.1 se ilustra cómo el lenguaje AMDL permite especificar la máquina P, incluyendo estructuras e instrucciones.

7.2. Lenguajes Funcionales: Máquina CAM

La estructura de la máquina CAM se compone de las siguientes estructuras:

- Pila
- Memoria auxiliar
- Segmento de código

Adicionalmente, la máquina CAM contiene los dos registros necesarios para administrar la pila y el segmento de código: los registros SP y PC respectivamente.

El conjunto de instrucciones de la máquina CAM se identifica en la tabla 4.2.

En la figura 7.2 se ilustra cómo el lenguaje AMDL permite especificar la máquina CAM, incluyendo estructuras e instrucciones.

7.3. Lenguajes Lógicos: Máquina WAM

La estructura de la máquina WAM está compuesta de cuatro estructuras principales:

- Pila
- *Heap*
- Zona de memoria *Trail*
- Zona de memoria *Push-down list*

Respecto a los registros, la máquina WAM tiene cinco registros principales, a saber:

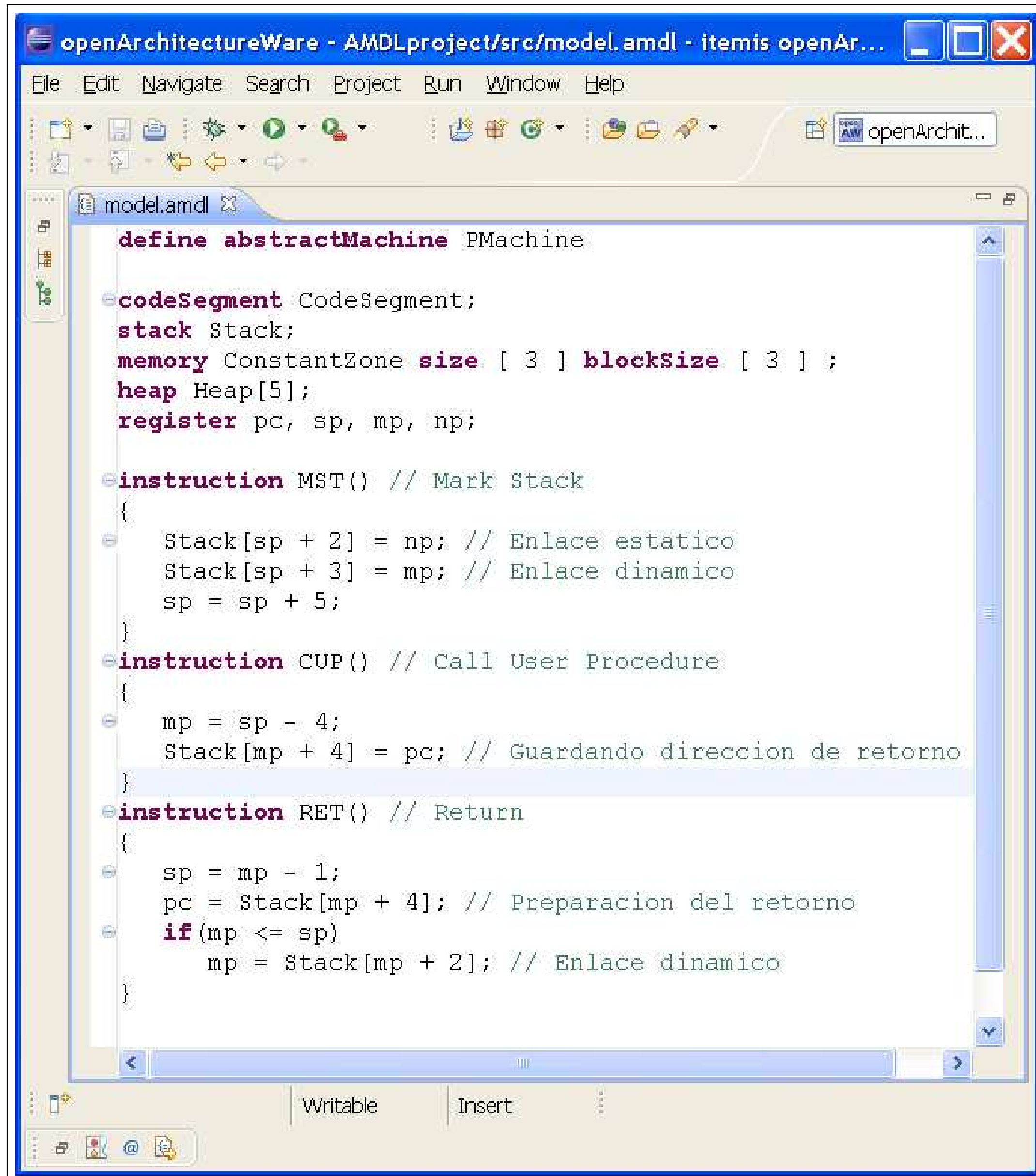

Figura 7.1: *Especificación en AMDL de la Máquina P*

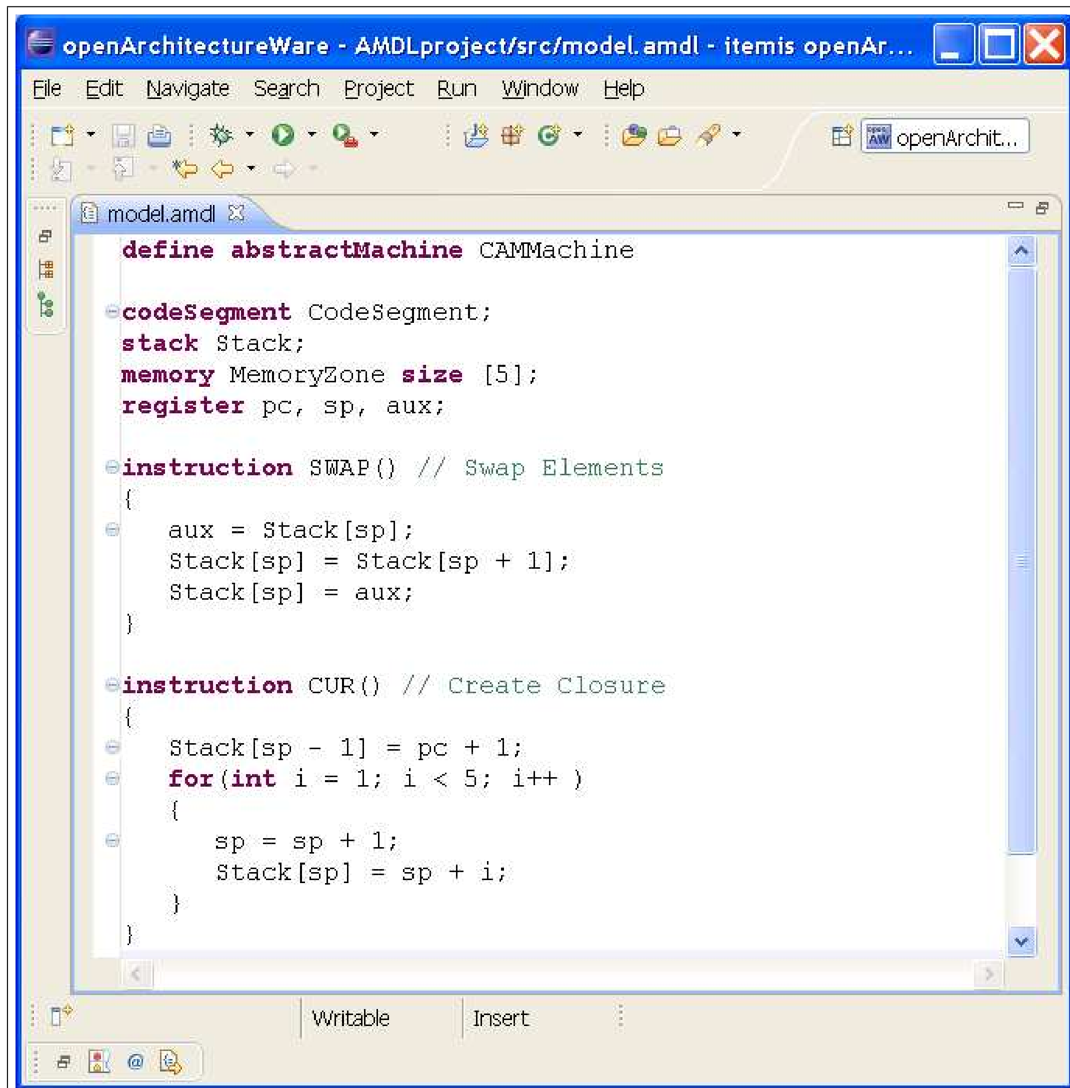

Figura 7.2: *Especificación en AMDL de la Máquina CAM*

- **P** (*Program pointer*). Referencia a la instrucción actual siendo ejecutada.
- **E** (*Environment pointer*). Referencia al último ambiente ubicado en la pila.
- **B** (*Choice point pointer*). Referencia al último *choice point* generado.
- **TR** (*Top of Trail*). Referencia al tope de la zona *Trail*.
- **H** (*Top of Heap*). Referencia a la última zona ocupada del *Heap*.

El conjunto de instrucciones de la máquina WAM se identifica en la tabla 4.3.

En la figura 7.3 se ilustra cómo el lenguaje AMDL permite especificar la máquina WAM, incluyendo estructuras e instrucciones.

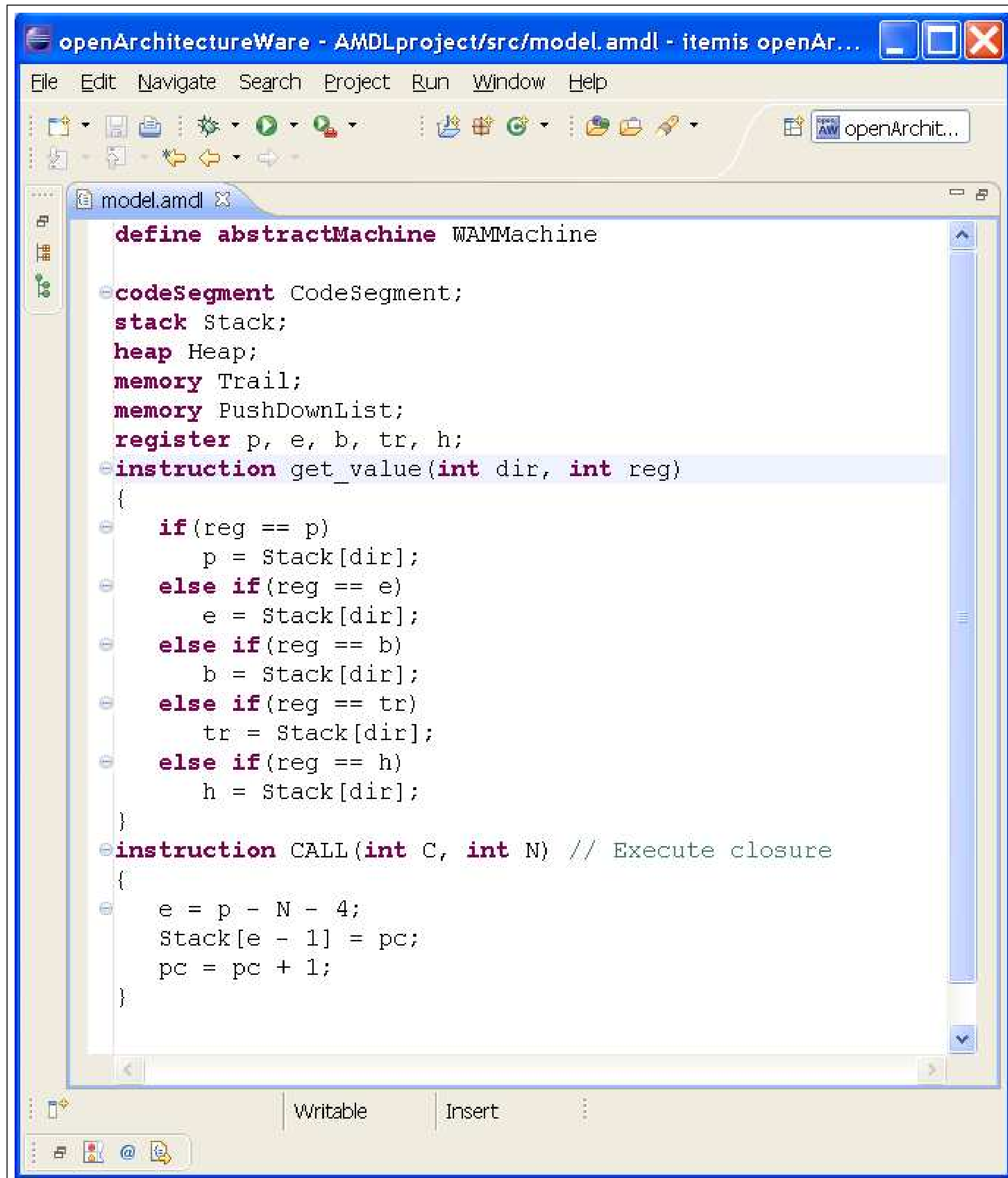

Figura 7.3: *Especificación en AMDL de la Máquina WAM*

CAPÍTULO 8

Contribuciones y Trabajo Futuro

8.1. Contribuciones

1. Identificar las características comunes en términos de estructura y de instrucciones de las máquinas abstractas que sirven para procesar lenguajes de diferentes paradigmas de programación. Estas características comunes fueron modeladas en un lenguaje de propósito específico denominado AMDL.

2. Desarrollar un editor para el lenguaje AMDL en el ambiente de desarrollo ECLIPSE que da soporte integral a las necesidades de un desarrollador como coloreado de sintaxis, detección y corrección de errores, navegabilidad en el código, etc.

3. Desarrollar un generador de implementaciones de máquinas abstractas el cual permite, a partir de una especificación de una máquina abstracta desarrollada en AMDL, generar una implementación de la misma en el lenguaje de programación Java. Esta implementación es totalmente integrable al ambiente de desarrollo ECLIPSE.

4. Identificar las necesidades futuras del lenguaje AMDL y del *framework* LAMBDA para poder desarrollar máquinas abstractas cada vez más robustas que no solo se limiten al procesamiento del lenguaje, sino que incluyan funcionalidades como verificación y optimización de código, seguridad, etc.

8.2. Trabajo Futuro

El *framework* LAMBDA tiene mucho desarrollo aún por hacer para lograr que sea un ambiente de desarrollo robusto que permita especificar y desarrollar máquinas abstractas con sus implementaciones. A continuación se listan unas tareas a desarrollar en el futuro:

1. Depurar y complementar el lenguaje AMDL para dar soporte a especificaciones de máquinas cada vez más complejas. Los lenguajes de programación siempre están en permanente evolución.

2. Especificar detalladamente e implementar cada uno de los requerimientos adicionales propuestos para el *framework* LAMBDA. Esto llevará a que LAMBDA sea un laboratorio de máquinas abstractas que de soporte a requerimientos complejos.

3. Desarrollar generadores de implementaciones de máquinas abstractas en lenguajes de bajo nivel. Esto con el objetivo de eliminar a la máquina virtual de Java como intermediario para la ejecución de una máquina generada. De esta forma será posible incrementar el desempeño general de ejecución de programas sobre las máquinas generadas.

CAPÍTULO 9

Conclusiones

1. Se pudo diseñar e implementar a cabalidad una primera versión del lenguaje propuesto. Esta versión del lenguaje satisface el alcance planteado para este trabajo de grado. Sin embargo, un lenguaje siempre está en permanente evolución. Por lo tanto, el lenguaje se debe seguir mejorando, depurando y ampliando para que cubra, en lo posible, especificaciones de máquinas abstractas que no se identificaron en la labor de investigación.

2. Al implementar el lenguaje propuesto por medio de un generador de máquinas virtuales, e integrarlo a un ambiente de desarrollo como ECLIPSE, se logró poner la piedra angular del *framework* LAMBDA. La identificación de otras funcionalidades aparte del procesamiento de un programa de entrada, que implementan las máquinas virtuales comerciales como la máquina virtual de Java, da lugar a la continuación de este proyecto de investigación en el futuro.

Bibliografía

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, 2006.

[2] Bowles, K. The UCSD P-System. `http://www.threedee.com/jcm/psystem/`. Visitado en Julio de 2009.

[3] Cousineau, G., Curien, P. L., and Mauny, M. The Categorical Abstract Machine. In *Procedures of a Conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1985), Springer-Verlag Inc., pp. 50–64.

[4] Craig, I. D. *Virtual Machines.* Springer, 2005.

[5] Daniels, M., and Pemberton, S. The Pascal Implementation. `http://homepages.cwi.nl/~steven/pascal/book/pascalimplementation.html`. Visitado en Junio de 2009.

[6] Finkel, R. A. *Advanced Programming Language Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] Fowler, M. Domain Specific Language. `http://martinfowler.com/bliki/DomainSpecificLanguage.html`. Visitado en Mayo de 2009.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] Garshol, L. M. BNF and EBNF: What are they and how do they work? `http://www.garshol.priv.no/download/text/bnf.html`. Visitado en Junio de 2009.

[10] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[11] Gregg, D., and Ertl, M. A. A Language and Tool for Generating Efficient Virtual Machine Interpreters. In *Domain-Specific Program Generation* (2003), C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, Eds., vol. 3016 of *Lecture Notes in Computer Science*, Springer, pp. 196–215.

[12] Harvey, B. *Computer Science Logo Style (2nd Edition): Volume 1: Symbolic Computing*. MIT Press, Cambridge, MA, USA, 1997.

[13] Hinze, R. The Categorical Abstract Machine: Basics and Enhancements. `http://www.informatik.uni-bonn.de/~ralf/publications/IAI-TR-92-1.ps.gz`. Visitado en Julio de 2009.

[14] Kelly, P. M., Coddington, P. D., and Wendelborn, A. L. A Distributed Virtual Machine for Parallel Graph Reduction. In *PDCAT '07: Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 331–338.

[15] Kluge, W. *Abstract Computing Machines*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[16] Lindholm, T., and Yellin, F. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] McAffer, J., and Lemieux, J.-M. Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications. `http://www.eclipse.org/documentation/`, 2005.

[18] McCarthy, J. History of LISP. `http://www-formal.stanford.edu/jmc/history/lisp/lisp.html`. Visitado en Julio de 2009.

[19] OpenArchitectureWare. `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/`. Visitado en Mayo de 2009.

[20] Parr, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[21] Pattis, R. E. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[22] Scott, M. L. *Programming Language Pragmatics (2nd Edition)*. Morgan Kaufmann, November 2005.

[23] SPINELLIS, D. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software 56*, 1 (2001), 91–99.

[24] STARK, R. F., BORGER, E., AND SCHMID, J. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[25] TAKAHASHI, S. Teoría de Lenguajes. Notas de Clase. `http://xue.uniandes.edu.co/~stakahas/web/Teoria/LIBRO/`. Visitado en Mayo de 2009.

[26] WARREN, D. An Abstract Prolog Instruction Set. `http://www.ai.sri.com/pubs/files/641.pdf`. Visitado en Junio de 2009.

[27] WILHELM, R., AND MAURER, D. *Compiler Design (1st Edition).* Addison Wesley, 1995.