INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

UN PARADIGMA PROACTIVO ORIENTADO A OBJETOS

T E S I S

QUE PARA OBTENER EL GRADO DE
DOCTOR EN CIENCIAS DE LA COMPUTACIÓN
PRESENTA:

Juan Carlos Sarmiento Tovilla

DIRECTORES DE TESIS
Dr. Juan Luis Díaz de León Santiago
Dr. Juan Carlos Chimal Eguía

México D.F. a Junio 2009

Acta de revisión

Cesión de derechos

Resumen

En la actualidad algunos investigadores conciben que los lenguajes como C++, Java y C# poseen una orientación interactiva. Esta forma de programación, por lo regular, tiende a generar costos innecesarios en el desarrollo, el diseño y principalmente en el manejo de los mensajes. Lo que implica que el desarrollador tenga un conocimiento extra del problema al aplicar una reingeniería de software. En este documento se presenta un enfoque basado en la computación proactiva, la cual busca que los objetos o dispositivos interactúen en beneficio del ser humano. Es por esta razón que surge la necesidad de desarrollar y formalizar la base de un paradigma proactivo orientado a objetos. Es decir, el paradigma propuesto da una alternativa para resolver algunos problemas que requieren ser incrementales tomando como base el paso de mensajes. Esta representación agrega reglas al paradigma orientado a objetos, lo que permite a éstos comunicarse por sentencias llamadas: *Activadores* y *Activados*.

Durante el presente trabajo se toma como base la definición de una teoría de objetos denominada $\varsigma - cálculo$ para definir $\beta\varsigma - cálculo$; en donde se muestran las reglas y derivaciones que se utilizaron para representar un cálculo proactivo. Asimismo, se realiza un lenguaje proactivo orientado a objetos para analizar el diseño y comportamiento de los objetos mismos, además se propone una forma de modelarlos con el uso de diagramas del Lenguaje Unificado de Modelado.

Abstract

At present, some researchers consider that languages like C + +, Java and C # have an interactive guide. The use of interactive programming by developers often produces unnecessary system development's costs. This involves the developer to extra knowledge when applying the software re-engineering. This paper presents an approach based on proactive computing, which looks electronic devices to interact in benefit of the human being. Due to this need, we developed and formalized the base of an object-oriented proactive-paradigm. That is, the proposed paradigm provides an alternative to solve some problems that need to be incremental, based on the passage of messages. This perspective adds rules to the object-oriented paradigm, which allows itself the objects to communicate by called methods: *Activators* and *Activated*.

During this work the basis is the definition of a theory called $\varsigma - calculus$ to define $\beta\varsigma - calculus$; it shows the rules and derivations that were used to represent the proactive calculus. It also develops an object-oriented language proactive to design and analyze the behavior of the objects, also proposes a model using diagrams of the Unified Modeling Language.

Agradecimientos

Muchas veces he pensando en la opción de comenzar y acabar esta sección únicamente con la palabra "*Gracias a todos.*", y nada más. Sencillamente para evitar el resumir en una sola página la mención de tantas personas a las que debo mucho.

Primero quiero dar gracias a Dios, por estar conmigo en cada paso que doy, por haber puesto en mi camino a aquellas personas que han sido mi soporte y compañía durante todo este periodo.

A mi familia que me acompaño durante este trabajo y que me atreví a quitarles parte de su tiempo, mil gracias Karlas, no existe palabra alguna que describa esta acción.

A mis apoyos morales, porque a pesar de no estar presentes físicamente sé que procuran mi bienestar. A mi madre Luz y tías: Lupita y Linita.

A mis hermanos: Jorge, Marco, Miguel, Margarita, gracias.

A mis directores de tesis, quienes con sus valiosos comentarios hicieron posible este documento.

También quiero agradecer a mi Jurado, quienes con sus comentarios acertados han dado un sentido a esta tesis.

Gracias a todos mis amigos que me ayudaron y que me pidieron no ser mencionados.

Índice general

Acta de revisión III

Cesión de derechos V

Resumen VII

Abstract IX

Agradecimientos XI

Símbolos XXV

Glosario XXVII

1. Introducción **3**

1.1. Resumen 3

1.2. Objetivos del capítulo 3

1.3. Preámbulo 4

1.4. Motivación 5

1.5. Problemática 7

1.6. Justificación 8

1.7. Objetivos 8

1.7.1. Objetivo general 9

1.7.2. Objetivos específicos 9

1.8. Alcance de la tesis 9
1.9. Descripción del documento 10

2. Estado del Arte **13**

2.1. Resumen 13
2.2. Objetivos del capítulo 13
2.3. Paradigmas de programación 14
2.4. Paradigma imperativo 14
2.5. Paradigma funcional 15
2.6. Paradigma de programación lógica 16
2.7. Programación Orientada a Objetos 17
2.7.1. Origen 17
2.7.2. Características de la POO 18
2.8. Otros lenguajes y paradigmas de programación 19
2.8.1. Programación estructurada 19
2.8.2. Programación dirigida por eventos 21
2.8.3. Programación orientada a aspectos 21
2.8.4. Programación con restricciones 23
2.9. Punto de vista sobre POO 24
2.9.1. Una perspectiva biológica y matemática 25
2.9.2. Lógica y computación 28
2.9.3. Abstracción y Especificación 28
2.9.4. Lenguajes de programación con tipos 29
2.9.5. Clasificación versus comunicación 30
2.9.6. Formalismos matemáticos para tipos de datos 32
2.9.6.1. ¿ Los Tipos deberían ser modelados por álgebras o cálculos? 32
2.9.6.2. Características de los tipos 33

3. Marco Teórico **35**

3.1. Resumen 35

3.2. Objetivos del capítulo 35

3.3. Introducción a $\lambda - cálculo$ 36

3.4. Programación Funcional 37

3.4.1. Variables libres y ligadas 40

3.4.2. Subtérminos y contextos 42

3.4.3. Semántica operacional 43

3.4.4. Relaciones definibles en Lambda 44

3.5. Introducción al cálculo Sigma 46

3.5.1. Las primitivas semánticas de $\varsigma - cálculo$ 47

3.5.2. Sintaxis de $\varsigma - cálculo$ 49

3.5.3. Las primitivas de $\varsigma - cálculo$ imperativo y sin tipo 50

3.5.4. Ecuaciones 52

3.5.5. Semántica Operacional 54

3.5.6. Análisis de un imp$\varsigma - cálculo$ sin tipos 56

3.6. Semántica 58

3.6.1. Semántica operacional 59

3.6.2. Semántica denotacional 61

3.6.3. La semántica axiomática 62

3.7. Modelos Concurrentes 64

3.7.1. Introducción a $\pi - cálculo$ 65

3.7.1.1. Sintaxis 65

3.8. Un cálculo para el paradigma basado en eventos 66

3.8.1. Sintaxis y una semántica informal 67

3.8.2. Sintaxis de un objeto 67

3.8.3. Abstracción de valor 68

3.8.4. Sintaxis de eventos y semántica informal 69

3.8.5. Sintaxis de tipos 69

4. Paradigma Proactivo **71**

4.1. Resumen 71

4.2. Objetivos del capítulo 71

4.3. Introducción 72

4.4. Definición de un paradigma proactivo orientado a objetos 75

4.4.1. Teoría de ecuaciones 81

4.5. Definición de $\beta\varsigma - cálculo$ 82

4.5.1. Análisis de procedimientos 84

4.5.2. Semántica operacional 85

4.5.3. Ejemplos de reducciones 88

4.6. Función secuencial de activación 90

4.7. Función $FindIF$ concurrente 92

4.8. Un intérprete basado en $\beta\varsigma - cálculo$ con tipos 93

4.9. Una propuesta para modelar $\beta\varsigma - cálculo$ con diagramas UML 96

4.9.1. Activación con disyunción 98

4.9.2. Activación bajo la conjunción 99

4.9.3. Activación bajo la negación 100

5. Disquisiciones experimentales **101**

5.1. Resumen 101

5.2. Objetivos del capítulo 101

5.3. Introducción 102

5.4. Propuesta de solución al problema del estanque 103

5.5. Ventajas y desventajas en el paradigma proactivo propuesto 105

5.5.1. Disquisiciones en el paradigma proactivo orientado a objetos . . 106

5.5.2. Disquisiciones en el uso del paradigma proactivo 107

6. Conclusiones y trabajos futuros **111**

6.1. Resumen 111

6.2. Objetivos del capítulo 111

6.3. Conclusiones 112

6.4. Trabajos Futuros 112

Appendix **113**

A. Pro-Object **115**

A.1. Resumen 115

A.2. Objetivos del apéndice 115

A.3. Diseño de Pro-Object 115

A.4. BNF de Pro-Object 116

A.5. Consideraciones para Pro-Object 117

A.6. Uso del intérprete 118

A.7. Ejemplo del uso del Intérprete 119

B. Propuesta para diagramar Pro-Object **123**

B.1. Resumen 123

B.2. Objetivos del capítulo 123

B.3. Propuesta de un diagrama estático 124

B.4. Propuesta de un diagrama dinámico 126

C. Ejemplos en Pro-Object **129**

C.1. Resumen 129

C.2. Objetivos del capítulo 129

C.3. Ejemplo básicos de reducciones en $\beta\varsigma - cálculo$ 130

C.4. Ejemplos con activadores 131

C.5. Ejemplos con clonación 132

C.6. Disquisiciones en Pro-Objects 133

Índice de tablas

3.1. Sintaxis básica para representar las abstracciones de lambda 38
3.2. Resumen de la noción del paradigma orientada a objetos 46
3.3. Expresiones del imperativo $\varsigma - cálculo$ sin tipos 50
3.4. Variables libres en $\varsigma - cálculo$ 51
3.5. Reglas de sustitución en $\varsigma - cálculo$ 51
3.6. Sintaxis del imperativo $\varsigma - cálculo$ sin tipos 57
3.7. Sintaxis del cálculo de eventos 68

4.1. Resumen de la noción del paradigma proactivo orientado a objetos . . . 76
4.2. Definición de variables libres 80
4.3. Reglas de sustitución 80
4.5. Lenguaje expresivo para denotar al imperativo-$\beta\varsigma - cálculo$ 83
4.6. Definición de un atributo 84
4.7. La sintaxis de un imperativo $\lambda - cálculo$ 85
4.12. Sintaxis básica de Pro-Object 93

5.2. Introducción a la sintaxis de $\beta\varsigma - cálculo$ 102

Índice de figuras

1.1. Diagrama general del problema del estanque 6
1.2. Diagrama de clases para dar solución al problema del estanque 6
1.3. Ejemplo gráfico simple de un paradigma proactivo orientado a objetos 7

4.1. Solución gráfica al problema del estanque 74
4.2. Definición de Almacenamiento y Pila 85
4.3. Relación de activación con un único atributo 97
4.4. Diagrama proactivo con activación de dos objetos 98
4.5. Diagrama proactivo para la disyunción 99
4.6. Diagrama proactivo con conjunción 99
4.7. Diagrama proactivo para la negación 100

5.1. Diagrama proactivo a la posible solución del estanque 104

B.1. Estereotipo de relación para objetos proactivos 125
B.2. Diagrama de objetos 125
B.3. Diagrama de secuencia y objetos proactivos 126

Índice de algoritmos

4.1. Código abstracto para activar los objetos 75
4.2. Semáforos 92
5.1. Calculadora básica 103
5.2. Código de los objetos para el problema del estanque 104
5.3. Código de un objeto proactivo llamado Monitor2 105
5.4. Código para representar un ciclo *for* imperativo 107
5.5. Código para representar el problema de paro 108
5.6. Actualización de una función de activación 110
A.1. Problema de ambigüedad con la sentencia else 118
A.2. Solución al problema de la ambigüedad 118
A.3. Código que representa a los Numerales de Church 119
A.4. Código que representa al numeral uno 120
A.5. Código que representa al numeral dos 120
C.1. Transitividad 130
C.2. Selección 130
C.3. Método 130
C.4. Paso de parámetros 131
C.5. Activador cerrrado 131
C.6. Activador libre 131
C.7. Una solución simple al problema del Estanque 132
C.8. Clonación en $\beta\varsigma - cálculo$ 132
C.9. Clonación superficial y en profundidad 133
C.10. Solución al problema del estanque con el uso de clonación 133
C.11. Problema de Paro 133
C.12. Inestabilidad entre objetos 134
C.13. Moviles en la calle 135

Símbolos

Cálculo sin tipos

$=$	Igual informal
$\equiv$	Sintácticamente idéntico
$\triangleq$	Igual por definición
$::=$	Definición sintáctica
$\Phi_i^{i \in 1 \ldots n}$	Secuencia de $\Phi_1, \ldots, \Phi_n$
ς	Auto ligadura o ligadura al objeto $Self$
a, b, c	Términos
x, y, z	Variables para términos
$b\langle x \leftarrow c \rangle$	Sustitución donde el término c es remplazado por todas las ocurrencias libres de x en el término b
l	Etiquetas para nombrar a un método
$[l_i = \varsigma(x_i)b_i]$	Definición básica de un objeto donde $i \in 1 \ldots n$
$\varsigma(x)b$	Método con parámetro x y cuerpo b
$a.l$	Seleccionar un atributo o invocar a un método
$o.l \Leftarrow \varsigma(x)b$	Actualización de un método
$FV(a)$	Variables libres de a
$\rightarrowtail$	Reducción de un paso a nivel más alto
$\rightarrow$	Reducción de un paso
$\twoheadrightarrow$	Reducción de muchos pasos
$b\{x\}$	x puede ocurrir libre en b
$b\langle c \rangle$	Similar a $b\langle x \leftarrow c \rangle$, pero con una representación simple cuando x no se encuentra en el contexto
$[\ldots, l = b, \ldots]$	Atributo o campo
$a.l \Leftarrow b$	Actualización de un atributo o campo

$\vdash \Im$	Juicio de evaluación de expresiones
$\vdash a \leftrightarrow b$	Juicios equivalentes
v	Resultado
$\vdash a \rightsquigarrow v$	Reducción completa del juicio
$\lambda(x)b$	Función de abstracción
$b(a)$	Función de aplicación
$\lceil l_i = a_i \, ^{i \in 1 \ldots n} \rceil$	Registro o estructura

Cálculo de primer orden

$\emptyset$	Ambiente con tipos vacío
$E \vdash \Im$	Juicio
$\frac{E_1 \vdash \Im_1 \ldots \, E_n \vdash \Im_n}{E \vdash \Im}$	Regla de inducción
$\downarrow \frac{T_1}{T_n}$	Árbol de derivación
$E \vdash \Diamond$	Juicio del ambiente o entorno
A, B, C	Tipos

Glosario

Azúcar sintáctico	Es un término acuñado por Peter J. Landin para las adiciones a la sintaxis de un lenguaje de programación que no afectan su expresividad, se refiere a hacerla "*más dulce*" para los seres humanos al momento de su uso.
Bisimulación	Es una relación de equivalencia entre dos sistemas.
Clase	Una función que regresa un objeto, en el cual se definen las propiedades y comportamientos de un objeto concreto. La referencia es la lectura de estas definiciones y la creación de un objeto a partir de ellas.
Compilador	Traduce un programa fuente a un programa objeto. Parte de su tarea consiste en verificar que la cadena de entrada sea válida.
Componentes de un objeto	Atributos, identidad, relaciones y métodos.
Concurrencia	En computación, la concurrencia es la propiedad de los sistemas que permiten que múltiples procesos sean ejecutados al mismo tiempo y que potencialmente puedan interactuar entre sí. En propias palabras de Edsger Dijkstra, la concurrencia ocurre al momento en que dos o más flujos de la ejecución pueden efectuarse simultáneamente.
Correctness o correctud	Afirmación que ocurre al momento en el que un algoritmo cumple una especificación. La correctud funcional refiere al comportamiento de la entrada-salida del algoritmo es decir, cada entrada produce una salida correcta.

Curried	El origen del nombre *curry*, proviene de la persona que popularizó el uso de la parcialización: Haskell Curry.
Delegación	Un mecanismo de la orientación a objetos para optimizar y reutilizar algunos componentes o partes del código.
Estado interno	Es una propiedad invisible de los objetos, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos).
Evaluación perezosa (o *lazy*)	Evaluación en orden normal con las siguientes características adicionales: Primero, el argumento de una función sólo se evalúa cuando es necesario para el cómputo. Segundo, un argumento no es necesariamente evaluado por completo; sólo se evalúan aquellas partes que contribuyen efectivamente al cómputo. Tercero, si un argumento se evalúa, tal evaluación se realiza una sola vez.
Evento	Es un suceso en el sistema tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto. El sistema maneja el evento lo que produce que se envíe el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir, la acción que genera.
Expresiones	Términos sintácticos.
Expresiones atómicas	Son las expresiones más simples llamadas también formas normales por abuso de lenguaje a las que se les denomina valores.
Herencia	Un mecanismo para crear una clase o tipo a partir de otro, en el cual el tipo *Self* o auto parámetro se encuentra referenciado de manera tardía.

Intérprete	Programa que computa las acciones indicadas por un programa fuente. Conocido de igual forma como un meta-programa, es decir, un programa que procesa a otro programa, que comúnmente es utilizado para realizar prototipos basados en la semántica de un lenguaje.
La lógica de Hoare	También conocida como lógica de Floyd-Hoare. Es un sistema formal desarrollado por el informático británico C.A.R. Hoare, y refinado posteriormente por él mismo y otros investigadores. Fue publicada en el artículo de Hoare en el año de 1969, "*una base axiomática para programar la computadora*". El propósito del sistema es proporcionar un sistema de reglas lógicas para razonar sobre la consistencia de los programas de computadora, con el rigor de la lógica matemática.
Lenguaje funcional puro	Lenguaje de expresiones con transparencia referencial y funciones de orden superior, cuyo modelo de cómputo es la reducción realizada mediante el reemplazo de iguales por iguales.
Máquinas abstractas de estados	Álgebra más Semántica Operacional, basada en la abstracción de estados con reglas de transición entre elementos de ese Estado.
Mensaje	La invocación de $a.l$ donde l es una función y a es un objeto. Una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.
Método	Un procedimiento dentro de un objeto, éste por lo regular se encuentra asociado a un mensaje. Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "*mensaje*". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer; en donde un método puede producir un cambio en las propiedades del objeto o la generación de un "*evento*" con un nuevo mensaje para otro objeto del sistema.

Objeto	Término primitivo, entidad provista de un conjunto de propiedades o atributos (*datos*) y de comportamiento o funcionalidad (*métodos*). Corresponden a los objetos reales del mundo que nos rodea o a objetos internos del sistema (programa).
Proactivo	Término acuñado por Victor Frankl. La palabra procede del latín, y está compuesta de dos palabras (pro, raíz latina: pro-, que significa «*delante de*», y actividad, que significa «*facultad de obrar*»). En computación el término proactivo se ha utilizado para definir la manera en que el ser humano interactúa con los diferentes dispositivos electrónicos.
Propiedad o atributo	Contenedor de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto, y cuyo valor puede ser alterado por la ejecución de algún otro método.
Representación de un objeto	Un objeto se representa por medio de una tabla o entidad que está compuesta por sus atributos y sus funciones correspondientes.
Semántica	Descripción del significado de instrucciones y expresiones puede ser informal (ejemplo: Castellano) o formal (basado en técnicas matemáticas). La semántica formal puede ser axiomática, operacional o denotacional.
Sintaxis	Descripción del conjunto de secuencias de símbolos considerados como programas válidos.
Teoría modelo del agente	Los agentes son los primitivos que forman la base del modelo del agente del cómputo digital concurrente. En respuesta a un mensaje que reciba: un agente puede tomar decisiones locales, crear sdf más agentes, enviar más mensajes y señalar cómo responder al mensaje siguiente recibido. La teoría modelo del agente incorpora las teorías de los acontecimientos y de las estructuras de los cómputos del agente, su teoría de la prueba y modelos denotacionales.

Tipificado (*type safe*)	Se refiere a que el programador no puede evaluar expresiones con tipos erróneos.
Tipo	En lenguajes de programación un tipo de dato es un atributo de una parte de los datos que indica al computador y al programador sobre la clase de datos que se van a procesar.
Valores	Entidades abstractas que son vistas como respuestas.

Capítulo 1

Introducción

1.1. Resumen

En este capítulo se exponen los objetivos generales y específicos de la presente tesis. Asimismo se indica la problemática general y su justificación.

1.2. Objetivos del capítulo

- Plantear los objetivos generales y específicos de la tesis.
- Establecer un contexto en el cual se permita tener una visión sintetizada del contenido del documento y su justificación de estudio.
- Mostrar el alcance del actual trabajo, así como una descripción de las partes que lo integran.

1.3. Preámbulo

Entre los paradigmas de programación han existido distintas clasificaciones de las cuales se pueden mencionar: imperativo o por procedimientos, funcional, lógico y orientado a objetos. De estos, la programación orientada a objetos ha tenido énfasis en la elaboración de los sistemas de cómputo [6]. Los lenguajes que de este paradigma surgieron son diseñados para permitir una forma de ver los datos y el cómputo de manera unida [4]. Esto ha permitido crear representaciones entre el software y el mundo de los objetos físicos de una forma simple e intuitiva, por así llamarlo.

Algunos lenguajes han sido realizados con la noción de tipos, subtipos y procesos. Estos lenguajes han tomado como base características de $\lambda - cálculo$ para establecer sus propias condiciones de operación. Por otra parte, para Martin Abadí y Luca Cardelli esta noción se encuentra diseñada para los lenguajes basados en secuencias de procesos y no para modelos basados en objetos [6, 7]. Debido a esto, ellos proponen un cálculo llamado $\varsigma - cálculo$, el cual permite una representación formal del paradigma orientado a objetos. Esta formalización fue realizada de dos maneras: mediante rasgos funcionales y rasgos imperativos [6].

En $\varsigma - cálculo$ se realizan ciertos ajustes en comparación con su homólogo $\lambda - cálculo$ [15]. Los cálculos y demostraciones por medio de este proceso están basados exclusivamente en objetos y métodos, y no en estructuras de datos o funciones. Estos principios ayudan a clasificar y a explicar algunos de los rasgos de los lenguajes orientados a objetos [97].

Una forma simple de tener un acercamiento a la programación orientada a objetos, es basarse en una intuitiva analogía que puede hacerse entre la simulación de un sistema físico mediante un software y viceversa [6]. Un modelo físico está formado por objetos, pero se puede llegar a necesitar más objetos que no existen realmente en el sistema físico. Es de esta manera en que el software puede estar formado por objetos análogos y abstractos que representan al modelo físico [1].

La amplia aplicabilidad del acercamiento a los lenguajes orientados a objetos (LOO), puede ser atribuida a la analogía descrita anteriormente. También existen otros factores que hay que tener en cuenta dentro del diseño de un sistema, como son: la resistencia y la reusabilidad [6].

1.4. Motivación

En la programación orientada a objetos existen diferentes maneras de realizar modelos o analogías a modelos físicos o mecánicos del mundo real. Muchos de estos modelos proponen utilizar aspectos como: herencia, polimorfismo, agregación y composición [61, 88, 31]. Todos estos puntos de análisis buscan dar facilidad para implantar un sistema de cómputo, tratando de ser equivalente a los sistemas físicos. Se pide en la analogía ser *resistentes* a los modelos planteados y permitir la *reutilización* de los componentes que lo integran [6].

Uno de los objetivos de este documento, es proponer otra manera intuitiva de modelar algunos problemas bajo el principio de activaciones llamadas *activaciones proactivas*. Para ello, se deberán tomar en cuenta los puntos fundamentales para la reutilización, la resistencia y una intuición apropiada para modelar ciertos problemas físicos [6].

En este trabajo se mostrará un vértice de la programación orientada a objetos, así como los fundamentos teóricos y prácticos que permitan mostrar las implicaciones en el análisis, diseño e implantación de un software.

Para tener una mejor noción de dicho paradigma basado en mensajes de activación, se plantea el siguiente problema:

Se posee un estanque t_1 de agua con tres sensores (s_1, s_2 y s_3) que detecta tres niveles (n_1, n_2 y n_3), donde n_1 es el nivel bajo, n_2 es un nivel aceptable y n_3 nivel alto (peligro de desbordamiento). Se solicita realizar un sistema de cómputo basado en objetos que simule dicho proceso físico, con la restricción de que en el momento en que el agua llegue al nivel n_3 se active una alarma a_1.

Si bien, uno de los objetivos de la programación orientada a objetos es que éste represente y simule a un sistema físico mediante alguna analogía; también es importante que cumpla con otros requisitos como son: el de *reuso* y la *evolución*.

Primero se tomará un esbozo industrial (ver Figura 1.1), el cual permite tener una visión del problema del estanque a simular.

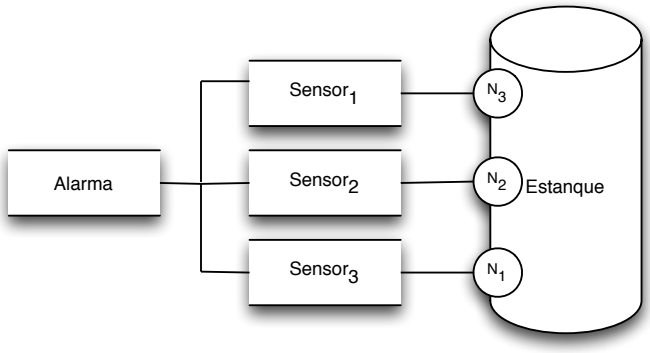

Figura 1.1: Diagrama general del problema del estanque

Posteriormente se analizará con más profundidad un diseño apegado al patrón Observador[1] [41, 44]. En éste se podrá estudiar una posible solución al problema del estanque.

Se denota que para realizar un diseño con las características del patrón Observador, es necesario delegar a un objeto llamado *monitor* la responsabilidad de revisar y tener la lógica de selección. Este objeto se encargará de observar los diferentes niveles y de emitir la señal correspondiente a la alarma seleccionada.

Una manera simple de representar este diseño es mediante un diagrama UML[2] [19], en el cual se muestren los objetos abstractos que interactúan en el modelo del estanque (ver Figura 1.2).

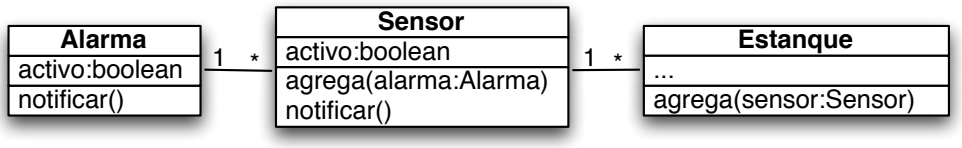

Figura 1.2: Diagrama de clases para dar solución al problema del estanque

Un problema que se presenta en este diseño sucede al momento en que el objeto *Sensor* delega la responsabilidad de realizar la verificación lógica (mensaje de activación) a Monitor. Esto es, en el instante en el que nivel n_3 es alcanzado se activará la alarma correspondiente. Este problema se discutirá con detenimiento en el Capítulo 4; donde se muestran algunas alternativas de solución y sus implicaciones en el diseño y análisis del problema.

[1] También conocido como Observer en el libro de Diseño de Patrones conocido como GOF.
[2] Siglas en inglés que significan Unified Process Model.

Cabe indicar que el diseño mostrado en la Figura 1.2 está basado en un modelo jerárquico con tipos y sin covarianza [6]; por lo que se deberá tener cuidado con la naturaleza de los objetos al momento de implantarlo en algún lenguaje orientado a objetos.

Ahora bien, este mismo problema será bosquejado en la Figura 1.3 bajo el *paradigma proactivo orientado a objetos* propuesto. No se intenta ahondar en este punto por el momento, sólamente es una ilustración para observar una manera simple de cómo resolver el proceso involucrado.

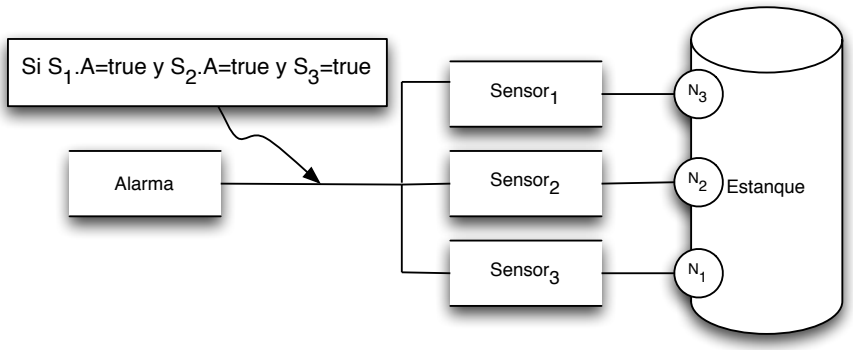

Figura 1.3: Ejemplo gráfico simple de un paradigma proactivo orientado a objetos

En la Figura 1.3 se presenta un esbozo general de cómo podría operar el paradigma propuesto. Se observa que la responsabilidad está en cada parte física, como se ve en la Figura 1.1. Además, se muestra que la condición de activación de la alarma está dentro del mensaje o relación, a la que se llamará *mensaje de activación*. Esta indica cual será el objeto que se ha de activar y en qué condiciones funcionales o imperativas se realizará dicho proceso. En los Capítulos 4 y 5 se explicará con detalle esta última forma de modelar.

1.5. Problemática

El paradigma orientado a objetos ha tenido un impacto importante en el desarrollo de sistemas de cómputo, lo que ha provocado que existan clasificaciones dentro del mismo paradigma. Su jerarquía se encuentra definida en dos ramas principales: la primera está dada por clases y la segunda está basada en prototipos [6]. Sin embargo, las dos representaciones antes mencionadas están fundamentadas en pasos de mensajes secuenciales e interactivos. Originando con esto, presentar un problema al modelar sistemas *proac-*

tivos en forma *interactiva*. Estos modelos tienden por consiguiente, a generar costos elevados en los ciclos de desarrollo informático.

1.6. Justificación

En el trabajo "*Un paradigma proactivo orientado a objetos*" se propone crear un modelo que exponga la creación de sistemas proactivos en función de mensajes de activación [6, 89]. Esto permite que el modelo sea robusto, incremental, evolutivo e independiente de una jerarquía implícita y obligada en la parte del diseño [98, 94].

El principal problema al que se enfrenta un diseñador informático es el de modelar un sistema. Esta actividad por lo regular representa una transferencia de actividades proactivas a un esquema basado en una secuencia de procesos y durante el análisis, el esquema es interpretado por una máquina o autómata en particular. Posteriormente, en el ciclo de desarrollo los costos aumentan principalmente por la reingeniería de procesos. Esto es debido a que, el diseñador deberá conocer los aspectos importantes de la programación y del modelo para "modificar o adecuarlo" a los nuevos requerimientos.

Hay que recordar que uno de los objetivos principales de la programación orientada a objetos es el de la reutilización del código, que en este caso son los objetos. Realizar una reingeniería de procesos no es una tarea simple y en ocasiones es preferible rehacer el módulo al momento en que el diseño no es claro.

Uno de los retos en este documento es el de proyectar una alternativa de programación mediante *un paradigma proactivo orientado a objetos*. Esta opción permitirá experimentar de manera formal con diferentes modelos y diseños de cómputo [77, 76]. Además de desarrollar una herramienta formal de programación que haga uso de dicho paradigma incremental [84, 66].

1.7. Objetivos

A continuación se describe el objetivo general y los objetivos específicos del presente trabajo.

1.7.1. Objetivo general

Presentar un paradigma proactivo orientado a objetos el cual sirva como base para modelar sistemas de cómputo de manera incremental. Igualmente, se pretende proporcionar las bases formales que definan la funcionalidad de los objetos dentro de un contexto. Esto se realizará mediante una herramienta de programación creada para este fin llamada $\beta\varsigma - cálculo$.

1.7.2. Objetivos específicos

Los objetivos particulares de esta tesis son los siguientes:

- Analizar y estudiar los conceptos que apoyan a los paradigmas orientados a objetos.
- Investigar y estudiar las formas evolutivas que han presentado los lenguajes orientados a objetos para analizar futuras alternativas.
- Mostrar el estudio formal de los lenguajes orientados a objetos.
- Conjuntar los conceptos de la programación orientada a objetos para establecer los aspectos relevantes de un paradigma proactivo orientado a objetos.
- Definir las reglas fundamentales que deberá poseer dicho paradigma.
- Formalizar un paradigma proactivo orientado a objetos mediante $\beta\varsigma - cálculo$.
- Desarrollar y diseñar una herramienta computacional llamada *pro-object*, la cual permitirá experimentar con algunos problemas comunes de cómputo y programación.
- Realizar pruebas y observaciones que ayuden en un futuro a complementar dicho paradigma.

1.8. Alcance de la tesis

En el documento se revelan los aspectos fundamentales de la programación orientada a objetos. Además de exponer los conceptos que rodean a este paradigma, se formalizará

un paradigma proactivo orientado a objetos de manera funcional e imperativa. Asimismo, se creará un marco de trabajo que permitirá experimentar algunos ejemplos de programación incremental, así como una propuesta para modelar los mismos. También se exhiben aspectos relevantes de interés y de discusión dentro del paradigma propuesto, al igual que algunas soluciones bajo aspectos formales y de modelado, como son:

- Abstracción por parametrización: Utilizada para definir las precondiciones y poscondiciones de activación de los objetos que lo componen [59, 77].
- Abstracción por especificación: Se busca examinar la robustez de un sistema bajo los principios de aseveración [59, 23, 46, 58].
- Principio de diseño orientado a objetos: Se analizarán algunos principios del diseño orientado a objetos [41, 51, 84].
- Sobre-escritura y Mutabilidad: Se examinarán estos aspectos debido a que permiten una evolución del software sin dependencias explícitas *fuertes* [6, 21, 99].
- Clases y Objetos: Se presentan alternativas para complementar los dos aspectos dentro del paradigma propuesto [6, 35].

1.9. Descripción del documento

Se analizarán los lenguajes orientados a objetos en busca de la abstracción que de ellos emana para retomar los conceptos necesarios que rodean a la programación con este paradigma [93, 97, 98, 96]. Éstos permitirán implantar un marco teórico con $\varsigma - cálculo$ con una visión proactiva. De este modo se busca dar respuesta a los aspectos de: abstracción/especificación, composición y agregación; que son algunas de las características que han rodeado a la programación orientada a objetos [87, 59].

El presente trabajo está definido por la siguiente estructura:

En el Capítulo 1 se hace una **Introducción** al trabajo realizado.

En el Capítulo 2 se describe el **Estado del arte**. En el cual se pueden leer los puntos de los diferentes paradigmas de programación. El estudio que se realizó es histórico y de una manera descriptiva. También se presentan algunas de sus características importantes y su uso en el desarrollo de programas de cómputo.

En el Capítulo 3 se presenta el **Marco teórico**. Se mostrarán las herramientas formales y las bases teóricas que dan soporte a la realización de la presente tesis.

En el Capítulo 4 **Un paradigma proactivo orientado a objetos**. Se da una introducción informal al funcionamiento básico del paradigma propuesto. De la misma manera se proporciona una formalización funcional e imperativa, y se muestran ciertas implicaciones en el diseño de sistemas. También se enuncia una serie de diagramas que pueden ayudar a comprender de manera intuitiva el desarrollo de aplicaciones basadas en este concepto.

En el Capítulo 5 **Disquisiciones experimentales**. Se exponen algunos ejemplos sobre la funcionalidad de este paradigma de programación (calculadora, problema del estanque, numerales, entre otros), y se reflejan las implicaciones en su uso. Asimismo, se propone la definición de varios problemas de programación de uso común.

El Capítulo 6 **Conclusiones y trabajos futuros**. Se mostrarán las deducciones que durante el desarrollo de la actual tesis se encontraron; junto con algunas ventajas y desventajas en el uso del tema propuesto. De este modo, se presenta una lista de nueve puntos que se consideran los más relevantes para realizar una futura implantación del paradigma propuesto.

Por último, se agregaron dos Apéndices A, B y C. En el Apéndice A se exhibe el diseño, implantación y uso de una herramienta desarrollada y llamada Pro-Object. Con esta herramienta se pueden realizar los ejercicios de $\varsigma-$*cálculo* y los de $\beta\varsigma-$*cálculo* mostrados en el Capítulo 3. En el Apéndice B se presentan algunas propuestas para representar a los objetos mediante diagramas del tipo UML. En el Apéndice C se muestran algunos ejemplos que pueden ser interpretados con Pro-Objects (software que se encuentra en el disco adjunto).

Capítulo 2

Estado del Arte

2.1. Resumen

En este capítulo se presentan los conceptos de los diferentes paradigmas de programación y sus derivaciones, así como un análisis de las nociones del paradigma orientado a objetos.

2.2. Objetivos del capítulo

- Analizar los conceptos básicos que rodean a la programación y a sus paradigmas.
- Presentar un panorama general sobre algunos paradigmas de programación y en especial el orientado a objetos.

2.3. Paradigmas de programación

Un paradigma de programación representa un enfoque particular para la realización del software, que ofrece una serie de construcciones de software que al combinarse ayude a la resolución de problemas dentro de un determinado contexto. Sin embargo, la forma de resolver un problema en el contexto del paradigma debe ser en cierto sentido satisfactoria. Algunos paradigmas de programación se muestran en la siguiente lista:

- El paradigma imperativo el cual es considerado uno de los más comunes en la industria que se encuentra representado por lenguajes como: Pascal y C [101, 53].
- El paradigma funcional es considerado como uno de los más eficientes y robustos. Es utilizado en el área de la investigación y está representado por lenguajes como: LISP, ML y Haskell [15, 30, 54, 79].
- El paradigma lógico está basado en un motor inferencial, un ejemplo es: PROLOG [24, 86].
- El paradigma orientado a objetos que en los últimos años ha tenido un auge importante en la industria. Sus representantes más importantes son: Smalltalk, C++, Java y C#, entre otros [66, 59, 22, 55].

2.4. Paradigma imperativo

La programación imperativa en contraposición con la programación declarativa es un paradigma de programación que describe a dicho proceso en función de estados y sentencias [53, 85, 101].

La implementación de hardware en la mayoría de las máquinas de cómputo se realiza de manera imperativa. Esto se debe, a que implementan el modelo de las Máquinas de Turing [42, 80, 62]. Desde esta perspectiva a bajo nivel, el estilo del programa está definido por los contenidos de la memoria y las sentencias (instrucciones en el lenguaje de máquina nativa a dicho hardware).

Los primeros lenguajes imperativos fueron los lenguajes de máquina. En esos lenguajes, las instrucciones fueron simples, lo cual hizo la implementación de hardware de manera fácil, pero se obstruyó la creación de programas complejos. La evolución de estos lenguajes tuvo sus inicios en el año de 1954, con John Backus de IBM, quién desarrolló uno

de los primeros lenguajes imperativos los cuales superaron los obstáculos presentados por el código de máquina ante la creación de programas complejos [13].

2.5. Paradigma funcional

La Programación funcional es un paradigma de programación declarativa basada en la utilización de funciones matemáticas; sus orígenes provienen del Cálculo Lambda (o $\lambda - cálculo$), una teoría matemática elaborada por Alonzo Church [15, 14, 79]. Esta teoría fue realizada como apoyo a sus estudios sobre la computabilidad [91, 54, 78].

El objetivo de este paradigma es el de conseguir lenguajes expresivos y matemáticamente elegantes. No es necesario bajar al nivel de la máquina para describir el proceso que lleva a cabo; con lo que se evita el concepto de estado en el cómputo. La secuencia de cálculos efectuados por el programa se regiría única y exclusivamente por la reescrituración de definiciones más amplias a otras cada vez más concretas y definidas. En donde se hace uso de lo que se denominan *definiciones dirigidas*.

Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de funciones. Esto se entiende no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas; en las que se verifican ciertas propiedades como la transparencia referencial (el significado de una expresión depende únicamente del significado de sus sub-expresiones). Por lo tanto, existe la carencia total de efectos laterales [14, 16, 67].

Otras características propias de estos lenguajes son: la no existencia de asignaciones de variables únicas y la falta de construcciones estructuradas, como: la secuencia o la iteración (lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas).

Existen dos grandes categorías de lenguajes funcionales: los funcionales puros y los híbridos. La diferencia entre ambos estriba en que los lenguajes funcionales híbridos son menos dogmáticos que los puros, al admitir conceptos tomados de los lenguajes imperativos; como las secuencias de instrucciones o la asignación de variables. En contraste, los lenguajes funcionales puros tienen una mayor potencia expresiva; por lo que conservan su transparencia referencial, algo que no se cumple en un lenguaje funcional híbrido.

Entre los lenguajes funcionales puros, cabe destacar a Haskell y Miranda [70, 17, 91]. Los lenguajes funcionales híbridos conocidos son: Lisp, Scheme, Ocaml y el estándar ML.

2.6. Paradigma de programación lógica

Históricamente los equipos de cómputo que se han programado utilizan lenguajes cercanos a las peculiaridades de la propia máquina, en si: operaciones aritméticas simples, instrucciones de acceso a memoria, etc. Un programa escrito de esta manera puede ocultar parte de su propósito a la comprensión del ser humano. Hoy en día, los lenguajes pertenecientes al paradigma de la programación imperativa han evolucionado de manera que ya no son crípticos. Sin embargo, aún existen casos en donde el uso de lenguajes imperativos es inviable debido a la complejidad del problema a resolver.

En cambio, la lógica matemática es la manera sencilla de expresar formalmente problemas complejos y de resolverlos mediante la aplicación de reglas, hipótesis y teoremas [24]. De ahí que el concepto de *programación lógica* resulta atractivo para los diversos campos, donde la programación tradicional no muestra los resultados esperados.

La programación lógica consiste en la aplicación del corpus de conocimiento sobre lógica para el diseño de lenguajes de programación. La programación lógica comprende dos paradigmas: la programación declarativa y la programación funcional [86]. La primera gira en torno al concepto de predicado o relación entre elementos. La segunda se basa en el concepto de función (que no es más que una evolución de los predicados), de corte matemático [90]. Encuentra su hábitat natural en aplicaciones de inteligencia artificial o algún área relacionada a:

- Sistemas expertos, en el que un sistema de información imita las recomendaciones de un experto sobre algún dominio de conocimiento.
- Demostración automática de teoremas, en el cual un programa genera nuevos teoremas sobre una teoría existente.
- Reconocimiento de lenguaje natural, donde un programa es capaz de comprender (con limitaciones) la información contenida en una expresión lingüística humana, etc.

De igual forma se utiliza en aplicaciones *comunes* pero de manera limitada, debido a que la programación tradicional es más adecuada a las tareas de propósito general.

La mayoría de los lenguajes de programación lógica se basan en la teoría lógica de primer orden, aunque también incorporan algunos comportamientos de orden superior. En este sentido destacan los lenguajes funcionales, ya que están fundamentados en el $\lambda - cálculo$, la cual es una teoría lógica de orden superior, que es computable.

2.7. Programación Orientada a Objetos

La Programación Orientada a Objetos (POO u OOP según siglas en inglés) es un paradigma de programación que define los programas en función de *clases* y *objetos*. Objetos que son entidades que combinan estado (es decir, datos), comportamiento (esto es, procedimientos o métodos) e identidad (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos expresa un cálculo como un conjunto de estos objetos que se relacionan entre si para realizar ciertas tareas, esto permite que los programas sean fáciles de escribir, mantener y reutilizar.

2.7.1. Origen

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones; creado por Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo [52, 57].

El lenguaje para simulaciones surgió a partir de la necesidad de agrupar los diversos tipos de objetos. Para ello, se establecieron las condiciones en las que cada objeto debería ser responsable de su definición y comportamiento. Éste fue mejorado más tarde, dando origen al llamado Smalltalk siendo desarrollado en Simula en Xerox PARC. Una de las características que lo hacían atractivo era su diseño, el cual establecía sus bases en la teoría en que los objetos deberían ser dinámicos, pudiéndose agregar y modificar éstos en *tiempo de ejecución*[1].

De esta forma, un objeto contiene toda la información (atributos) que permite definirlo e identificarlo frente a otros objetos. A su vez, dispone de mecanismos de interacción (métodos) que favorecen a la comunicación entre objetos, y en consecuencia el cambio de estado en los propios objetos. Esta característica lleva a relacionarlos como unidades indivisibles, en las que no se separan (ni deben separarse) información (datos) y procesamiento (métodos).

Dada las propiedades antes mencionadas el programador debe pensar indistintamente en ambos términos, ya que no debe separar o dar mayor importancia a los atributos en favor de los métodos, ni viceversa. Hacerlo, puede llevar al programador a seguir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen esa información por otro (esto genera una programación estructurada oculta en un lenguaje de programación orientado a objetos).

[1] En la literatura de habla inglesa se describe como *run time*.

2.7.2. Características de la POO

Existe un cierto desacuerdo sobre exactamente qué características de un método de programación o lenguaje se califican como *orientado a objetos*, sin embargo hay un consenso en el que las características siguientes son algunas de las más relevantes [47, 52]:

Abstracción: Cada objeto en el sistema sirve como modelo de un agente abstracto que puede realizar algún trabajo, informar y cambiar su estado, al que se le permite comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características o acciones. Los procesos, las funciones o los métodos pueden también ser abstraídos. En el momento que éstos estén presentes se requiere de algunas técnicas para especificar dichas abstracciones.

Encapsulamiento: También llamada ocultación de la información, esto asegura que los objetos no pueden cambiar el estado interno de otros objetos de manera inesperada. Solamente los propios métodos internos al objeto pueden acceder a su estado. Cada objeto expone una interfaz a otros objetos que especifica cómo otros objetos pueden interactuar con él. Algunos lenguajes permiten esto, por lo que ceden a un acceso directo a los datos internos del objeto de una manera controlada y limitan el grado de abstracción.

Polimorfismo: Las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos. La invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del referente. En el momento en que esto ocurre en tiempo de ejecución, esta última característica se llama asignación tardía o asignación dinámica. Algunos lenguajes proporcionan medios estáticos (en tiempo de compilación) de polimorfismo, tales como: las plantillas y la sobrecarga de operadores de C++ o C#.

Herencia: Organiza, así como facilita el polimorfismo y la encapsulación, esto permite a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Éstos pueden compartir (y extender) su comportamiento sin tener que reimplantarlo. Esto suele hacerse habitualmente con grupos de objetos y clases que reflejan un comportamiento común.

La programación orientada a objetos introduce nuevos conceptos que a veces no son más que nombres nuevos aplicados a conceptos antiguos. Entre ellos destacan los siguientes:

Método: es un programa asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena mediante un mensaje. En analogía con un lenguaje basado en procedimiento se le llamaría función.

Mensaje: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros.

Propiedad: atributo o variable: datos asociados a un objeto o a una clase de objetos.

2.8. Otros lenguajes y paradigmas de programación

Las restricciones impuestas por un paradigma implican que éste no pueda ser el más conveniente para resolver ciertos problemas, ello conduce a la pluralidad de modelos de programación y por lo consiguiente, a la aparición de nuevos lenguajes de programación. Así por ejemplo, la programación funcional, la programación lógica (PL) o la programación orientada a objetos (POO), han proporcionado las bases de diferentes paradigmas.

La diferencia entre los distintos paradigmas radicará en el punto de vista de la resolución que el programador tiene del problema. Esta parcialidad en la visión del modelo abstracto del problema es la base del éxito de un paradigma de programación. Lo que lleva a simplificar el planteamiento de la solución, pero al mismo tiempo puede hacer que se muestre inadecuado en la resolución de problemas.

Situaciones como ésta hacen que algunos paradigmas sean reemplazados por modelos más expresivos, y consecuentemente los lenguajes de programación correspondientes sean extendidos.

2.8.1. Programación estructurada

La programación estructurada permite realizar programas para equipos de cómputo de manera *clara*. Para ello se utiliza únicamente algunas estructuras como son: secuencial, selectiva, iterativa y de transferencia. Esta última es innecesaria ya que su uso puede complicar la comprensión del código (también son conocidas como instrucciones *Goto*).

A finales de los años sesenta surgió una nueva forma de programar sistemas de cómputo que no solamente daba lugar a programas fiables y eficientes, sino que además estaban escritos de manera que facilitaba su comprensión posterior.

Un Teorema denominado Dijkstra, realizado por él mismo (nombre adquirido por su creador; Edsger W. Dijkstra), demostró que todo programa puede escribirse únicamente con tres instrucciones de control [36]:

- Secuencia de instrucciones.
- Instrucción condicional.
- Iteración o bucle de instrucciones.

Con la programación estructurada es posible elaborar programas para diferentes equipos de cómputo con una labor que demanda: esfuerzo, creatividad, habilidad y cuidado. Sin embargo, con este estilo se puede obtener las siguientes ventajas:

- Los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia de arriba hacia abajo[2], sin necesidad de estar moviéndose de un sitio a otro en la lógica. La estructura del programa es clara puesto que las instrucciones están ligadas o relacionadas entre sí, por lo que es fácil comprender lo que realiza cada función.
- Reducción del esfuerzo en las pruebas. El programa se puede tener listo para producción normal en un tiempo menor del tradicional. Por otro lado, el seguimiento de las fallas (*depuración*) se simplifica debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir fácilmente.
- Disminución de los costos en el mantenimiento del software.
- Se facilita la utilización de las otras técnicas para el mejoramiento de la productividad en programación.
- Los programas quedan mejor documentados internamente.

El principal inconveniente de este método de programación es que se obtiene un único bloque de programa, que al momento en que se hace demasiado grande puede resultar problemático en su manejo. Esto se resuelve con la programación modular, con el uso de módulos interdependientes programados y compilados por separado; cada uno de los cuales ha podido ser desarrollado con programación estructurada. Sin embargo, esto lleva a otra problemática: la unión de módulos y el diseño que ellos presentan.

[2]También conocido en inglés como programación top-down.

2.8.2. Programación dirigida por eventos

La programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema o que ellos mismos provoquen [83, 32, 20, 73, 60, 69].

Para comprender a la programación dirigida por eventos podemos decir que es lo contrario a la programación secuencial, bajo el concepto de que el programador es quién definirá cuál será el flujo del programa. Lo que significa que en la programación dirigida por eventos será el propio usuario o el suceso quien accione el programa. Aunque en la programación secuencial puede haber intervención de un agente externo al programa, estas intervenciones ocurrirán en el momento, en el que el programador lo haya definido y no en cualquier momento como puede ser en el caso de la programación dirigida por eventos.

El creador de un programa dirigido por eventos deberá definir los eventos que manejarán su programa y las acciones que se realizarán al producirse cada uno de ellos, a lo que se conoce como el manejador de evento. Los eventos soportados estarán determinados por el lenguaje de programación utilizado, por el sistema operativo e incluso por eventos creados por el mismo programador.

En la programación dirigida por eventos la ejecución de un programa inicializa el código, en donde el programa quedará bloqueado hasta que se produzca algún evento. Al momento en que alguno de los eventos esperados por el programa tenga lugar, el programa pasará a ejecutar el código del correspondiente manejador de evento. Por ejemplo, si el evento consiste en que el usuario haga clic en el botón de *Siguiente* de un reproductor de películas, se ejecutará el código del manejador de evento, que será el que haga que la película se mueva a la siguiente escena.

2.8.3. Programación orientada a aspectos

La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente. Su intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos [92].

Con la POA se pueden capturar los diferentes conceptos que componen una aplicación. Las entidades deberán estar bien definidas de manera apropiada en cada uno de los casos y eliminar las dependencias inherentes entre cada uno de los módulos. De esta

forma, se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan comprensibles, adaptables y reusables.

Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así el término POA es usado para referirse a varias tecnologías relacionadas como los métodos *adaptivos*, los filtros de composición, la programación orientada a sujetos o la separación multidimensional de competencias.

En ocasiones, los programadores encuentran problemas que no se pueden resolver de una manera adecuada con las técnicas habitualmente usadas en la programación imperativa o en la programación orientada a objetos. Con estas técnicas, se ven forzados a tomar decisiones de diseño que repercuten de manera importante en el desarrollo de la aplicación y que nos alejan con frecuencia de otras posibilidades.

Por otra parte, la implementación de dichas técnicas a menudo implica escribir líneas de código que están distribuidas en gran parte de la aplicación; junto con los efectos que esto conlleva, como son: el tener dificultades de mantenimiento y desarrollo. Esto se conoce en el idioma inglés como *tangled code*, el cual se puede traducir como código enmarañado. El hecho es que existen ciertas decisiones de diseño que son complejas de implementar con las técnicas antes citadas.

Esto se debe a que ciertos problemas no permiten encapsular de igual forma que los que habitualmente se han resuelto con funciones u objetos. La resolución de éstos supone sea mediante la utilización de repetidas líneas de código por diferentes componentes del sistema o bien la superposición dentro de un componente de funcionalidades dispares.

La programación orientada a aspectos permite de una manera comprensible y clara la definición de aplicaciones con estos problemas. Por aspectos se entiende dichos problemas que afectan a la aplicación de manera horizontal y la forma en que este paradigma persigue el poder tenerlos de manera aislada, adecuada y comprensible.

Su consecución implicaría las siguientes ventajas:

- Un código menos enmarañado, natural y reducido.
- Mayor facilidad para razonar sobre los conceptos, debido a que están separados y las dependencias entre ellos son mínimas.
- Un código fácil de depurar y de mantener.
- Conseguir que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.

- Tener un código reusable y que se puede acoplar y desacoplar en cualquier momento.

2.8.4. Programación con restricciones

La Programación con restricciones es un paradigma, donde las relaciones entre las variables son expresadas en función de restricciones [37, 56]. Actualmente es usada como una tecnología de software para la descripción y solución de problemas de combinatoria, especialmente en las áreas de planificación y calendarización.

Este paradigma representa uno de los desarrollos interesantes en los lenguajes de programación desde 1990, y no es sorprendente que recientemente haya sido identificada por la ACM[3] como una dirección estratégica en la investigación en computación.

Básicamente es un paradigma de programación en el cual se especifica un conjunto de restricciones, las cuales deben satisfacer una solución en vez de especificar los pasos para obtener dicha solución.

La programación con restricciones se relaciona con la programación lógica. De hecho, cualquier programa lógico puede ser traducido en un programa basado en restricciones y viceversa. En muchas ocasiones los programas lógicos son traducidos a programas basados en restricciones, debido a que la resolución de un programa se puede desempeñar mejor que su contraparte.

La diferencia entre ambos se debe principalmente en sus estilos y enfoques en la percepción del mundo. Para ciertos problemas es más natural (y por ende simple) escribirlos como programas lógicos, mientras que en otros es más natural escribirlos como programas basados en restricciones.

El enfoque de la programación con restricciones se basa principalmente en buscar un estado con una gran cantidad de restricciones, para que éstas sean satisfechas simultáneamente. Un problema se define típicamente, como un estado de la realidad en el cual existe un número de variables con valor desconocido. Un programa basado en restricciones busca dichos valores para todas las variables.

Algunos dominios de aplicación de este paradigma son:

- Dominios booleanos, donde sólo existen restricciones del tipo verdadero/falso.
- Dominios en variables enteras y racionales.

[3] Asociación de Maquinaria Computacional

- Dominios lineales, en el que sólo describen y analizan funciones lineales.
- Dominios finitos, en donde las restricciones son definidas en conjuntos finitos.
- Dominios mixtos, los cuales involucran dos o más de los anteriores.

Los lenguajes basados en restricciones son típicamente incrustados en un lenguaje huésped. Este campo fue llamado inicialmente Programación Lógica con Restricciones y su primer lenguaje fue Prolog. Ambos paradigmas comparten características similares, tales como las variables lógicas (una vez que una variable es asignada a un valor, no puede ser cambiado) o backtracking.

En el desarrollo de este documento uno de los principales puntos de interés es la programación orientada a objetos, por ello, se presentará un análisis de los conceptos que comprenden este paradigma desde diferentes puntos de estudio.

2.9. Punto de vista sobre POO

Para el estudio de este trabajo se realizará una investigación de cada término utilizado en el paradigma orientado a objetos. Se iniciará con una introducción para comprender los puntos antes analizados, así como las coyunturas que puedan dar una alternativa a la problemática planteada. Adicionalmente, se pretende que el trabajo presente una congruencia con la forma de programación actual, por lo que se busca ser evolutivo e incremental en ese sentido o en su caso hacer mención de las consecuencias que origina un planteamiento de esta naturaleza.

En general, un objeto es algo que podría poseer características y relaciones. Por ende, un objeto en particular es básicamente un cuerpo material o una idea en particular [93, 97, 94]. Para tener una visión de la importancia del paradigma orientado a objetos se revisarán los siguientes puntos de vista.

- Perspectiva de Clasificación.
- Perspectiva Matemática.
- Perspectiva de Herencia.

2.9.1. Una perspectiva biológica y matemática

La clasificación de los objetos por su jerarquía evolutiva es un método comúnmente utilizado para intentar describir una jerarquía específica. Wegner ve esta analogía y la trata con las siguientes semejanzas desde el punto de vista biológico y la clasificación de los objetos [97].

- Ambos usan el primer orden de clasificación basado en individuos y se extiende a un segundo orden de clasificación basado en clases, por lo tanto, determina una estructura de clases jerárquica sobre su dominio de discurso.
- Los dos utilizan una perspectiva incremental e histórica al momento que discuten el dominio del problema. Esto proporciona una caracterización evolutiva más que funcional.
- La herencia biológica de genes es paralela a la herencia computacional de métodos.

Mientras los primeros dos puntos son por lo general entendidos y existen pocos desacuerdos, en el último punto el significado de campo o método posee una sutil diferencia. Para Wegner existen diferencias entre los esquemas de clasificación [97], los cuales se encuentran resumidos en:

- Mientras la evolución y la herencia son la hipótesis para explicar la relación entre especies, la herencia de tipos es un mecanismo de diseño.
- La evolución se da con mutaciones arbitrarias, mientras la herencia de tipos es un grupo y un diseño basado en una metodología conducida.
- Mientras los organismos heredan de sus padres, la herencia de tipos permite que las subclases elijan de quién ellos heredan.
- La herencia de tipos se da al nivel de clases mientras la herencia biológica se genera al nivel de individuos.

Para Martin Abadi existe poca visión en los primeros puntos del análisis de Wegner [97, 6]. El último punto puede ser argumentado con el punto de vista de que existen programas que pueden tomar un objeto individual, y hacer aquel objeto parte de sus antecesores en una rama de herencia. Esto es probablemente debido a la naturaleza, y puede ser dispensado como verdadero en un sentido general o popular, pero técnicamente incorrecto.

Wegner impulsa el modelo biológico para hablar *de la Clasificación por Clases de Equivalencia* [97]. Esto mismo sucede con Abadi y Cardelli [6] durante su discusión sobre subsuposición[4] (clases de equivalencia), en el momento en que ellos dicen: En efecto, "si a es un tipo de A y A es un subtipo de B entonces a es un tipo de B ".

$$if\ a : A \wedge A <: B\ then\ a : B$$

Wegner habla de abstracción de una clase con relación a otra [96]. Él usa un ejemplo de números enteros y propone un punto de vista ortogonal, en el que define tres tipos de equivalencias; las cuales podrían ser usadas para hablar de las semejanzas y las diferencias entre clases y objetos:

- Equivalencia de transformación: Captura la equivalencia computacional.
- Equivalencia de denotación: Captura la noción de expresiones en función de las denotaciones de sus componentes.
- Equivalencia de tipo: Donde la clase es del mismo tipo.

En la definición de la relación entre los tres tipos de equivalencia, Wegner postula que la relación puede ser declarada por la expresión siguiente:

$ET = EquivalenciaTransformacional$

$ED = EquivalenciaDenotacional$

$TE = Tipodequivalencia$

$$ET <: ED <: TE$$

Un ejemplo sería:

$$f(x) = (3 \times 3) = (9) = (Entero) \rightarrow todos\ son\ iguales$$

La suficiencia de una semántica operacional para capturar denotaciones requeridas es determinada por la noción de *consistencia*[5] y *completo*[6] . Lo que propone una lógica para seguir la discusión de denotaciones y cómo pueden pensarse los conceptos en la

[4]En inglés se conoce como subsumption
[5]*En inglés significa soundness.*
[6]*En inglés significa completeness.*

equivalencia denotacional. Abadi y Cardelli también hablan de los conceptos, pero tienen cuidado en encubrir el sentido de estos conceptos al momento en que ellos presentan relaciones con nociones de equivalencia. Wagner se refiere a la consistencia al decir que *todo lo demostrable es verdadero*, mientras que completo indica "todo lo verdadero es demostrable".

Al hacer uso de estas definiciones, la consistencia significa que el cálculo conserva la denotación y el significado completo de cualquier expresión que tiene el mismo valor, lo que puede ser transformado uno en el otro.

Las expresiones de $\lambda - cálculo$ son usadas para denotar reglas generales o cálculos, sin embargo son por lo general sin tipo. Wegner describe dos modos de llegar a una denotación de tipos desde una expresión carente de esta, cuando menciona: "*la clasificación es a menudo útil como un primer paso en el pensamiento y en el entendimiento o control de cualquier dominio de discurso...*". Esto se considera no relevante si se tiene una expresión escrita con tipos o sin tipos, ya que los tipos pueden ser deducidos. Los lenguajes de programación generalmente se encuentran basados en tipos y sin tipos, por lo que se pueden encontrar algunas de las siguientes características:

Los lenguajes basados en tipos presentan:

- Desventajas:
  - Sintaxis compleja.
  - Requiere disciplina.
- Ventajas:
  - Los programas son mejor estructurados.
  - Más legibles.
  - Eficientes.
  - Modificables.
  - Revisión de tipos en tiempo de compilación.

Los lenguajes que carecen de una representación de tipos básicos:

- Desventajas:
  - Semántica operacional y de denotación más compleja (definiciones).

  - Menos comprobación en tiempo de compilación.

- Ventajas:

  - Menos comprobación de sintaxis.
  - Menos disciplina de programación requerida.

2.9.2. Lógica y computación

En la primera parte se habló de la lógica y los conceptos de inferencia y la unificación. También se discutió cómo puede ser usado éste para clasificar clases sobre objetos en un mundo orientado a objetos. El siguiente tema a analizar es aquel visto desde la lógica en la computación.

Los predicados y el cálculo de los mismos tienen un interés especial en la relación de equivalencia, que clasifica objetos (a priori) o elementos de un dominio dentro de verdadero o falso. La inferencia de la regla "*modus ponens*" expresa esencialmente la preocupación que existe por la clasificación [95].

Por ejemplo:

- Sócrates es un estudiante (s es un S).
- Todo estudiante es una persona (S es un subconjunto de P).
- Implica que Sócrates es una persona (s es a P).
- Esta relación está basada en la inferencia que (s) pertenece a un súper conjunto (P) que tiene a un subconjunto (S).

2.9.3. Abstracción y Especificación

Cuando se trata con cálculo y sistemas grandes, se hace importante la noción de abstracción como una técnica para distribuir la información sobre un sistema complejo. Los métodos orientados a objetos usan esta técnica para tener clases generales (conceptos), que se especializan en objetos específicos (actores).

Existe una unión cercana entre clasificaciones y abstracciones, que se deriva del hecho de que las abstracciones determinan clases de equivalencia (las realizaciones) de las abstracciones [93].

2.9.4. Lenguajes de programación con tipos

Los lenguajes de programación pueden ser divididos en tres paradigmas basados en lenguajes con tipos [93, 98, 96]:

- Los lenguajes de estado que son vistos en computación como una secuencia de transformaciones, desde un estado inicial a un estado final (1950-1960).
- Los lenguajes de comunicación que ven el cálculo como mensajes que pasan entre módulos u objetos (los años 70).
- Los lenguajes de clasificación que ven él cálculo como algo incremental y clasificable o por la abstracción de la solución (1980 – 1990).

De lo anterior se puede inferir que:

$$Estado :> Comunicación :> Clasificación$$

En secciones anteriores se presentaron algunas de las características 2.7.2 que definen a la Programación Orientada a Objetos, así como la herencia de clases, esto es una base para clasificar a los objetos. También se analizó la ampliación de los sub-lenguajes basados en expresiones de tipos, que por lo regular son Clases y Herencia (Tipo base, Subtipos). La aplicación de esta definición de Herencia creará una clasificación estructurada en forma de árbol, de esta manera actúa como un mecanismo de clasificación.

Los objetos tienen: un conjunto de operaciones, memoria y la capacidad de comunicarse entre ellos enviándose mensajes. Las clases especifican una interfaz para los objetos. En el momento en que una clase no puede tener una instancia se le llama *Clase Abstracta*. La herencia de una clase es un mecanismo para formar las interfaces de una o varias clases heredadas con las reglas de la interfaz de clase base. Otra manera de definir a la Programación Orientado a Objetos es con los siguientes términos (algunos ya explicados con anterioridad).

- Objetos
- Clases (los tipos de los objetos)
- Herencia
- Fuertemente basado en tipos

- Abstracción de datos
- Concurrencia
- Persistencia

En este trabajo se pretende realizar que los objetos no solamente envíen mensajes, sino también que estos reaccionen a ciertos estímulos en función de otros objetos. Esto conlleva a estudiar la comunicación e interacción entre objetos, y a buscar las reglas que permitan realizar esto de manera simple. Uno de los puntos importantes para realizar estos estudios es la clasificación aunada con la comunicación.

2.9.5. Clasificación versus comunicación

La comunicación y la clasificación son puntos importantes en el paradigma de los lenguajes orientados a objetos. Se ha sostenido que la comunicación es la manera de pasar información entre un conjunto de objetos. Para realizar este proceso, existen muchos mecanismos para el paso de mensajes [93, 97]:

- Síncrono: RPC [7]
- Asíncrono: coroutines[8]
- Combinación: rendezvous [9]

La única exigencia es que los objetos deben ser capaces de comunicarse, pero que a su vez los objetos tengan una clasificación basada en clases. "*La programación orientada a objetos es preceptiva*[10] *en su metodología para clasificar objetos, pero es permisiva en su metodología para comunicarse*" [97].

Una diferencia importante en este punto es que la comunicación se muestra como una relación, mientras que una clasificación está dada por un nivel de definición. Si los mecanismos de clasificación como clases de herencia se basarán en función de realización, existirían muchas relaciones posibles. La clasificación es todavía fundamental a lo

[7]Del inglés Remote Procedure Call, Llamada a Procedimiento Remoto, que es un protocolo que permite a un programa ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos.

[8]Las coroutines son componentes de programas que generalizan subrutinas para permitir puntos de entrada múltiples y la suspensión y reanudar de la ejecución en ciertas posiciones.

[9]Es una palabra que viene del francés Rendez-vous que significa "la cita". (literalmente, venga usted)

[10]Mandato u orden que el superior hace observar y guardar al inferior o súbdito.

orientado a objetos que la programación misma. El paso de mensajes es fundamental también en la definición de objetos [6].

La herencia única no estricta aumenta la expresividad del lenguaje, pero a costa de la clasificación débil. Se habla en [93] que la herencia sin clases introduce mecanismos de clasificación. También se habla de las interfaces abstractas que son un refuerzo de la herencia estricta, en la cual los programadores confían en firmas o definiciones. Finalmente, se presenta la herencia múltiple, por lo que no es estricta en su forma expresiva sobre la herencia única [31].

Los verdaderos sistemas son raramente compatibles con sus precursores. La herencia estricta es por lo regular limitada, pero en la relación de esta clasificación de exigencia es débil. ¿Cómo puede uno guardar la clasificación sin la herencia estricta? La herencia estricta requiere que la relación entre el comportamiento de un subtipo y supertipo sea una relación "es un (is a)". Una relación que modela la semejanza "como", tiene la forma más débil en cuanto a la clasificación sin la herencia estricta. El uso simple de esta notación es $A \rightarrow B$, donde el grado de semejanza y de desacuerdo no es establecido. Se considera la relación como un conjunto de propiedades similares pudiéndose ampliar esta idea. Las semejanzas pueden ser definidas por un $A \rightarrow_x B$ si para algún $x\,[A\ es\ un\ X \wedge B\ es\ un\ X]$. El grado de semejanza entre tipos puede ser descrito en función de tamaños relativos a conjuntos semejantes. Como puede ser también definido en función de semejanzas y diferencias: $B \rightarrow A \Rightarrow C$ (A como B comparten C), $B \rightarrow A \Leftarrow C$ y (B como A excepto C). Otro método para tratar con la herencia no estricta es un caso especial donde los subtipos con cambios de comportamiento incompatibles, son tratados como excepción de clases.

En los lenguajes orientados a objetos donde no hay idea de la clase, la herencia usa el mecanismo de delegación. La delegación permite a los objetos realizar operaciones de su clase base. Esta herencia sin clases permite compartir los recursos dinámicos, pero la herencia sin clase puede ser una noción débil y menos estructurada. De este modo se gana flexibilidad y se deja a un lado la clasificación.

Con una interfaz abstracta los clientes de un módulo dependen únicamente de una definición y no sobre la lógica de implantación. El mecanismo primario es asegurar este concepto, y es que la interfaz o definición únicamente debe proporcionar acceso a métodos. Wegner propone idealmente se aplique a casos como subtipos, es decir, únicamente se sabe la firma o definición de los tipos, y no su contenido.

La herencia múltiple puede ser definida como un: subtipo T = que hereda de T_1, T_2,T_3,...,T_4, por lo que se tiene cuidado con la definición de nombres; entonces T es

la unión de todos los atributos heredados y locales, y a su vez T es la intersección de todos los supertipos T_n.

2.9.6. Formalismos matemáticos para tipos de datos

La teoría de tipos ocurre en varias áreas de aplicación de las matemáticas. Los tres acercamientos son: el cálculo, el álgebra y el cálculo de tiempo de compilación. Los acercamientos basados en el cálculo son usados para modelar el comportamiento de problemas en general.

El $\lambda - cálculo$ es un formalismo sintáctico para el cálculo de uso general. Esto es, el cálculo de lambda es una gramática definida que puede ser usada para definir muchos modelos semánticos.

Los acercamientos algebraicos semánticos son comportamientos más específicos. Eso significa que son modelos semánticos que expresan un comportamiento de un tipo específico. Con la especificación de álgebras clasificadas por orden, las nociones de subtipos y polimorfismo pueden ser introducidas.

El cálculo es un acercamiento axiomático para escribir un sistema con tipos. Los tipos pueden ser definidos por la especificación de un sistema formal o por reglas de inferencia.

2.9.6.1. ¿ Los Tipos deberían ser modelados por álgebras o cálculos?

Con base a lo anterior existe una pregunta, ¿los tipos deben ser modelados por cálculos? A fin de ayudar en la descripción algebraica de tipos y para una comparación a la noción de cálculo de tipos, Wegner examina y define la relación entre álgebras y cálculos.

Generalmente, un cálculo es un conjunto de reglas para contar o razonar; mientras un álgebra es usada para denotar el comportamiento abstracto de una clase de objetos. Un cálculo define reglas que son unidireccionales, con objetivos que finalizan con la aplicación de reglas. Es decir, en un sistema sintáctico simple la sustitución es la que gobierna a éste, y cuyo objetivo es reducir expresiones a alguna forma normal. Las álgebras son sistemas semánticos con destino especial que capturan comportamientos específicos. Esto es, son descripciones específicas de comportamientos que pueden ser observados dentro de un sistema.

Los cálculos y las álgebras están relacionados. Wegner describe esta relación como sigue: *"los Cálculos son álgebras (sintácticas) concretas, mientras que las álgebras son cálculos (semánticos) abstractos*". Para ver esto, primero considere un álgebra como

un sistema semántico que describe comportamientos específicos. Esta álgebra puede ser generalizada en un sistema sintáctico, con lo que es esencialmente un cálculo. A la inversa considere un cálculo como un sistema sintáctico de reglas de cálculo. Estas reglas pueden ser abstraídas a un conjunto de comportamientos; este conjunto abstracto de comportamientos es esencialmente un álgebra.

Hay una diferencia entre cálculos y álgebras en el modo que ellos modelan los tipos. El cálculo de lambda de segundo orden proporciona una base para modelar propiedades de un sistema con tipos en conjunto, mientras las álgebras multiclasificadas proporcionan una base para modelar propiedades de tipos individuales.

2.9.6.2. Características de los tipos

En esta parte final, se describe algunas cuestiones que provienen del estudio de tipos. Wegner trata de abordar el concepto de tipo desde un punto de vista filosófico. La pregunta fundamental es obviamente ¿Qué es un tipo? Sin embargo, a fin de contestar esto, él decide responder con la versión siguiente: *"¿Qué propiedades deben tener los tipos que permitan constituirse individualmente y colectivamente en un sistema de tipos aceptables para un lenguaje de programación orientada a objetos?".*

Wegner describe varios puntos de vista para contestar qué es un tipo o qué podría ser posiblemente un tipo. Ninguna de las ópticas parece capturar la esencia de lo que realmente es un tipo al momento en que uno ve la programación. Un ejemplo es hablar de tipos como clasificadores de valores lo que propicia un conflicto dentro de los contextos generales para el cálculo (tipos polimórficos).

Una dicotomía es la de los realistas con el intuicionismo. La vista realista del mundo confía únicamente en la verdad indicada, mientras la vista intuicionista se basa en pruebas juntadas de la observación para describir el mundo. Al relacionar éste con tipos y valores, los realistas ven valores simplemente como existencias y los tipos son un mecanismo usado para explicar, clasificar y manejar valores. Sin embargo, los intuicionistas ven tipos como un comportamiento básico, y los valores existen porque ellos pueden ser construidos a partir de algún tipo.

Existen varios puntos de vista dentro de los cuales se habla de tipos como clasificación y como ellos también pueden ser inferidos. Para nuestro estudio se utilizarán los dos puntos de vista. No obstante, en cada paso de la formalización se hará mención de manera explicita para dejar claro el momento en el que se utilizan los tipos.

En la próxima sección se describirán brevemente los principios básicos de la teoría

orientada a objetos tomados de forma inductiva. El primero de éstos se realizará de una manera funcional y la segunda será imperativa. Esto es con el objetivo de tener un antecedente formal, sobre el cual se basará el desarrollo del presente trabajo.

Capítulo 3

Marco Teórico

3.1. Resumen

En este capítulo se presentan los conceptos básicos para la comprensión de este trabajo, así como una descripción del paradigma orientado a objetos; con énfasis en el contexto de las diferentes formas evolutivas. Además, se mostrarán los esquemas y modelos utilizados para su representación y análisis en el diseño de programas.

3.2. Objetivos del capítulo

- Definir los conceptos básicos para la comprensión de la teoría orientada a objetos.
- Presentar un panorama general sobre algunos paradigmas orientados a objetos.
- Mostrar los esquemas utilizados para la representación formal de los paradigmas orientados a objetos.

3.3. Introducción a $\lambda - cálculo$

El cálculo lambda ($\lambda - cálculo$) es un lenguaje simple que permite la descripción de las funciones matemáticas y de sus propiedades; fue introducido por Church en los años 30 como fundamento de la matemática (funciones y lógica), y constituye un modelo formal. Muchos lenguajes funcionales [54, 70] son a menudo descritos como un súper $\lambda - cálculo$ o $\lambda - cálculo$ *extendido*. De hecho, los programas funcionales pueden ser traducidos bajo esta notación [14, 67].

Para los modelos de cómputo el calcular se convierte en la transformación de información de una manera implícita a una forma explicita, realizada de manera precisa y formal. Los lenguajes que utilizan la programación funcional se encuentran basados en $\lambda - cálculo$, que es un modelo computacional. Los lenguajes de programación imperativos por lo regular se encuentran basados en las Máquinas de Turing, que también es otro modelo computacional.

Los conceptos fundamentales en los lenguajes de programación imperativos son:

- Almacenamiento.
- Variables - es posible modificar ("celdas de memoria").
- Asignación.
- Iteración.

En la programación funcional no se presentan los conceptos antes mencionados, en el caso de existir, esto sería en un sentido diferente. En la programación funcional se encuentran:

- Expresión.
- Recursión (diferente de la noción de iteración).
- Evaluación (diferente de la noción de ejecución).
- Al momento se evalúa una expresión matemática sin modificar algún área de código.
- Variable (en un sentido matemático).
- Entidad desconocida.
- Abstracción de un valor concreto.

3.4. Programación Funcional

Un programa basado en un lenguaje funcional es un conjunto de definiciones de funciones. El intérprete de este lenguaje evalúa las expresiones por medio de pasos basados en el cálculo y reducción de expresiones.

Un ejemplo de $\lambda - cálculo$ se muestra en (3.1):

$$\lambda x. * 2x \tag{3.1}$$

El cálculo mostrado en (3.1) se codificó en Haskell, lo que queda de la forma:

$$\backslash x \rightarrow 2 * x \tag{3.2}$$

Éste denota una función de un sólo argumento, tal que al objeto x se le asocia al objeto $*2x$. En (3.1) se observa que:

- El símbolo λ sirve para denotar funciones.
- El punto . se usa para separar el argumento (o variable ligada) del cuerpo de la función.
- En él se utiliza notación prefija.

Las expresiones del ejemplo anterior se les conocen como $\lambda - Abstracciones$ y son un caso particular de $\lambda - términos$ o $\lambda - expresiones$.

La variable ligada o asociada es *muda*. Si sustituimos el identificador de la variable por una nueva variable, se sustituye también por consiguiente en el cuerpo, y por lo tanto se realiza una $\alpha - conversión$. En la que se obtienen equivalencias alfabéticas, como en (3.3):

$$\lambda x. * 2x \xrightarrow{\alpha} \lambda y. * 2y \tag{3.3}$$

Una $\alpha - equivalencia$ con dos funciones significa que las dos funciones son la misma y por lo tanto se deben indentificar de la siguiente forma: Se escribirá $\equiv_\alpha$ en lugar de $\xrightarrow{\alpha}$ (o también directamente), con lo cual se puede observar en (3.4) algunos ejemplos:

$$\lambda x.x \equiv \lambda y.y \tag{3.4}$$
$$\lambda x.y \equiv \lambda z.y$$
$$\lambda x.x \neq \lambda z.y$$

Para introducir una igualdad en el $\lambda - cálculo$ (es decir, una teoría con igualdad) es necesario la $\alpha - regla$ o identificación sintáctica; el uso de ambas puede plantear problemas como se verá a continuación:

Una expresión de $\lambda - abstracciones$ puede contener otras $\lambda - abstracciones$, de tal manera que se puede generar una sintaxis básica que represente a dichas abstracciones lambda como se muestra en la Tabla 3.1.

$x, y, z, \ldots$	Variables
$\lambda x.M$ donde M es un término y x es una variable	Abstracción
MN donde M y N son términos	Aplicación

Tabla 3.1: Sintaxis básica para representar las abstracciones de lambda

Se observa en la Tabla 3.1 que las tres nociones son los fundamentos para cualquier modelo computacional. Por lo que es posible inferir una gramática (3.5), que se describe a continuación:

$$\Lambda ::= X \mid (\Lambda\Lambda) \mid \lambda X.\Lambda \tag{3.5}$$

Donde Λ (Lambda Capital) significa el conjunto de términos de lambda y X es una meta variable que se extiende sobre el conjunto de variables $(x, y, z, x_1, x_2, \ldots, x_n)$.

Un ejemplo del uso de (3.5) se puede observar en (3.6):

$$\lambda xy. * (+xy)2 \tag{3.6}$$

Codificado la expresión (3.6) en Haskell, queda de la forma (3.7):

$$\backslash x \longrightarrow \backslash y \longrightarrow (x + y) * 2 \tag{3.7}$$

Como nota, se observa que las $\lambda - abstracciones$ tienen un sólo argumento; si es necesario especificar varios se escriben en forma separada o bien se usa la forma compacta presentada en la expresión (3.6).

También se obtiene una $\lambda - expresión$ al aplicar un objeto a una función. De esta manera, si se aplica el objeto 3 a la expresión (3.1), se obtiene la expresión (3.8):

$$(\lambda x. * 2x)\, 3 \tag{3.8}$$

Si aparecen varios argumentos es conveniente realizar una asociación por la izquierda, de esta manera la siguiente (3.9) $\lambda - expresión$ es equivalente sintácticamente:

$$\lambda x.\lambda y.\lambda z.E\ x\ y\ z \equiv \lambda x.((\lambda y.(\lambda z.E)x)y)z \tag{3.9}$$

Si M y N son $\lambda - expresiones$ la combinación (MN) es una $\lambda - expresión$ que se llama aplicación. En la abstracción $\lambda - expresión$ la variable x se llama *variable ligada* y a E se le llama cuerpo de la abstracción.

Convenio 3.1 *Se sobrecargará el significado de (igualdad sintáctica) con:*

1. *Los paréntesis más externos no se escriben.*
2. *La abstracción es asociativa por la derecha:*
   - $\lambda x_1 x_2 \ldots x_n.M \equiv \lambda \vec{x}.M \equiv \lambda x_1.(\lambda x_2.(\lambda x_3.(\ldots(\lambda x_n.M)\ldots))$
3. *La aplicación es asociativa a la izquierda:*
   - $MN_1N_2 \ldots N_n \equiv (\ldots((MN_1)N_2)\ldots N_n)$

Con el Convenio señalado en 3.9 se puede tener las siguientes igualdades sintácticas:

- Eliminación de paréntesis externos:

$$\lambda x.(+2x) \equiv (\lambda x.(+2x))$$
$$\lambda x.x \equiv (\lambda x.x)$$

- Asociación por la derecha para la abstracción:

$$\lambda xy.yx \equiv \lambda x.(\lambda y.(yx))$$
$$\lambda xy.yx \equiv \lambda x(\lambda y(yx))$$

- Asociación por la izquierda para la aplicación:

$$\lambda x.x\ y\ N \equiv \lambda x.((xy)N)$$
$$\lambda x.x(yN) \equiv \lambda x.(x(yN))$$
$$(\lambda x.x)yN \neq \lambda x.x\ yN$$
$$(\lambda x.x)yN \equiv ((\lambda x.x)y)N$$

3.4.1. Variables libres y ligadas

Las apariciones (ocurrencias) de variables en una expresión pueden darse de tres formas:

1. Ocurrencias de ligadura[1]
2. Ocurrencias ligadas[2]
3. Ocurrencias libres [3]

Las variables de ligadura son aquellas que están entre el símbolo λ y el punto (.). Por ejemplo, siendo E una expresión lambda como:

$$(\lambda xyz.E)$$

Las ligaduras serían xyz.

Las ocurrencias ligadas de una variable están definidas recursivamente sobre la estructura de $\lambda - expresión$, de esta manera:

1. En expresiones de la forma V, donde V es una variable, es una ocurrencia libre.
2. En expresiones de la forma $\lambda V.E$, las ocurrencias son libres en E salvo aquellas de V. En este último caso las V en E se dicen ligadas por λ antes de V.

[1]En inglés binders
[2]En inglés bound occurrences
[3]En inglés free occurrences

3. En expresiones de la forma (EE'), las ocurrencias libres son aquellas ocurrencias de E y E' .

Las $\lambda - expresiones$ tales como $\lambda x.(xy)$ no definen funciones porque las ocurrencias de y se encuentran libres. Si la expresión no tiene variables libres, se dice que es cerrada.

Definición 3.1 *Conjunto de variables libres.*

El conjunto de variables libres está denotado por $FV(E)$, donde E es cualquier $\lambda -$ expresión y donde las variables de E aparecen libres, éste se define inductivamente de la siguiente manera:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \end{aligned}$$

Un ejemplo de la Definición 2 se puede observar en (3.10):

$$FV(\lambda xy.x) = FV(\lambda x(\lambda y.x)) = FV(\lambda x.y) - \{x\} = 0 \tag{3.10}$$

En su contraparte existen variables ligadas o asociadas, y para ello la definición es la siguiente:

Definición 3.2 *Conjunto de variables ligadas o asociadas.*

El conjunto de variables libres está denotado por $BV(E)$, donde E es cualquier $\lambda -$ expresión y donde las variables de E aparecen asociadas, esta propiedad se define inductivamente de la siguiente manera:

$$\begin{aligned} BV(x) &= \{\} \\ BV(\lambda x.P) &= \{x\} \cup BV(P) \\ BV(PQ) &= BV(P) \cup BV(Q) \end{aligned}$$

3.4.2. Subtérminos y contextos

Definición 3.3 *Donde M es un subtérmino de $N (M \subseteq N)$ sí $M \in sub[N]$, en el que el conjunto $sub[N]$ está formado por todos los subtérminos de N y se define inductivamente de la forma que se describe en (3.19).*

$$\begin{aligned} sub[x] &= \{x\} \\ sub[\lambda x.N] &= \lambda x.N \cup sub[N] \\ sub[MN] &= sub[M] \cup sub[N] \end{aligned}$$

Definición 3.4 *Un contexto $C[.]$ es una $\lambda -$ expresión de la cual se extrae algún subtérmino; es decir, una $\lambda -$ expresión con un "*hueco*". Se define mediante la sintaxis BNF siguiente:*

$$\begin{aligned} Contexto ::= & [.] \\ & \mid var \\ & \mid (\lambda var.contexto) \\ & \mid (contexto\ contexto) \end{aligned}$$

Al colocar una expresión M en el "hueco" de un contexto $C[.]$ se obtiene una nueva $\lambda - expresión$ denotada con $C[M]$. Ciertas variables libres de una expresión M pueden quedar asociadas al *colocarlas* en un contexto.

Ejemplos:

$$C[.] = ((\lambda x.\lambda y.[.])E)F$$

Dada la expresión $M \equiv \lambda x.yx$ que tiene como variable libre a y como se muestra en (3.11).

$$FV(M) = FV(\lambda x.yx) = y \tag{3.11}$$

Si se agrega el término M en el contexto $C[.] \equiv \lambda xy.[.]$ se infiere a (3.12):

$$FV(C[M]) = FV(C[\lambda x.yx]) = FV(\lambda xy.(\lambda x.yx)) = 0 \tag{3.12}$$

Todo lenguaje tiene su semántica por lo que es importante analizarla de manera formal. Para ello, se utilizan diferentes formas de análisis, la más utilizada es la Semántica Operacional, la cual se analizará a continuación.

3.4.3. Semántica operacional

La evaluación de una expresión se compone de pasos de reducción, donde cada uno de los pasos se obtiene por reescritura. Tal evaluación permite describir la semántica operacional de un lenguaje funcional: El inicio está dado por un estado inicial (expresión inicial), mediante un cómputo (reescritura) se obtiene un estado final (expresión final) y se expresa mediante (3.13).

$$e \longrightarrow e' \tag{3.13}$$

Cada reducción de (3.12) en (3.13) reemplaza cierta subexpresión de acuerdo con ciertas reglas; tales subexpresiones se llaman *redexes* (expresión reducible). Se considera finalizado el cómputo al momento en que ya no aparecen más *redexes*.

Se llaman $\delta - reducciones$ a las reducciones con reglas que transforman constantes. Se describen con $\rightarrow_\delta$ (son predefinidas y caracterizan a las constantes).

La evaluación de la expresión $(+12)(-41)$ puede generar el siguiente cómputo:

$$\begin{aligned} &\Rightarrow * (+12)(-41) \\ &\quad \rightarrow_\delta *(+12)3 \\ &\quad \rightarrow_\delta *33 \\ &\quad \rightarrow_\delta 9 \end{aligned}$$

En $\delta - reducciones$ existen diferentes reducciones, una de las más utilizadas es la llamada $\beta - reducción$, y es el proceso de copia del argumento sobre el cuerpo de una abstracción, lo que remplaza todas las ocurrencias de la variables instanciables por el argumento:

$$(\lambda x.E)u \rightarrow_\beta [x := u]E \tag{3.14}$$

Con lo cual $\beta - reducción$ es equivalente a una sustitución en el cuerpo de la abstracción; la sustitución $[x : u]\, E$ se lee: *sustituir E en todas las apariciones libres de x por u.*

La regla de $\beta - reducción$ puede utilizarse también en el sentido opuesto, ver (3.15):

$$(\lambda x.E)\, u \leftarrow_{\beta} [x := u]E \tag{3.15}$$

En general la reducción de una $\lambda - expresión$ produce otra $\lambda - expresión$; una cadena de reducciones se describe en forma simplificada con el símbolo $\leftarrow$.

La reducción $\leftarrow_{\beta\delta}$ genera una relación $=_{\beta\delta}$ de igualdad, que es una relación de equivalencia (tiene la propiedad reflexiva, simétrica y transitiva); esta equivalencia debe verificar entre otras cosas que:

$$A \leftarrow_{\beta\delta} B \Rightarrow A \rightarrow_{\beta\delta} B \tag{3.16}$$

Se busca que la igualdad $=_{\beta\delta}$ sea una igualdad *sustitutiva*: Si dos términos u y v son iguales, al reemplazar en una expresión un subtérmino por expresiones iguales no se altera la igualdad, es decir:

$$u =_{\beta\delta} v \Rightarrow [F := u]E =_{\beta\delta} [F := v]E$$

3.4.4. Relaciones definibles en Lambda

Siendo R una relación binaria definida en (3.5):

$$R \subseteq \Lambda \times \Lambda \tag{3.17}$$

La expresión (3.17) se escribirá también xRy en el momento en el que $(x, y) \in R$. Entre las relaciones importantes para el presente trabajo se tiene:

$$\begin{aligned} \beta &= \{((\lambda x.M)N, [x := N]M) \mid M, N \in \Lambda\} \\ \eta &= \{(\lambda x.Mx, M) \mid M \in \Lambda\} \\ \beta\eta &= \beta \cup \eta \end{aligned}$$

Definición 3.5 *Una relación R es una relación compatible sí $\forall A, B, M, N \in \Lambda$, donde:*

$$(A, B) \in R \Rightarrow (MA, MB) \in R, (AN, BN) \in R, (\lambda x.A, \lambda x.B) \in R$$

Definición 3.6 *Una igualdad (sustitutiva) es una relación de equivalencia compatible.*

Definición 3.7 *Una reducción es una relación: compatible, reflexiva y transitiva (no necesariamente simétrica).*

Dada una relación R en Λ, se pueden definir de manera inductiva las siguientes relaciones:

Definición 3.8 *Cierre compatible de una relación $\rightarrow_R$ es el cierre compatible de R o reducción de un paso, es decir, la mínima relación que verifica los Axiomas 3.18, 3.19, 3.20, 3.21.*

$$\frac{(A, B) \in R}{A \rightarrow_R B} \tag{3.18}$$

$$\frac{A \rightarrow_R B}{MA \rightarrow_R MB} \tag{3.19}$$

$$\frac{A \rightarrow_R B}{AN \rightarrow_R BN} \tag{3.20}$$

$$\frac{A \rightarrow_R B}{\lambda x.A \rightarrow_R \lambda x.B} \tag{3.21}$$

Definición 3.9 *El cierre reflexivo y transitivo de una relación $\leftarrow_R$ es el cierre reflexivo y transitivo de $\rightarrow_R$, es decir, la mínima relación que lo verifica.*

Durante el desarrollo del actual documento se ha mencionado repetidas veces el uso de $\varsigma - cálculo$, este es una formalización sobre *Una Teoría Orientada a Objetos*. En la siguiente sección se explicará brevemente el uso de este cálculo.

3.5. Introducción al cálculo Sigma

Dentro de los paradigmas de programación existen dos áreas importantes: la primera es el paradigma funcional que se explicó en el apartado anterior y el segundo es el imperativo. Dentro de la formalización de $\varsigma - cálculo$ se presentan estas dos formas, en donde el autor de [6] trata de hacer una relación estrecha entre $\lambda - cálculo$ y $\varsigma - cálculo$. Así como presentar el cálculo $imp\varsigma - cálculo$ donde establece el uso de máquinas abstractas para establecer su funcionamiento de manera imperativa [2].

En esta próxima sección se describirán brevemente los principios básicos de la teoría orientada a objetos tomados de manera inductiva. El primero de éstos se realizará de una manera funcional y el segundo, será de manera imperativa. Esto es con el objetivo de tener un antecedente formal sobre el cual se basará el desarrollo del presente trabajo.

En $\lambda - cálculo$ existe un cálculo desarrollado para objetos llamado $\varsigma - cálculo$, que consistente en un conjunto mínimo de constructores sintácticos y reglas de cálculo.

En esta sección se mostrará de manera informal la estructura que compone dicho cálculo para objetos [6, 2, 5, 4]. En la siguiente Tabla 3.2 se presenta un resumen de la noción usada por los objetos y métodos.

$\varsigma(x)b$	Método $Self$ con parámetro x y cuerpo b.
$[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$	Un objeto con n métodos etiquetados de $l_1, \ldots, l_n$.
$o.l$	Invocación de un método l del objeto o.
$o.l \Leftarrow \varsigma(x)b$	Actualización del método l del objeto o con $\varsigma(x)b$.

Tabla 3.2: Resumen de la noción del paradigma orientada a objetos

Para más detalle, un objeto $[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$ es una colección de componentes $l_i = \varsigma(x_i)b_i$ para distintas etiquetas l_i y con asociación a los métodos $\varsigma(x_i)b_i$, para $i \in 1 \ldots n$; el orden de esos componentes no importa. La letra (sigma) es usada como una ligadura (*binder*) o enlace con un parámetro $Self$ (*self parameter*) de un método; el $\varsigma(x)b$ es un método con un parámetro $Self$ que tiene el nombre de x , éste se encuentra relacionado con el objeto anfitrión y un cuerpo b.

Una invocación de método es escrito de la forma $o.l$ donde l es una etiqueta de o. La intención es ejecutar el método llamado l de o con el objeto ligado al parámetro $Self$ y devolver el resultado de la ejecución.

Una actualización de un método se escribe de la forma $o.l \Leftarrow \varsigma(x)b$. La semántica de la actualización es funcional: una actualización produce una copia de o donde el método l es sustituido por $\varsigma(x)b$.

El ejemplo siguiente es un objeto que contiene dos métodos.

$$[l_1 = \varsigma(x)[], l_2 = \varsigma(x_2)x_2.l_1]$$

El primer método llamado l_1 regresa un objeto vacío [], El segundo método llamado l_2 invoca al primer método por medio del parámetro $Self$.

Los atributos no son parte del cálculo, pero un método que no hace uso de sus propias variables las considera como atributos. Un método que usa sus propias variables es entonces llamado un método apropiado. Con estas convenciones, la invocación de un método se convierte en la selección de los atributos, y la actualización de un método se convierte en la actualización de un atributo. Cabe señalar que es posible actualizar un atributo con un método apropiado, lo que convierte los datos pasivos a código activo. A la inversa se puede actualizar un método propio con un atributo.

Notación:

- $[\ldots, l = b, \ldots]$significa $[\ldots, l = \varsigma(y)b, \ldots]$, para una y no usada o asociada. Aquí $l = b$ es un atributo.
- $o.l \Leftarrow b$ significa $[o.l \Leftarrow \varsigma(y)b]$, para una y no usada, Aquí $o.l \Leftarrow b$ es la actualización de un atributo.

3.5.1. Las primitivas semánticas de $\varsigma - cálculo$

La ejecución de un término de $\varsigma - cálculo$ puede ser expresado como una secuencia en reducción de pasos [5, 6]. Se llamará a esto primitivas semánticas: Esto representa simple y directamente una posible semántica intencionada para los objetos. A fin de definir las primitivas semánticas, se introdujo la siguiente notación.

- Se usará una notación de indexación de la forma $\phi_i^{i\in 1\ldots n}$ para denotar una secuencia $\phi_1, \ldots, \phi_n$.
- La notación $b \twoheadrightarrow c$ significa que b reduce a c en un paso.
- La sustitución de un término c para los acontecimientos libres de x en b es escrita de la forma $b\{x \leftarrow c\}$.

Las primitivas semánticas y las reducciones de pasos para las operaciones de $\varsigma - cálculo$ son las siguientes:

Dado el objeto: $o \equiv [l_i = \varsigma(x_i)b_i^{i\in 1\ldots n}]$ con l_i distintas se obtiene:

- Invocación a la reducción para $o.l_i \xrightarrow{t} b_j\{x_j \leftarrow o\}$ para $j \in 1\ldots N$.
- Reducción de actualización $o.l \Leftarrow \varsigma(y)b \xrightarrow{t} [l_j = \varsigma(y)b, l_i = \varsigma(x_i)b_i^{i\in(i\ldots n)-\{j\}}]$ para $j \in 1\ldots N$.
- Una invocación del método $o.l$ reduce el resultado por la sustitución del objeto huésped para el auto parámetro $Self$ en el cuerpo del método llamado l_j. Note que la semántica de invocación es definida en forma abstracta por la sustitución.
- Una actualización de método $o.l \Leftarrow \varsigma(y)b$ reduce a una copia del objeto anfitrión, donde el método actualizado ha sido remplazado por una actualización.

Ejemplos de reducción:

El siguiente ejemplo de reducción es para un objeto con un simple método llamado l; el cuerpo de este método es un objeto vacío []. Se usará $\stackrel{\triangle}{=}$ para decir *igual por definición* y $=$ para decir *sintácticamente igual*.

Dado: $o_1 \stackrel{\triangle}{=} [l = \varsigma(x)[]]$

$$\begin{array}{ll} \text{entonces se tiene} & o_1.l \xrightarrow{t} []\{x \leftarrow o_1\} \equiv [] \\ \text{además} & o_1.l \Leftarrow \varsigma(x)o_1 \xrightarrow{t} [l = \varsigma(x)o_1] \end{array}$$

Dado: $o_2 \stackrel{\triangle}{=} [l = \varsigma(x)x.l]$

$$\text{entonces se tiene} \quad o_2.l \xrightarrow{t} x.l\{x \leftarrow o_2\} \equiv o_2.l \xrightarrow{t} \ldots$$

Dado: $o_3 \stackrel{\triangle}{=} [l = \varsigma(x)x]$

$$\text{entonces se tiene} \quad o_3.l \xrightarrow{t} x\{x \leftarrow o_3\} \equiv o_3$$

Dado: $o_4 \stackrel{\triangle}{=} [l = \varsigma(y)(y \Leftarrow \varsigma(x)x)\}$

$$\text{entonces se tiene} \quad o_4.l \xrightarrow{t} (o_4.l \Leftarrow \varsigma(x)x) \xrightarrow{t} o_3$$

3.5.2. Sintaxis de $\varsigma - cálculo$

Se describirá una *Teoría de Objetos Primitivos* la cual muestra los cálculos de objetos simples de manera formal [6]. Esto permite mostrar reglas de sobreescritura (*override*) y subsuposición (*subsumption*), lo que proporciona una visión amplia basada en un cálculo sin tipos [46, 7, 10, 74]. Se ilustrará la expresividad de los cálculos realizados y la solución a diferentes problemas.

El $\lambda - cálculo$ con tipos se utiliza para representar de manera formal los lenguajes basados en funciones dentro de un dominio de discurso. Estos lenguajes son conocidos como *lenguajes funcionales*, pero si se intenta modelar un lenguaje Orientado a Objetos, el principio de $\lambda - cálculo$ puede ser utilizado para definir el comportamiento de los tipos y objetos.

Con base a lo anterior se pretende estudiar las propiedades intrínsecas de los objetos para desarrollar un cálculo basado en objetos que sean simples en vez de luchar con las codificaciones complejas de objetos como los términos de $\lambda - cálculo$. Para ello, se han propuesto realizar reglas para objetos primitivos y un concentrado de reglas semánticas que se deberán respetar en el diseño del lenguaje.

Se investigó el cálculo que apoya la realización de la sobreescritura de métodos con la presencia de subsuposición de objetos [8, 79]. En donde la subsuposición es la capacidad de emular un objeto mediante otro objeto que tiene más métodos especializados. Y el sobreescribir es la operación que modifica el comportamiento de un objeto o clase, por lo que se sustituye uno de sus métodos mientras que los otros son heredados [31, 61, 88].

Todos los lenguajes comunes que hacen uso de objetos permiten la combinación entre sobreescribir y subsuposición, y la mayoría lo presenta de una manera correcta. Sin embargo, la corrección de tipos a menudo es realizada bajo condiciones bastante sutiles. Se espera enfocar el origen de algunas de estas condiciones e ilustrar las habilidades que se usan en el nivel semántico y teórico de tipos. Para esto se propone el mostrar aquellos rasgos que dan origen al cálculo llamado sigma.

Este documento se inició con un desafío: encontrar un sistema para un cálculo con objetos y sin tipos. La respuesta a este reto se torna interesante en virtud de que el cálculo presentado está basado en objetos, debido a que contiene un manejo de construcción, tipos $Self$ y la característica semántica de la invocación a métodos; así como la sobreescritura de los mismos. El cálculo mostrado es simple, con solamente cuatro formas sintácticas y aun sin funciones, este puede codificar a $\varsigma - cálculo$ sin tipos y puede expresar ejemplos más elaborados de una manera directa.

Como en el cálculo puro llamado $\lambda - cálculo$, el cálculo de objetos está basado en $\varsigma - cálculo$. Este último consiste en un conjunto mínimo de constructores sintácticos y reglas. En esta sección se explicarán dichas primitivas de manera informal que son el fundamento básico para desarrollar el tema propuesto.

Se comienza por la semántica operacional de un cálculo sin tipos a máquina orientada por objetos, con lo que se busca tener presente que en algún futuro se escribirá una máquina para tales exigencias. Nuestro objetivo en esta sección es el de definir una semántica directa de objetos, considerándolos como primitivos.

3.5.3. Las primitivas de $\varsigma - cálculo$ imperativo y sin tipo

Los objetos son únicamente estructuras computacionales para nuestro cálculo. Un objeto es simplemente una colección de atributos identificados, todos los atributos son métodos. Cada método tiene variables ligadas que se representan y un cuerpo que produce un resultado. La única operación en objetos son la invocación de métodos y la actualización de los mismos.

La siguiente sintaxis formal describe un objeto de cálculo puro sin funciones.

$a, b :=$	Términos.
x	Variables.
$[l_i = \varsigma(x_i)b_i{}^{i\in 1\ldots n}]$	Objetos con l_i distintas.
$a.l$	Selección de un atributo / invocación a un método.
$o.l \Leftarrow \varsigma(x)b$	Actualización de un atributo / sobrescritura de un método.

Tabla 3.3: Expresiones del imperativo $\varsigma - cálculo$ sin tipos

Notación:

- un objeto $[l_i = \varsigma(x_i)b_i{}^{i\in 1\ldots n}]$ tiene métodos llamados l_i y métodos $\varsigma(x_i)b_i$, donde x es la variable $Self$ en el cuerpo b.
- $o_j.l \Leftarrow b$ soporta $o_j.l \Leftarrow \varsigma(x)b$ para una y no asociada. Se llamará $o.l_j = b$ que es una operación de actualización.
- $[\ldots, l = b, \ldots]$soporta $[\ldots, l = \varsigma(y)b, \ldots]$ para una no usada. Se llamará $l = b$ como un atributo.
- Se identificará $\varsigma(x)b$ con $\varsigma(y)(b\{x \leftarrow y\})$ para cualquier ocurrencia que no sea libre en b.

Para completar la sintaxis formal de $\varsigma - cálculo$ se dará la definición de variables libres FV (por sus siglas en inglés) y sustituciones ($b\{x \leftarrow a\}$) en función de sigma.

$$
\begin{aligned}
FV(\varsigma(y)b) &\triangleq FV(b) - \{y\} \\
FV(x) &\triangleq \{x\} \\
FV([l_i = \varsigma(x_i)b_i{}^{i\in 1\ldots n}]) &\triangleq \cup^{i\in 1..n} FV(\varsigma(x_i)b_i) \\
FV(o.l) &\triangleq FV(o) \\
FV(o.l \Leftarrow \varsigma(y)b) &\triangleq FV(o) \cup FV(\varsigma(y)b)
\end{aligned}
$$

Tabla 3.4: Variables libres en $\varsigma - cálculo$

$$
\begin{aligned}
(\varsigma(y)b\{x \leftarrow a\} &\triangleq \varsigma(y')(b\{y \leftarrow y'\}\{x \leftarrow a\}) \\
&\quad para\ y' \notin FV(\varsigma(y)b) \cup FV(a) \cup \{x\} \\
x\{x \leftarrow a\} &\triangleq a \\
y\{x \leftarrow a\} &\triangleq y \quad Para\ y \neq x \\
[l_i = \varsigma(x_i)b_i{}^{i\in 1\ldots n}]\{x \leftarrow a\} &\triangleq [l_i = \varsigma(x_i)b_i\{x \leftarrow a\}^{i\in 1\ldots n}] \\
(o.l)\{x \leftarrow a\} &\triangleq (o\{x \leftarrow a\}.l \\
(o.l \Leftarrow \varsigma(y)b)\{x \leftarrow a\} &\triangleq (o\{x \leftarrow a\}.l \Leftarrow \varsigma(y)b\{x \leftarrow a\}
\end{aligned}
$$

Tabla 3.5: Reglas de sustitución en $\varsigma - cálculo$

Notación:

- Un término cerrado es un término sin variables libres.
- Se escribirá $b\{x\}$ para resaltar que es posible que x ocurra como variable libre en b.
- Se escribe $b\langle c\rangle$, en lugar de $b\langle x \leftarrow c\rangle$, en el momento en el que $b\{x\}$ está presente en el mismo contexto.
- Se identifica $\varsigma(x)b$ con $\varsigma(y)(b\langle x \leftarrow y\rangle)$ para cualquier y que no posea variables libres en b, (Por ejemplo: $\varsigma(x)x$ y $\varsigma(y)y$ con el mismo método).

- Se identifican dos objetos cualesquiera que difieran únicamente en el orden de sus componentes. (Por ejemplo: $[l_i = \varsigma(x_i)b_i, l_2 = \varsigma(x_2)b_2]$ y $[l_2 = \varsigma(x_2)b_2, l_i = \varsigma(x_i)b_i]$ son el mismo objeto).

En la definición anterior se captaron las primitivas semánticas de los objetos. Este conjunto de definiciones está conformado por tres relaciones de reducción: reducción de un paso a nivel superior ($\rightarrowtail$), reducciones de un paso ($\rightarrow$) y la reducción de muchos ($\twoheadrightarrow$). Buscando simplificar términos de la forma [2, 101].

Definición 3.10 *Relación de reducción*

- *Se escribe* $a \rightarrowtail b$ *si para algún* $o \equiv [l_i = \varsigma(x_i)b_i{}^{i\in 1\ldots n}]$ *y* $j \in 1\ldots n$ *cualesquiera de ambos:*
  - $a \equiv o.l_j$ *y* $b \equiv b_j\{x\}$ o
  - $a \equiv o.l_j \Leftarrow \varsigma(x)c$ *y* $b \equiv [l_j = \varsigma(x)c, l_i = \varsigma(x_i)b_i{}^{i\in 1\ldots n-\{j\}}]$
- *Un contexto se escribirá* $C[-]$ *que es un término con un hueco, y* $C[d]$ *representa el resultado de colocar dentro del hueco el término* d *(posiblemente con algunas variables libres de* d*). Se escribirá* $a \twoheadrightarrow b$ *sí* $a \equiv C[a']$, $b \equiv C[b']$, *y* $a' \rightarrowtail' b$, *donde* $C[-]$ *es cualquier contexto.*
- *Se escribe* $\rightarrow$ *para la cláusula reflexiva y transitiva de* $\twoheadrightarrow$.

Teorema 3.1 *Church-Rosser*

Si $a \rightarrow b$ *y* $a \rightarrow c$ *entonces existe una* d *tal que* $b \rightarrow d$ *y* $c \rightarrow d$.

3.5.4. Ecuaciones

Se infiere una teoría de ecuaciones sin tipo para las reglas de reducción sin tipo. El propósito de definir esta teoría es el interés de capturar la noción de igualdad para los términos de sigma [6]. Se podría usar esta noción de *igualdad* al momento en el que se quiera decir que dos objetos tienen similar comportamiento. Para este fin se muestran reglas de reducción mediante ecuaciones y se agregan reglas: simétricas, transitivas y de congruencia.

En la siguiente tabla se muestran las reglas para una teoría de ecuaciones. Cada regla tiene un número de juicios arriba de la línea horizontal y un simple juicio de conclusión

debajo de ésta. Cada juicio tiene la forma $\vdash \Im$. Una premisa de la forma $\vdash \Im_i \forall i \in 1 \ldots n$ es una abreviación para n premisas $\vdash \Im_1 \ldots \vdash \Im_n$. En lugar de $j \in 1 \ldots n$; lo que indica que hay n reglas separadas una por cada $j \in 1 \ldots n$. Las abreviaciones usadas dentro de una regla son algunas veces dadas posteriormente al nombre de la regla misma.

Definición de las ecuaciones:

Ecuación 3.1 *Simétrica*

$$\frac{\vdash b \cup a}{\vdash a \cup b}$$

Ecuación 3.2 *Transitiva*

$$\frac{\vdash a \cup b \ \vdash b \cup c}{\vdash a \cup c}$$

Ecuación 3.3 *de x*

$$\frac{}{\vdash x \cup x}$$

Ecuación 3.4 *Objeto*

$$\frac{\vdash b_i \leftrightarrow b_i \, \forall i \in 1 \ldots n}{\vdash [l_i = \varsigma(x_i) b_i \, ^{i \in \ldots n}] \leftrightarrow [l_i = \varsigma(x_i) b_i \, ^{i \in 1 \ldots n}]} (l_i \; distintas)$$

Ecuación 3.5 *Selección*

$$\frac{\vdash a \leftrightarrow a'}{\vdash a.l \leftrightarrow a'.l}$$

Ecuación 3.6 *Actualización*

$$\frac{\vdash a \leftrightarrow a' \ \vdash b \leftrightarrow b'}{\vdash a.l \Leftarrow \varsigma(x) b \leftrightarrow a'.l \Leftarrow \varsigma(x) b'}$$

Ecuación 3.7 *Selección (Select)*

Donde: $a \equiv [l_i = \varsigma(x_i) b_i \{x_i\}^{i \in 1 \ldots n}]$

$$\frac{j \in 1 \ldots n}{\vdash a.l_j \leftrightarrow b_j \, \langle a \rangle}$$

Ecuación 3.8 *Actualización (update)*

Donde: $a \equiv [l_i = \varsigma(x_i)b_i \, ^{i\in 1\ldots n}]$

$$\frac{j \in 1 \ldots n}{\vdash a.l \Leftarrow \varsigma(x)b \leftrightarrow [l_j = \varsigma(x_j)b_j, l_i = \varsigma(x_i)b_i \, ^{i\in 1\ldots n-\{j\}}]}$$

La relación de igualdad ($\longleftrightarrow$) como se ha definido genera un paso en la reducción ($\rightarrow$). Por lo tanto, el teorema de Church-Rosser implica que si $\vdash b \longleftrightarrow c$ entonces existe una d tal que $b \twoheadrightarrow d$ y $c \twoheadrightarrow d$.

3.5.5. Semántica Operacional

Las reducciones y estudio de ecuaciones hasta ahora no imponen un orden de evaluación específica. A continuación se definirá un sistema de reducciones para términos cerrados dentro de $\varsigma - cálculo$. Este sistema de reducciones es determinístico.

La intención que se muestra es la de describir una estrategia de evaluación de la forma comúnmente utilizada en los lenguajes de programación. Una característica de dicha estrategia de evaluación es que es débil, en el sentido que no trabaja bajo ligaduras (binders). Este punto de vista significa que dado un objeto $[l_i = \varsigma(x_i)b_i \, ^{i\in 1\ldots n}]$ se difiere de simplificar el cuerpo b_i mientras l_i es invocado.

El propósito de un sistema de reducciones es el de reducir cualquier expresión cerrada para generar un *resultado*. Dicho resultado es en sí mismo una expresión; para un puro $\varsigma - cálculo$ se define un resultado para ser un término de la forma $[l_i = \varsigma(x_i)b_i \, ^{i\in 1\ldots n}]$. Por ejemplo, ambos $[l_i = \varsigma(x)[]]$ y $[l_2 = \varsigma(y)[l_1 = \varsigma(x)[]].l_1]$ son resultados.

Los pasos de reducción se denotan por $\rightsquigarrow$. Se escribirá $\vdash a \rightsquigarrow v$ par dar a entender que a se reduce a un resultado v o que v es el resultado de a. Esta regla es axiomatizada con las siguientes tres reglas.

Semántica operacional

Reducción 3.1 *De un objeto* $v \equiv [l_i = \varsigma(x_i)b_i \, ^{i\in 1\ldots n}]$

$$\frac{}{\vdash v \rightsquigarrow v}$$

Reducción 3.2 *Selección donde* $v' \equiv [l_i = \varsigma(x_i)b_i\{x\} \, ^{i\in 1\ldots n}]$

$$\frac{\vdash a \rightsquigarrow v \quad \vdash b_j \langle v' \rangle \rightsquigarrow v \quad j \in 1 \ldots n}{\vdash a.l_j \rightsquigarrow v}$$

Reducción 3.3 *Actualización*

$$\frac{\vdash a \rightsquigarrow [l_i = \varsigma(x_i) b_i \, {}^{i \in 1 \ldots n}] \quad j \in 1 \ldots n}{\vdash a.l_j \Leftarrow \varsigma(x) b \rightsquigarrow [l_j = \varsigma(x) b, l_i = \varsigma(x_i) b_i \, {}^{i \in \{1 \ldots n\} - \{j\}}]}$$

En la primera regla los resultados no se pueden reducir más. En la segunda regla, el orden para evaluar una expresión $a.l_j$ es de la forma siguiente: se debe calcular primero el resultado de a, revisar que este se encuentre en la forma $[l_i = \varsigma(x_i) b_i \{x\} \, {}^{i \in 1 \ldots n}]$ con $j \in 1 \ldots n$, y entonces se evalúa $b \langle [l_i = \varsigma(x_i) b_i \, {}^{i \in 1 \ldots n}] \rangle$. En la tercera regla, el orden de evaluación en una expresión $a.l_j \Leftarrow \varsigma(x) b$, se debe primero calcular el resultado de a, verificar que este se encuentre en la forma $[l_i = \varsigma(x_i) b_i \, {}^{i \in 1 \ldots n}]$ con $j \in 1 \ldots n$, y regresar $[l_j = \varsigma(x) b, l_i = \varsigma(x_i) b_i \, {}^{i \in \{1 \ldots n\} - \{j\}}]$. Nótese que no se calcula dentro de b o dentro de b_i.

Se observa que el sistema de reducciones es determinístico: Si $\vdash a \rightsquigarrow v$ y $a \rightsquigarrow v'$, entonces $v \equiv v'$. La próxima proposición dice que $\rightsquigarrow$ es consistente (*sound*) con respecto a $\twoheadrightarrow$.

Proposición 3.1 *Consistencia de los pasos de reducción*

$$\textit{Si } \vdash a \rightsquigarrow v \quad \textit{entonces } a \twoheadrightarrow v \quad \textit{y por lo tanto} \quad \vdash a \leftrightarrow v$$

La proposición anterior puede ser verificada por medio de inducción en la estructura de prueba $\vdash a \rightsquigarrow v$.

$\rightsquigarrow$ es completo con respecto a $\twoheadrightarrow$.

Teorema 3.2 *Complitud de los pasos de reducción*

Dado que el término a será cerrado y v su resultado.

Si $a \twoheadrightarrow v$ entonces existe una v' tal que $\vdash a \rightsquigarrow v'$.

Los pasos de reducción en $\varsigma - cálculo$ son análogos a los pasos de reducción realizados en $\lambda - cálculo$. En $\lambda - cálculo$ los pasos de reducción se realizan para simplificar una parte de la función de una aplicación mientras éste produce una abstracción. Ahora bien el argumento es sustituido dentro de la abstracción sin evaluación en el momento en que se realiza la *llamada por nombre* o después de la evaluación para *llamadas por*

valor. En $\varsigma - cálculo$ la distinción entre la llamada por nombre o la llamada por valor no es complejo. Por lo que se adoptará el uso de llamada por nombre debido al Teorema 3.2; esto es, por claridad y simplicidad.

Se ha analizado un sistema de reducciones para $\varsigma - cálculo$, ahora se procederá a analizar a éste de manera imperativa.

3.5.6. Análisis de un imp$\varsigma - cálculo$ sin tipos

En este apartado, se desarrollará un cálculo basado en el tratamiento de objetos imperativos. Los cálculos de objetos son formalismos en el mismo nivel de la abstracción que el de $\lambda - cálculo$, pero basado en objetos más que en funciones [6, 30, 2, 5, 4]. A diferencia de $\lambda - cálculo$, el cálculo de objetos está diseñado específicamente para clarificar características de los lenguajes orientados a objetos. Existe un gran espectro para realizar cálculos basados en objetos. Así como se encuentran una gran variedad de variantes en $\lambda - cálculo$.

El cálculo basado en objetos al igual que lambda, es un pequeño conjunto de construcciones sin tipo. Estos se encuentran concentrados dentro de un núcleo y con posibilidad a ser incrementados con un sistema de tipos sofisticados. Mientras las características de los lenguajes pueden ser realistamente modeladas; el sentido de lo compacto en un núcleo inicial puede dar una unidad para el cálculo.

En esta parte del capítulo se introduce un pequeño, pero expresivo cálculo imperativo. Este provee un mínimo conjunto en el cual se estudiará una semántica imperativa operacional y el conjunto de reglas que rigen los tipos para un lenguaje orientado a objetos.

El cálculo comprende: objetos, métodos de invocación, métodos de actualización, clonación de objetos y definiciones locales. En una búsqueda por minimizar, se toman objetos únicamente para ser colecciones de métodos. Los atributos son importantes también, pero ellos pueden ser vistos como un concepto derivado. Por ejemplo, un atributo puede ser observado como un método que no hace uso del parámetro $Self$. En este cálculo, los métodos son mudables, entonces se puede prescindir de los atributos.

Los principales constructores para el cálculo de objetos son los mismos objetos. Un objeto es una lista de nombre de métodos y de resultados de los métodos. La relación de subtipos entre objetos es soportada por la subsuposición, la cual permite que un objeto sea usado donde se espera un objeto con menos métodos. Las anotaciones permiten los subtipos flexibles y la protección de efectos laterales.

$[l_i = \varsigma(x_i)b_i\ ^{i\in 1...n}]$	Objetos con l_i distintas.
$a.l$	Selección de un atributo / invocación a un método.
$o.l \Leftarrow \varsigma(x)b$	Actualización de un atributo / sobrescritura de un método.
$clone(a)$	Clonación superficial.
$let\ x = a\ in\ b$	Asignación.

Tabla 3.6: Sintaxis del imperativo $\varsigma - cálculo$ sin tipos

A continuación se hará una descripción de los términos mostrados en la sintaxis de la Tabla 3.6.

- Un objeto es una colección de componentes $l_i = \varsigma(x_i)b_i$, con distintas etiquetas l_i asociadas a métodos de la forma $\varsigma(x_i)b_i$; el orden de estos componentes no es de importancia para el tema de estudio. Cada ligadura ς encapsula el parámetro $Self$ de un método, por lo que $\varsigma(x)b$ es un método con una variable x y un cuerpo b.

- La invocación de un método se realiza mediante el llamado a $a.l$ lo que provoca la evaluación de a seguido por la evaluación del cuerpo del método llamado l, con el valor de a enlazado o ligado con $Self$ como variable de un método.

- La actualización de un método se construye mediante $o.l \Leftarrow \varsigma(x)b$. Este simple método actualiza al término a, lo que remplaza el método llamado l con un nuevo método $\varsigma(x)b$ y regresa el objeto modificado. La forma general de actualizar un método es permitir agregar la habilidad de evaluar un término en un momento de la actualización y usar el resultado para una evaluación posterior.

- Una operación de clonación llamada $clone(a)$ que produce un nuevo objeto con las mismas etiquetas de a. A esta operación se le conoce como clonación superficial.

- El cálculo sin tipos $a.l \Leftarrow (y, z = c)\varsigma(x)b$ puede ser expresado en función de let y de un simple método de actualización, como: $let\ y = a\ in\ let\ z = c\ in\ y.l \Leftarrow \varsigma(x)b$.

La actualización de un método por medio de let es realizada de la forma siguiente:

$$a.l \Leftarrow \varsigma(x)b \triangleq a.l(y, z = y)\varsigma(x)b \quad donde\ y, z \notin FV(b)$$

$$let\ x = a\ in\ b \triangleq (val = \varsigma(y)y.val].val \Leftarrow (z, x = a)\varsigma(w)b).val$$

$$donde\ y, z, w \notin FV(b) \wedge FV(a)$$

$$a; b \triangleq let\ x = a\ in\ b\ donde\ x \notin FV(b)$$

Un aspecto importante al comparar los lenguajes imperativos con los funcionales radica en los atributos que un objeto puede tener, ya que el funcionamiento depende básicamente de asignaciones; mientras que el segundo carece de éstas respectivamente.

En el cálculo todos los componentes de un objeto contienen métodos, sin embargo, se puede codificar los atributos por lo que se hace uso del siguiente análisis: se escribe $[l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i\ ^{i\in\{1...n\}-\{j\}}]$ para un objeto donde $l_i = b_i$ son atributos y $l_j = \varsigma(x_j)b_j$ son métodos. También se puede escribir $a.l := b$ para realizar una actualización, y $a.l$ como se mencionó en la sección anterior para seleccionar atributos.

3.6. Semántica

El propósito de esta sección del documento es describir algunas de las principales ideas y métodos usados para la descripción semántica. Ilustrar los conceptos básicos en una aplicación y analizar los métodos para utilizarlos en el presente trabajo [48, 62, 65, 68, 72, 95, 100].

El término semántica se refiere a los aspectos del significado o interpretación del significado de un determinado símbolo, palabra, lenguaje o representación formal. En principio, cualquier medio de expresión (lenguaje formal o natural) admite una correspondencia entre expresiones de símbolos, palabras, situaciones o conjuntos de cosas que se encuentran en el mundo físico o abstracto, el cual puede ser descrito por dicho medio de expresión.

La semántica formal se encarga de especificar rigurosamente el sentido o comportamiento de programas o partes de hardware. La necesidad de realizar dicha formalización se debe a que esto puede revelar ambigüedades y complejidades sutiles en definiciones que aparentan ser evidentes, y esto puede formar la base para realización, análisis y verificación en pruebas particulares de correctud.

Para ello, se usará notación informal en la que se pueda representar algunos conceptos sobre la semántica. Se iniciará por distinguir algunas diferencias entre la sintaxis y la semántica en un lenguaje de programación. Donde la sintaxis es un concepto concerniente a la estructura gramatical de un programa, tal que un análisis sintáctico de un programa puede ser visto en (3.22):

$$z := x; x := y; y := z \tag{3.22}$$

Se tiene tres sentencias (3.22) separadas por un símbolo ';'. Cada sentencia tiene la forma de una variable seguida por un símbolo ':=' acompañada de una expresión, la cual es solamente otra variable.

La semántica se encarga del correcto significado gramatical de los programas. Lo que expresará que el sentido del programa deba cambiar los valores de las variables x e y (y se coloca z al valor final de y). Si se explicara esto más detalladamente se observaría la estructura gramatical del programa y las explicaciones del uso de los conceptos de:

- Secuencia de sentencias separadas por ';' y
- Una declaración que consiste en una variable seguida de ':=' y una expresión.

La explicación anterior puede ser formalizada de diferentes maneras. En esa sección se describirán tres maneras de realizar esto.

3.6.1. Semántica operacional

En informática la semántica operacional es una manera de dar significado a los programas de computadora de una forma matemáticamente rigurosa [72].

Para un lenguaje de programación la semántica operacional describe cómo un programa válido debe ser interpretado [62]. En el contexto de los programas funcionales esto sería como el paso final en una secuencia de reducciones del programa en cuestión.

Una manera común y rigurosa de definir la semántica operacional, fue dada por Gordon Plotkin en su documento de 1981: "*un acercamiento estructural a la semántica operacional*" [72]. En este documento se describe un sistema de transiciones de estados para un lenguaje. Dicha definición permite un análisis formal y hace posible el estudio de relaciones entre los programas. Las más importantes incluyen la simulación y la bisimulación, especialmente útiles en el contexto de la concurrencia [68].

Definir la semántica operacional a través de un sistema es definir una forma posible de transiciones. Esto generalmente es proporcionado por medio de un sistema de reglas de inferencia que definen las transiciones válidas en el sistema.

En el siguiente párrafo se propondrá un ejemplo donde se intercambien valores entre dos variables, para ellos se deberá tener en cuenta lo siguiente:

Ejecutar una secuencia de declaraciones separadas por ';' donde se ejecuta individualmente una declaración después de la otra y de izquierda a derecha.

Ejecutar una declaración que consiste en una variable seguida de ':=' y otra variable, se determina el valor de la segunda variable y lo adjudicamos a la primera variable.

$$\begin{aligned}\langle z := x; x := y; y := z\rangle\,[x \mapsto 5, y \mapsto 7, z \mapsto 0] &\implies \langle x := y; y := z\rangle \\ &\quad\ [x \mapsto 5, y \mapsto 7, z \mapsto 5] \\ &\implies \langle y := z\rangle\,[x \mapsto 7, y \mapsto 7, z \mapsto 5] \\ &\implies [x \mapsto 7; y \mapsto 5; z \mapsto 5]\end{aligned}$$

En el primer paso se ejecuta la declaración $z := x$ y el valor de z es cambiado por 5 mientras que x e y no cambian. El programa restante queda de la forma $x := y; y := z$. Después del segundo paso el valor de x es 7 y quedará en la parte izquierda $y := z$. El tercero y último paso del cálculo cambiará el valor de y por 5. Por lo tanto, el valor inicial de x e y tienen que ser cambiados, con el uso de z como variable temporal.

La explicación anterior muestra una abstracción de cómo el programa está ejecutándose en una máquina. Es importante observar que esto es en efecto una abstracción. En este ejemplo se omiten detalles de cómo se usan el registro de direcciones y de variables. La semántica operacional es independiente de la arquitectura de la máquina y de las estrategias de implantación. Una forma de realizar esto es mediante el uso de reducciones, un ejemplo de lo anterior se puede ver en la siguiente ecuación:

$$\frac{\dfrac{\langle z := x, s_0\rangle \to s_1 \quad \langle x := y, s_1\rangle \to s_2}{\langle z := x; x := y, s_0\rangle \to s_2} \quad \langle y := z, s_2\rangle \to s_3}{\langle z := x; x := y; y := z.s_0\rangle \to s_3}$$

Si se hace uso de las abreviaciones, se tiene:

$$\begin{aligned}s_0 &= [x \mapsto 5, y \mapsto 7, z \mapsto 0] \\ s_1 &= [x \mapsto 5, y \mapsto 7, z \mapsto 5] \\ s_2 &= [x \mapsto 7, y \mapsto 7, z \mapsto 5] \\ s_3 &= [x \mapsto 7, y \mapsto 5, z \mapsto 5]\end{aligned}$$

Otra manera de expresarlo es mediante la representación:

$$\langle z := x; x := y; y := z, s_0 \rangle \rightarrow s_3$$

3.6.2. Semántica denotacional

En informática, la semántica denotacional es un acercamiento a formalizar la semántica de los sistemas informáticos con las estructuras de los objetos matemáticos (llamados *denotations* o los *significados*) que expresan la semántica de estos sistemas [43, 65].

El acercamiento a la semántica denotacional fue desarrollado originalmente para ocuparse solamente de los sistemas definidos por un programa de computadora.

La semántica denotacional tuvo su origen en el trabajo de Christopher Strachey y de Dana Scott en los años 60. Donde Scott introdujo un acercamiento generalizado a la semántica denotacional basada en dominios (Abramsky y Jung 1994). Recientes investigaciones han introducido los acercamientos para tratar dificultades con la semántica de los sistemas concurrentes (Clinger 1981).

En la semántica denotacional se toma el efecto de la ejecución del programa y se modela éste por medio de funciones matemáticas:

- El efecto de una secuencia de declaraciones separadas por ';' como la mencionada en (3.22) es una composición funcional de los efectos de una declaración individual.
- El resultado de una declaración consistente de una variable seguida por ':=' y otra variable, es la función que da un estado que producirá un nuevo estado. Esto es como el concepto original mencionado, excepto que el valor de la primera variable de la declaración es igual a la segunda variable.

Para el ejemplo mostrado en la ecuación (3.22) se obtendrán funciones escritas de la forma $S\lceil z := x\rceil, S\lceil x := y\rceil, S\lceil y := z\rceil$, y para cada una de las declaraciones de asignación y el programa total, se tiene la función:

$$S\lceil z := x; x := y; y := z\rceil = S\lceil y := z\rceil \circ S\lceil x := y\rceil \circ S\lceil z := x\rceil$$

Se observa que el orden de las declaraciones tienen cambios debido a que se usa la notación para la composición de funciones $(f \circ g)\, s$ que significa $f\,(g\, s)$. Si se quiere

determinar el efecto de la ejecución del programa en un estado particular, entonces se puede aplicar la función a aquel estado y calcular el estado siguiente:

$$
\begin{aligned}
S\lceil z := x; x := y; y := z\rceil\,([x \mapsto 5, y \mapsto 7, z \mapsto 0]) & \\
\Longrightarrow (S\lceil y := z\rceil \circ S\lceil x := y\rceil \circ S\lceil z := x\rceil \quad & ([x \mapsto 5, y \mapsto 7, z \mapsto 0]) \\
\Longrightarrow (S\lceil y := z\rceil \circ S\lceil x := y\rceil\,(S\lceil z := x\rceil \quad & ([x \mapsto 5, y \mapsto 7, z \mapsto 0])) \\
\Longrightarrow (S\lceil y := z\rceil\,(S\lceil x := y\rceil \quad & ([x \mapsto 5, y \mapsto 7, z \mapsto 5])) \\
\Longrightarrow (S\lceil y := z\rceil \quad & ([x \mapsto 7, y \mapsto 7, z \mapsto 5])) \\
\Longrightarrow \quad & [x \mapsto 7, y \mapsto 7, z \mapsto 5]
\end{aligned}
$$

Nótese que son sólo manipulaciones de objetos matemáticos, no se está interesado en la ejecución del programa. La diferencia puede parecer pequeña para un programa con sólo secuencias de asignaciones y declaraciones, pero para programas con mayor sofisticación la definición y construcción es sustancial.

3.6.3. La semántica axiomática

La semántica axiomática es un acercamiento basado en lógica matemática para aprobar la corrección de los programas de computadora. Se relaciona de forma muy cercana a la lógica de Hoare [100, 50, 49, 25].

La semántica axiomática define el significado de una instrucción en un programa por su efecto sobre aserciones en el estado del programa [100]. Las aserciones son las declaraciones lógicas - predicados con las variables, donde las variables definen el estado del programa.

A menudo, el desarrollador está interesado en propiedades de corrección parcial de programas: un programa es parcialmente correcto sobre una precondición y una poscondición, siempre y cuando el estado inicial cumpla la precondición y el programa termine o encuentre un estado de aceptación. Entonces el estado final es garantizado para cumplir la poscondición. Para el programa de ejemplo se tiene la propiedad de corrección parcial.

$$\{x = n \wedge y = m\}\, z := x; x := y; y := z\, \{y = n \wedge x = m\}$$

Donde $x = n \wedge y = m$ es una precondición y $y = n \wedge x = m$ es la poscondición. Los nombres n y m son usados para "*recordar*" el valor inicial de x e y respectivamente. El estado $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$ satisface la condición previa por lo que se toma $n = 5$ y $m = 7$, que es donde se tiene demostrado la propiedad de exactitud parcial. Se puede deducir que si el programa se termina, entonces, éste lo hará en un estado donde y es 5 y x es 7. Sin embargo, la propiedad de exactitud parcial no asegura que el programa se terminará, aunque éste sea claramente el argumento para el programa de ejemplo mostrado anteriormente.

La semántica axiomática proporciona un sistema lógico para demostrar propiedades de exactitud parciales de programas individuales. Una prueba de la susodicha propiedad de exactitud parcial, puede ser expresada por el "árbol de prueba", siguiente:

$$\frac{\dfrac{\{p_0\}z := x\{p_1\} \quad \{p_1\}x := y\{p_2\}}{\{p_0\}z := x; x := y\{p_2\} \quad \{p_2\}y := z\{p_3\}}}{\{p_0\}z := x; x := y := y := z\{p_3\}}$$

Donde se ha usado las abreviaturas:

$$p_0 := x = n \wedge y = m$$
$$p_1 := z = n \wedge y = m$$
$$p_2 := z = n \wedge x = m$$
$$p3 := y = n \wedge x = m$$

Se puede ver al sistema lógico como una especificación de ciertos aspectos de la semántica. Esto por lo general no captura todos los aspectos, por la simple razón que todas las propiedades de exactitud parciales puestas en la lista de abajo pueden ser probadas.

$$\{x = n \wedge y = m\}z := x; x := y; := z\{y = n \wedge x = m\}$$
$$\{x = n \wedge y = m\}\ if\ x = y\ then\ skip\ else(z := x; x := y; y := z)\{y = n \wedge x = m\}$$
$$\{x = n \wedge y = m\}\ while\ true\ do\ skip\{y = n \wedge x = m\}$$

Es importante hacer notar que estas clases de semántica no son contradictorias. Son técnicas diferentes para objetivos diferentes, y hasta cierto punto para lenguajes de programación diferentes.

3.7. Modelos Concurrentes

En este apartado se describirán brevemente algunos antecedentes y formalizaciones utilizadas para modelar problemas concurrentes. Cabe señalar que esta sección se introduce como un estudio somero que amplia la perspectiva del paradigma propuesto, y no como una herramienta a la que se accede de manera directa. Sin embargo, es posible llevar ciertas formalizaciones del $\beta\varsigma - cálculo$, pero eso se propondrá como un trabajo futuro.

Dentro de los antecedentes teóricos se encuentran determinados modelos que se han desarrollado para tratar los problemas de concurrencia y los de paralelismo. Algunos de los primeros modelos utilizados, fueron: CSP[4] propuesto por C.A.R. Hoare y el de CCS[5] propuesto por Milner [25, 49, 11, 68, 67].

La versión teórica de CSP fue presentada en el libro de Hoare Communicating Sequential Processes, que fue publicado en 1985. En mayo de 2005, la teoría de CSP ha experimentado algunos cambios. La mayor parte fueron motivados por el advenimiento de las herramientas automatizadas para el análisis de procesos y la verificación de CSP. El texto definitivo en CSP actual es [81].

El cálculo de sistemas comunicantes o CCS es un lenguaje de especificación formal basado en el álgebra de procesos, para la formalización y el modelado de los sistemas discretos comunicantes.

CCS propone una notación textual y otra visual para representar la existencia dentro de un sistema que llama *proceso* y la definición de éstos. Los procesos son vistos como bloques herméticos que se comunican con el mundo externo o ambiente por medio de puertos bien específicos, que conforman lo que se conoce como interfaz del proceso. Los procesos definen su comportamiento, al mostrar explícitamente la secuencia entera de operaciones elementales que dicho proceso efectúa durante toda su existencia.

[4]Communicating Secuential Proceses.
[5]Calculus of Communicating Systems.

3.7.1. Introducción a $\pi - cálculo$

El $\pi - cálculo$ es una notación desarrollada originalmente por Robin Milner, Joachim Parrow y David Walker, como un avance sobre el CCS con el fin de proveer movilidad al modelado concurrente [82].

El $\pi - cálculo$ se encuentra ubicado dentro de la familia de los denominados cálculos de proceso, los cuales han sido utilizados para modelar los lenguajes de programación concurrente; del mismo modo en que el $\lambda - cálculo$ ha sido utilizado para modelar los lenguajes de programación funcional [82].

3.7.1.1. Sintaxis

Dado X un conjunto de objetos denominados *nombres*. Los procesos de $\pi - cálculo$ son construidos por la sintaxis siguiente: Donde x e y son algunos de estos X.

$$
\begin{aligned}
P ::= & \; x(y).P \\
& \mid \overline{x}\langle y \rangle .P \\
& \mid P \mid P \\
& \mid (vx)P \\
& \mid !P \\
& \mid 0
\end{aligned}
$$

Los nombres pueden estar asociados por las restricciones y por el prefijo constructor de entrada. El conjunto de nombres libres o asociados de un proceso $\pi - cálculo$ están definidos inductivamente como se muestra a continuación:

Los procesos 0 no tienen nombres libres ni nombres asociados.

Los nombres libres de $a\langle x \rangle .P$ son a, x, y los nombres libres de P. Los nombres asociados de $a\langle x \rangle .P$ son los nombres asociados de P.

Los nombres libres de $a(x).P$ es a y los nombres libres de P, a excepción de x. Los nombres asociados de $a(x).P$ es x y los nombres asociados de P.

Los nombres libres de $P \mid Q$ es los de P junto con los del Q. Los nombres asociados de $P \mid Q$ es los de P junto con los del Q.

Los nombres libres de $(vx).P$ son los de P, a excepción de x. Los nombres asociados de $(vx).P$ son x y los nombres asociados de P.

Los nombres libres de $!P$ son los del P. Los nombres asociados de $!P$ son los del P.

Donde P y Q son procesos, x es un nombre y π es un *prefijo*. El prefijo denota una acción atómica previa a un proceso. Esta acción puede ser el envío o recepción de un dato por un canal. Por ejemplo, el envío de un dato por el canal x se denota por $\overline{x}\langle z\rangle$. Por su parte, la recepción de un dato b por el canal c se representa por $c(b)$.

El proceso $P \mid Q$ denota la composición paralela de los subprocesos P y Q . Además, permite que P y Q se comuniquen. El proceso $!P$ representa la replicación de P; es decir, define la ejecución infinita de P. Un nombre puede estar restringido al contexto de un proceso particular. Este es el propósito de $(vx)P$, en donde el nombre x es local a P y solamente visible dentro de él. Finalmente, la selección no determinista entre una serie de procesos $\pi_i.P_i(i \in I)$ se representa como su sumatoria $\sum_{i\in I} \pi_i.P_i$. Esta selección depende de la comunicación de dichos procesos.

Dos procesos importantes pueden derivarse de la sumatoria: 0 y $\pi.P$. El primero representa la acción nula y es la abreviación de la sumatoria en el momento en que $I = 0$, mientras que el segundo se da sólo si hay un proceso involucrado en la selección.

3.8. Un cálculo para el paradigma basado en eventos

Se podría suponer que una solución utilizada para resolver los problemas señalados en el Capítulo 1, es con el uso del paradigma basado en eventos, sin embargo, y de manera de introducción las formas de operar son diferentes. Esta diferencia está fundamentada en que un paradigma basado en eventos no muestra una reducción lógica de orden cero. También la forma de realizar la actualización no es de manera clara. Estos problemas serán comentados con más detalle en el Capítulo 4.

En esta parte del documento se presentará una definición no formal del $\rho\varpi\varsigma - cálculo$ (pronunciado cálculo *rho-pi-sigma*) [83, 32, 20, 73, 69]. Este cálculo es una representación particular del $\rho\varpi\varsigma - cálculo$ con un soporte al manejo de eventos [83, 69]. La sintaxis presentada en este cálculo da soporte a subsuposición, por lo que los tipos son congruentes con el manejo de la semántica [6, 31]. Se realizaron algunas pruebas de reducciones para garantizar la seguridad en el manejo de los tipos [27, 26, 9].

Jeremiah seleccionó como base para el $\rho\varpi\varsigma - cálculo$ a causa de dos razones: La primera, él modela $\rho\varpi\varsigma - cálculo$ en un estilo de programación imperativa que resulta ser impres-

cindible [69, 73]. Esto es porque es deseable modelar eventos, debido a que en la práctica los manejadores de acontecimientos se usan en demasía. Otra de las características que le ayudó a elegir este cálculo es que apoya varias dinámicas como son: El registrar, el des-registrar, el publicar y el volver a publicar. Todas estas dinámicas se utilizan como efecto para realizar una semántica dinámica. En segundo lugar, $\rho\varpi\varsigma - cálculo$ fue seleccionado debido a que modela un estilo de programación orientado a objetos, que es útil por las siguientes razones: Primero, el estilo orientado a objetos promueve la reutilización y la integración [87]. La segunda proporciona referencias por medio de etiquetas de los miembros de un objeto [34, 64, 2].

Lo anterior otorga una facilidad de expresión basado en $\lambda - cálculo$ en donde dichas etiquetas serían ocultas bajo los encierros o se manejaría de manera global. Lo que daría una carencia en el manejo de la estructura necesaria para determinar cuál es el sistema posible de acontecimientos, y cómo esos acontecimientos se encuentran relacionados. Finalmente, el $\rho\varpi\varsigma - cálculo$ proporciona las bases para controlar los eventos, así como también proporciona un almacenamiento para manejar avisos.

3.8.1. Sintaxis y una semántica informal

En esta sección del documento se describirá cada una de las partes principales que componen al $\rho\varpi\varsigma - cálculo$. Lo dividieron a éste en cuatro partes, que son: La sintaxis del objeto, la abstracción de valores, la sintaxis de eventos y la sintaxis de los tipos. La sintaxis de los objetos es igual al mostrado en $\rho\varpi\varsigma - cálculo$ [69]. Este método provee una manera de crear objetos, usar métodos, actualizar métodos y clonarse; como se puede apreciar en el apartado de introducción a $\varsigma - cálculo$ [6, 47]. Para la sintaxis que permite el manejo de eventos, éste deberá publicar, des-publicar y manejar los registros de cada objeto. Esta sintaxis provee una manera de describir a los objetos, valores y expresiones en el cálculo.

3.8.2. Sintaxis de un objeto

El cómputo que se expresa en $\rho\varpi\varsigma - cálculo$ son las bases preliminares para la creación y manipulación de objetos. La sintaxis de objetos tiene etiquetas que denotan cada nombre de un método seguida por la definición del método como se ve en (3.23).

$$[l_1 = \varsigma(x)x.l_2 = \varsigma(y)[o_2 = \varsigma(p)[]]] \tag{3.23}$$

La parte de la declaración de un método llamada $\varsigma(x)$ es una especie de auto parámetro (*Self parameter*), que puede tener referencias en el cuerpo de un método.

$a, b, c, d :=$			
	x	Variable	Sintaxis de un objeto
	$[l_i = \varsigma(x_i)b_i \ ^{i\in 1...n}]$	Objetos con l_i distintas	
	$a.l$	Invocación a un método	
	$a.l \Leftarrow \varsigma(x)b$	Actualización de un método	
	$clone(a)$	Clonación	
	$let\ x = a\ in\ b$	Asignación	Abstracción de valor
	$\rho(a.l, b \equiv c)d$	Registro	Sintaxis de eventos
	$\neg\rho(a)$	Des-registro	
	$\varpi(a.l)$	Publicar	
	$\neg\varpi(a.l)$	Des-publicar	
$A, B, C ::=$			
	K	Tipo constante	Sintaxis de tipos
	Top	Tipo más básico	
	$[l_i : B_i \ ^{i\in 1...n}]$	Objeto con tipo	
	R	Registro de tipo	

Tabla 3.7: Sintaxis del cálculo de eventos

Dada la sintaxis anterior se puede escribir un ejemplo como el que se ve en (3.24).

$$\begin{aligned} &let\ cpy = [lc = \varsigma(x)[]]\ in \\ &\quad let\ a = [l = \varsigma(x)[]]\ in \\ &let\ reg = \rho(a.l, [] \equiv [])\ let\ val = a.l\ in\ cpy.lc \Leftarrow \varsigma(x)val\ in \\ &\ let\ zz = \varpi(a.l)\ in \\ &\qquad a.l \Leftarrow \varsigma(y)[l_2 = \varsigma(z)[]] \end{aligned} \tag{3.24}$$

Existen tres constructores sintácticos dentro de la definición de objetos que pueden ser usadas para manipular dichos objetos: La selección de un método, la actualización de un método y la copia de un objeto.

3.8.3. Abstracción de valor

Los valores de un cálculo pueden ser guardados por así llamarlo, para un uso posterior dentro de *let*. Por ejemplo, uno puede tener la expresión $let\ a = [l = \varsigma(x)[]]\ in\ clone(a).l$

$\Leftarrow \varsigma(y)[l_2 = \varsigma(z)[]]$. También es posible usar la expresión para que de forma secuencial sean compuestas dos expresiones. Por ejemplo $let\ x = a.l \Leftarrow \varsigma(y)[l_2 = \varsigma(z)[]]\ in\ a.l.l_2$, la actualización será completada antes que la expresión de selección $a.l.l_2$ sea evaluada.

3.8.4. Sintaxis de eventos y semántica informal

El uso del mecanismo de eventos en $\rho\varpi\varsigma - cálculo$ a través de publicaciones, registros y métodos de actualización; permite que cualquier método de un objeto en el $\rho\varpi\varsigma - cálculo$ se puede utilizar como evento. Las construcciones de la *publicación* esencialmente proporcionan una manera de permitir o rechazar los métodos que se utilizarán como eventos [69]. Por ejemplo, $\varpi(a.l)$ causaría que $a.l$ se publicará cada vez que sea actualizado, por otro lado $\neg\varpi(a.l)$ aseguraría que $a.l$ no se publicará en la actualización.

El registro de manejadores se hace con el uso de los constructores de ρ. El ejemplo anterior muestra un registro de un manejador para $a.l$. Cada vez que $a.l$ es actualizado el manejador actualiza $cpy.lc$ con un nuevo valor de $a.l$.

El $b \equiv c$ la parte de la construcción del registro proporciona una manera de limitar los avisos que el manejador manejará realmente.

Específicamente si b y c evalúan el mismo valor se permite al manejador dirigir avisos, si no, no se ejecutará el manejador. En el ejemplo se usa $[] \equiv []$ para permitir que el manejador siempre permita controlar el aviso a $a.l$.

La idea detrás del uso de $\equiv$ es el de realizar el manejo de acontecimientos solamente si la condición lo permite. Se podría ver superfluo como una condición de la forma $if - then$. Sin embargo, la separación de la condición del cuerpo del manejador permite que las condiciones puedan ser ejecutadas en diferentes momentos desde diferentes manejadores. Se hablará a continuación de la flexibilidad que permite definir la semántica operacional en una mejor manera, que si se manejara el $if - then$ [48, 62, 68, 72].

Para deshacer el registro de un método, se usa la sintaxis que provee el constructor $\neg p$. También la forma de realizar esto con éxito es hacerlo con anticipación. De lo contrario, se le pediría al sistema eliminar algo que no existe, pudiendo existir efectos de otro tipo o excepciones.

3.8.5. Sintaxis de tipos

La sintaxis de tipos para objetos es similar a la expresión utilizada por los objetos, completa con corchetes (“[“,”]”) y sus etiquetas [3, 8, 71]. Una diferencia es que las

etiquetas utilizadas están asociadas con un valor de regreso o retorno. Por ejemplo, el tipo de $[l = \varsigma(x)[]]$ es $[l : []]$. Igualmente, el *tipo* de un objeto tiene las mismas etiquetas que el objeto relevante, donde cada una de ellas tiene el tipo del resultado del método asociado a la etiqueta del objeto. El tipo R es el tipo del resultado de un registro (ρ). Los tipos constantes se denotan por K, que pueden ser usados por tipos que son definidos de manera nativa en los lenguajes; por ejemplo, enteros, caracteres, etc. Dichos tipos están incluidos para completar las expresiones de tipos que no están definidos en $\rho\varpi\varsigma - cálculo$. Puesto que el tipo mostrado permite el uso de subtipos, deberá existir uno llamado Top.

Capítulo 4

Paradigma Proactivo

4.1. Resumen

Este capítulo presenta un paradigma proactivo orientado a objetos, así como sus implicaciones en algunas áreas pertinentes a la programación orientada a objetos.

4.2. Objetivos del capítulo

- Dar una visión general del paradigma propuesto.
- Definir $\beta\varsigma - cálculo$ de manera general.
- Describir $\beta\varsigma - cálculo$ de manera funcional y sin tipos.
- Definir $\beta\varsigma - cálculo$ de manera imperativa con tipos.
- Establecer algunas implicaciones de $\beta\varsigma - cálculo$ en el área de la programación orientada a objetos.

4.3. Introducción

La programación orientada a objetos ha tenido énfasis en los últimos años en la realización de los sistemas basados en cómputo. Los lenguajes orientados a objetos fueron diseñados para proporcionar una intuitiva forma de ver los datos, así como el cómputo de una manera unida. Esto permite crear una representación entre el software y el mundo de los objetos físicos.

De talante intuitivo se puede observar que los objetos del mundo físico interactúan entre sí. Esta interacción se realiza de diferentes maneras, ya sea por eventos, secuencias o concurrentemente. En esta parte del trabajo se darán a conocer los conceptos básicos necesarios para desarrollar un paradigma de programación llamado $\beta\varsigma - cálculo$; en el que se involucra la teoría orientada a objetos. Si bien es posible decir que mucho de lo que se modela en el mundo es a través de objetos, también hay que tener en cuenta la forma en que éstos se relacionan o se ven afectados por su entorno. Este trabajo se enfocará a analizar esta interacción y cómo estos objetos pueden ser afectados. Esta forma de interacción se presentará mediante lógica de primer orden.

El término proactivo fue acuñado por Viktor Frankl [40]. Este término se ha llevado a diferentes áreas de la sociedad, desde lo administrativo hasta el campo de la computación; donde el sentido de esta palabra toma sus peculiaridades. Hoy, la investigación en el campo de la informática se encuentra enfocada a un modelo interactivo de cómputo, esto se debe a que las personas interactúan directamente uno a uno con sus computadoras. Mientras que en un futuro se visualizan varias computadoras por persona y con sistemas que faciliten el realizar esta manera de interacción [33, 89, 12].

Con el paradigma de cómputo proactivo se pretende que las computadoras se anticipen a las necesidades del usuario y que éstas permitan tomar decisiones a nuestro favor. Esto es, mientras las personas están trabajando, las computadoras interactuarán unas con otras en busca de la solución a algún problema. Lo que puede propiciar una proactividad en la actividad humana.

En el campo de la computación proactiva actualmente existen desafíos importantes que se deben resolver: la conexión física de millones de nodos, modelos de cómputo, lenguajes y paradigmas. En este trabajo la palabra *proactivo* o *proactiva* tiene que ver con la noción de estar a favor de la acción, más que el significado de qué hacer con la acción misma.

Los retos que se proponen en la computación proactiva nos hacen definir un paradigma de programación orientado a objetos. Esta propuesta de paradigma deberá permitir la

interacción de los objetos, lo que promueve la proactividad entre los sistemas.

Los paradigmas actuales de la programación orientada a objetos se preocupan por la clasificación jerárquica, esto nos lleva a otro planteamiento. Dentro de un modelo jerárquico existe la evolución de los objetos como lo describe Darwin. Desde un punto de vista particular, si un objeto de jerarquía superior es modificado, ¿los objetos derivados o de jerarquías inferiores dependientes serán modificados? Dentro de la etapa de diseño de cualquier sistema esto es admisible, pero en el momento en que se trata de algo ya implantado esto tiene connotaciones colaterales. Muchos lenguajes orientados a objetos han agregado a sus clases/objetos mecanismos para indicar que algunos métodos han dejado de operar o se encuentran derogados.

Estas soluciones favorecen en mucho al trabajo de la reingeniería, aunque dentro del enfoque proactivo estas situaciones no son favorables. Esto es un factor importante que promueve el análisis de los objetos basándose en ciertas condiciones y reglas. Estas reglas deben permitir que un objeto evolucione sin llegar a permear con los objetos de su entorno de manera directa en el diseño en su creación.

Por un momento imaginemos un sistema planetario como es el nuestro, donde existe un sol y varios planetas que giran entorno él. Si agregamos un objeto con suficiente masa en algún instante, muchos de estos planetas se verían afectados. Esto se debe a la fuerza gravitacional y a la atracción que existe entre ellos, este efecto también se da, si retiráramos algún planeta del sistema solar descrito. Si se le pidiera a un grupo de desarrollo realizar dicho modelo físico, tendría que utilizar una serie de abstracciones comúnmente conocidas como interfaces de diseño en programación. Lo que lleva a tener que preveer de alguna manera el posible comportamiento del modelo físico y conocer desde un inicio, las posibles condiciones en las que podría funcionar el sistema.

Se pretende proponer un lenguaje que nos permita modelar los sistemas físicos antes mencionados. La idea básica es que los objetos se agreguen y retiren del entorno en cualquier momento, con los efectos que se puedan desencadenar dentro del sistema. Esto tendrá dos ventajas principales en el diseño de software: El primero es que el desarrollo de modelo puede darse de manera incremental lo que permite dar una forma simple de evolución. La segunda es al momento de retirar cualquier objeto en cualquier momento sin afectar de manera directa al sistema; logrando con esto, una dependencia en el diseño de los objetos.

Para ver lo explicado en el párrafo anterior, primero se lleva a cabo un análisis de los lenguajes de programación orientados a objetos, donde se deben tomar en un sentido estricto para resolver estos problemas. Un ejemplo es el mostrado en el Capítulo 1, en

donde se menciona que existe un estanque t_1 de agua con tres sensores que detecta tres niveles (n_1, n_2 y n_3), donde $n_1$es el nivel bajo, n_2 es un nivel aceptable y n_3 nivel alto (peligro de desbordamiento). Se solicita realizar un sistema basado en objetos que simule dicho proceso físico, con la restricción que al momento en que el agua llegue al nivel n_3 se activa una alarma a_1.

Se presentó un bosquejo en la Figura 1.2 de una posible solución bajo los esquemas conocidos actualmente dentro de la programación orientada a objetos. Para mostrar cómo debe operar este paradigma se iniciará con un ejemplo simple. Éste se puede observar en la Figura 4.1, el cual involucra el problema mostrado en la Figura 1.1. Posteriormente en este Capítulo se propone dar las bases formales para el modelado de este paradigma propuesto. Por el momento y para analizar el diagrama de la Figura 4.1 bastará con decir que existe una relación entre dos métodos de clases diferentes. Esta relación se encuentra basada con operaciones bajo la lógica de orden cero.

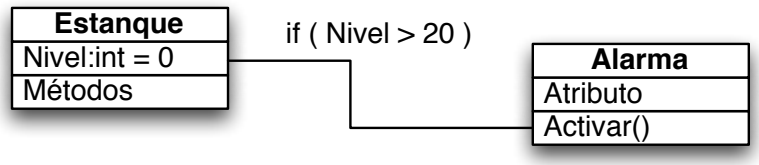

Figura 4.1: Solución gráfica al problema del estanque

En la Figura 4.1 se tiene al objeto denominado *Estanque* que realiza actualizaciones en su atributo *Nivel* y que el objeto *Alarma* se activará en el momento en el que el valor del *Nivel* sea mayor a 20. Es posible mostrar el problema en un código similar al de *Java* o *C#* el cual puede quedar como se observa en el Algoritmo 4.1. Este código no se puede implantar de manera directa en estos compiladores, pero sirve para mostrar de una manera general lo que se pretende hacer. No obstante, sus implicaciones, características y ventajas son descritas en secciones siguientes.

Algoritmo 4.1 Código abstracto para activar los objetos

```
Object Estanque
{
      int Nivel = 0;
      Sensar ()
      {
            Nivel = medir();
      }
}

Object Alarma
{
      if( Estanque.Nivel = 20)
            Activar();

      void Activar()
      {
             print("Alarma activada");
      }
}
```

Se puede observar en el Algoritmo 4.1 y específicamente en el objeto llamado *Alarma*, que existe una nueva regla o instrucción denominada *if*, la cual verifica que si el *Estanque* toma un valor de 20 se ejecuta la función *Activar* del objeto *Alarma*. Este diseño permite delegar la responsabilidad de activación al objeto cliente (que es *Alarma*); mientras que el objeto *Estanque* es independiente de cualquier relación explícita en el diseño. Esto dará mayor flexibilidad y expresividad al diseñar las clases u objetos. Debido a que el diseñador establece las condiciones que activan a los objetos en estudio. Por otro lado, también se pueden realizar actualizaciones a los objetos que dependen de su entorno o contexto. Esto último promueve un sentido a la evolución libre de las activaciones.

4.4. Definición de un paradigma proactivo orientado a objetos

Para realizar una definición a la propuesta de este documento se han tomado algunos aspectos de $\varsigma - cálculo$. Lo que ha originado una definición que se llamará $\beta\varsigma - cálculo$. En esta última, se definirá una función denominada *aif*[1] que se localiza dentro de la

[1]Se le ha dado el nombre de *aif* por la contracción de *Activar Si*

definición de un objeto, así como la especificación del método de actualización. Este permite activar los objetos cercanos o dentro de su contexto. Es posible crear un método similar a *if* con base al $\varsigma - cálculo$, pero existe un motivo para separarlos y es el llamado *activador*; además de las implicaciones en el diseño de un sistema de cálculo. Al sistema descrito se le llama de *evolución débil*, en el sentido que no se reducen los cuerpos de los métodos.

Para $\beta\varsigma - cálculo$ existe un cálculo basado en objetos que consiste en un conjunto mínimo de constructores sintácticos y reglas de cálculo. En esta sección se mostrará de manera informal la estructura que compone dicho cálculo para el paradigma proactivo orientado a objetos, propuesto en este trabajo.

La Tabla 4.1 presenta un resumen de la noción usada por los objetos proactivos y sus métodos.

Expresiones	Descripciones
$\varsigma(x)b$	Método $Self$ con parámetros x y cuerpo y.
$[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}$, $m_j = \varsigma(x_j)if(c_j)a_j : b_j\ ^{j\in n+1..m}]$	Un objeto con n métodos etiquetados de $l_1, \ldots, l_n$ y con i métodos de activación de $m_{n+1}, \ldots, m_m$.
$o.l$	Invocación de un método l del objeto o.
$o.l \rightleftharpoons \varsigma(x)b$	Actualización del método l del objeto o con el método $\varsigma(x)b$ y activa los métodos m de los objetos en el contexto.
$o.m \Leftarrow \varsigma(x)if(a)b : c$	Actualización de la activación m del objeto o por la condición de activación if.

Tabla 4.1: Resumen de la noción del paradigma proactivo orientado a objetos

Para más detalle, un objeto de la noción mostrada en la Tabla 4.1 es una colección de componentes $l_i = \varsigma(x_i)b_i$ con distintas etiquetas l_i. Asociados con los métodos $\varsigma(x_i)b_i$, para $i \in 1 \ldots n$; el orden de esos componentes no importa. El símbolo ς (sigma) es usado como un enlace (*binder*) con un parámetro $Self$ (*self parameter*) de un método. También se presenta una colección de condiciones de activación $m_j = \varsigma(x_j)if(a_j)b_j : c_j$, para distintas etiquetas m_j asociados a dos expresiones a y b; el orden de estas etiquetas no importa y al igual que las etiquetas l_n el símbolo $\varsigma(x_j)$ se usa como un enlazador.

Una invocación de método es escrito de la forma $o.l$ donde l es una etiqueta del objeto o. La intención es la de ejecutar el método llamado l de o con el objeto relacionado al parámetro $Self$y devolver un objeto por la reducción.

La actualización de un método se escribe de la forma $o.l \rightleftharpoons \varsigma(y)b$. La semántica de la actualización es funcional: una actualización produce una copia de o donde el método l es sustituido por $\varsigma(y)b$, además de activar a todos los métodos m que se encuentren definidos en el contexto y que tengan relación con el objeto *Activador*.

Un ejemplo del uso de dicha noción es:

$$\begin{aligned} &let\ a := [l_1 = \varsigma(x)b, l_2 = \varsigma(x)x.l_1 \rightleftharpoons \varsigma(x)c] \\ &let\ b := [l_1 = \varsigma(x)b, m_1 = \varsigma(x)if(a.l_1)x.l_1 : []] \end{aligned} \tag{4.1}$$

En $\beta\varsigma - cálculo$ se analiza la interacción entre objetos más que el cómputo que ellos pueden realizar, ya que dicho cómputo puede estar por $\lambda - cálculo$ [15, 14, 63, 39, 79, 68, 67].

En el objeto a definido en (4.1) tienen dos métodos l_1 y l_2, donde el primero hará la reducción de b y el segundo hará una actualización de $\varsigma(x)x.l_1$ por $\varsigma(x)c$. Al momento de realizar dicha actualización se activarán todas las condiciones de activación m que existan en el contexto y que tengan una relación con el objeto que los activa.

En el objeto denominado b de (4.1) se observa un método l_1 que reduce a b y una condición de activación $m_1$que estará en espera de ser activado. En caso de ser verdadera la expresión $a.l_1$ se ejecutará $x.l_1$. En caso contrario se ejecuta [] (*objeto vacío*), que en este caso expresa que no existen reducciones. Otra opción para definir $if(b)$ es el de utilizar alguna lógica diferente de la monótona, pero que converja en un resultado.

Se puede apreciar que es necesario la utilización y definición de algún tipo que permita clasificar a b en la expresión $if(b)$. Al igual que es factible realizar algún mecanismo de reducciones que permita conocer si una expresión es *Verdadera* (*True*) o *Falsa* (*False*), pero que no depende de una verificación basada en tipos.

Una posible interpretación de este tipo de reducciones se realiza en $\varsigma - cálculo$, en donde es posible representar el método if. Esto se puede realizar con la definición de dos objetos: El primero se denominará *True* y el segundo será llamado *False*; se puede observar respectivamente en las ecuaciones (4.2) y (4.3) que representan una forma de hacer el cálculo necesario para la operación mencionada.

$$\begin{aligned} True_A : Bool_A \triangleq [if = \varsigma(x : Bool_A)x.then, \\ then = \varsigma(x : Bool_A)x.then, else = \varsigma(x : Bool_A)x.else] \end{aligned} \tag{4.2}$$

$$\begin{array}{c} False_A : Bool_A \triangleq [if = \varsigma(x : Bool_A)x.else, \\ then = \varsigma(x : Bool_A)x.then, else = \varsigma(x : Bool_A)x.else] \end{array} \tag{4.3}$$

Si se toman las ecuaciones (4.2) y (4.3) se puede mostrar una sintaxis azucarada, como en la ecuación (4.4).

$$\begin{array}{c} if(b)c : d \triangleq ((b.then \rightleftharpoons \varsigma(x : Bool_A)c).else \rightleftharpoons \\ \varsigma(x : Bool_A)d)if \quad donde\, x \in FV(c) \cup FV(d) \end{array} \tag{4.4}$$

Una manera más simple de observar esto es por medio de un azúcar sintáctico de las expresiones (4.2), (4.3) y (4.4).

$$if_A(True_A)c : d \rightsquigarrow c \tag{4.5}$$

$$if(False_A)c : d \rightsquigarrow d \tag{4.6}$$

El símbolo $\rightsquigarrow$ muestra la reducción de la expresión, que en este caso será c y d de las ecuaciones (4.2) y (4.3) respectivamente.

Dentro de $\beta\varsigma - cálculo$ se incorpora la noción descrita en (4.5) y (4.6)con una sintaxis simple [36]. Para realizar la operación del proceso de activación dentro de la expresión if, se evaluará su contenido en la búsqueda de llegar a alguna reducción que pueda determinar la validez sin ambigüedad. Para esto, se hará uso de algunos operadores de la lógica de primer orden, de los cuales se puede tener una reducción.

Algunos ejemplos con el uso de las expresiones pueden (4.2) y (4.3) ser:

Conjunción: $a \wedge b \longrightarrow if(a)b : False_A$

Disyunción: $a \vee b \longrightarrow if(a)True_A : b$

Negación: $\neg a \longrightarrow if(a)False_A : True_A$

La asignación con el operador *let* puede definirse de una manera más simple como se muestra en:

$$a; b \;:=\; let\ c = a\ in\ b \quad para\ c \in FV(b) \tag{4.7}$$

$$a \;:=\; let\ c = a\ in\ [] \tag{4.8}$$

El ciclo con el operador *while* puede definirse de la siguiente manera:

$$while\ b\ do\ c \stackrel{\triangle}{=} [do = \varsigma(x)c; x.while, while = \varsigma(x)if(b)x.do : []].while \tag{4.9}$$

Las operaciones de adición y sustracción se realizan de la siguiente forma:

$$\begin{aligned}
x + y \quad &= [\\
plus = \quad &\varsigma(z)if(z.left.is0)z.result : begin \\
&z.result \rightleftharpoons \varsigma(zz)(math.inc1 \rightleftharpoons z.result).inc; \\
&z.left \rightleftharpoons \varsigma(zz)(math.dec1 \rightleftharpoons z.left).dec; \\
&z.plus \\
&end, \\
result = \quad &\varsigma(z)x, \\
left = \quad &\varsigma z)y \\
] \quad &.plus
\end{aligned}$$

La sustracción puede ser definida de la siguiente manera:

$$\begin{aligned}
x - y \quad &= [\\
minus = \quad &\varsigma(z)if(z.left.is0)z.result : begin \\
&z.result \rightleftharpoons \varsigma(z)(math.inc1 \rightleftharpoons z.result).dec; \\
&z.result \rightleftharpoons \varsigma(z)(math.dec1 \rightleftharpoons z.left).dec; \\
&z.minus \\
&end, \\
result = \quad &\varsigma(z)x, \\
left = \quad &\varsigma(z)y \\
] \quad &.minus
\end{aligned}$$

Para iniciar con la definición de la sintaxis de $\beta\varsigma - cálculo$ se dará la definición de variables libres (FV por sus siglas en inglés) y la sustitución ($b\{x \leftarrow a\}$) para los términos llamados $\beta\varsigma - términos$.

Ámbito de los objetos:

$FV(\varsigma(y)b)$	$\stackrel{\triangle}{=}$	$FV(b) - \{y\}$
$FV(x)$	$\stackrel{\triangle}{=}$	$\{x\}$
$FV([l_i = \varsigma(x_i)b_i\ ^{i\in 1...n},$ $m_j = \varsigma(x_j)if(a_j)b_j : c_j\ ^{j\in n+1...m}])$	$\stackrel{\triangle}{=}$	$(\cup^{i\in 1..n} FV(\varsigma(x_i)b_i))\cup$ $(\cup^{j\in n+1.m} FV(\varsigma(x_j)if(a_j)b_j : c_j))$
$FV(a.l)$	$\stackrel{\triangle}{=}$	$FV(a)$
$FV(a.l \leftharpoondown \varsigma(y)b)$	$\stackrel{\triangle}{=}$	$FV(a) \cup FV(\varsigma(y)b)$
$FV(\varsigma(y)if(a)b : c)$	$\stackrel{\triangle}{=}$	$FV(a) \cup FV(b) \cup FV(c) - \{y\}$
$FV(a.m \Leftarrow \varsigma(x)if(a)b : c)$	$\stackrel{\triangle}{=}$	$FV(a) \cup FV(\varsigma(x)if(a)b : c)$

Tabla 4.2: Definición de variables libres

Este cálculo podría ser similar al basado en eventos [83, 32, 20, 69, 73], un aspecto que lo distingue son las funciones de activación que se encuentran basadas en lógica de orden cero. Además, que la responsabilidad se delega totalmente a la máquina abstracta [28, 29, 45]. Esta pequeña modificación tiene sus implicaciones importantes al momento de realizar el diseño de un sistema de cómputo.

Por tratarse de un lenguaje funcional, se definen las reglas de reducción que tendrá el paradigma presentado.

$(\varsigma(y)b\{x\Re a\}$	$\stackrel{\triangle}{=}$	$\varsigma(y')(b\{y\Re y'\}\{x\Re a\})$	$Para:\ y' \notin FV(\varsigma(y)b)$ $\cup FV(a) \cup \{x\}$
$x\{x\Re a\}$	$\stackrel{\triangle}{=}$	a	
$y\{\leftarrow a\}$	$\stackrel{\triangle}{=}$	y	Para: y$\neq x$
$[l_i = \varsigma(x_i)b_i\ ^{i\in 1...n}]\ \{x\Re a\}$	$\stackrel{\triangle}{=}$	$[l_i = \varsigma(x_i)b_i\{x\Re a\}^{i\in 1...n}]$	
$(o.l)\{x\Re a\}$	$\stackrel{\triangle}{=}$	$(o\{x\Re a\}).l$	
$(o.l)\{x\Re a\}$	$\stackrel{\triangle}{=}$	$(o\{x\Re a).l$	

Tabla 4.3: Reglas de sustitución

Se puede derivar una teoría de ecuaciones sin tipo desde las reglas de reducción. El propósito de esta teoría es capturar la noción de igualdad en función de ς. Esto se usará para definir el momento en que dos objetos son de la misma forma. Se agregarán las reglas: simétrica, transitiva y de congruencia (esta última se utiliza para sustituir iguales por iguales).

4.4.1. Teoría de ecuaciones

Simétrica:

$$\frac{\vdash b \leftrightarrow a}{\vdash a \leftrightarrow b} \tag{4.10}$$

Transitiva:

$$\frac{\vdash a \leftrightarrow b \ \vdash b \leftrightarrow c}{\vdash a \leftrightarrow c} \tag{4.11}$$

Congruencia:

$$\frac{}{\vdash x \leftrightarrow x} \tag{4.12}$$

Objeto:

$$\frac{\vdash b_i \leftrightarrow b'_i \ a_j \leftrightarrow a'_j \ d_j \leftrightarrow d'_j \ c_j \leftrightarrow c'_j \ \forall i \in 1 \ldots n, \forall j \in n+1 \ldots m}{\vdash x \stackrel{\triangle}{=} [l_i = \varsigma(x_i) b_i^{\ i \in 1 \ldots n}, m_j = \varsigma(x_j) if(c_j) a_j : d_j \ ^{j \in n+1 \ldots m}] \leftrightarrow x} \tag{4.13}$$

Con (l_i y m_j distintas)

Selección:

$$\frac{\vdash a \leftrightarrow a'}{a.l \leftrightarrow a'.l} \tag{4.14}$$

Actualización de un método:

$$\frac{\vdash a \leftrightarrow a' \ \vdash b \leftrightarrow b'}{\vdash a.l \leftleftarrows \varsigma(x) b \leftrightarrow a'.l \leftleftarrows \varsigma(x) b'} \tag{4.15}$$

Actualización de un activador:

$$\frac{\vdash a \leftrightarrow a' \ \vdash b \leftrightarrow b' \ c \leftrightarrow c'}{\vdash a.m \Leftarrow \varsigma(x) if(a) b : c \leftrightarrow a'.m \Leftarrow \varsigma(x) if(a') b' : c'} \tag{4.16}$$

Evaluación de una selección, donde a$\equiv [l_i = \varsigma(x_i) b_i \{x_i\}^{\ i \in 1 \ldots n}]$:

$$\frac{j \in 1 \ldots n}{\vdash a.l_j \leftrightarrow b_j \langle a \rangle} \tag{4.17}$$

Evaluación de un método donde $a \equiv [l_i = \varsigma(x_i) b_i^{\ i \in 1 \ldots n}]$:

$$\frac{j \in 1 \ldots n}{\vdash a.l \leftleftarrows \varsigma(x) b \leftrightarrow [l_j = \varsigma(x_j) b_j, l_i = \varsigma(x_i) b_i^{\ i \in 1 \ldots n - \{j\}}]} \tag{4.18}$$

Evaluación de un activador donde $a \equiv [m_i = \varsigma(x_i)if(a_i)b_i : c_i \ ^{i\in 1\ldots n}]$:

$$\frac{j \in 1 \ldots m}{\vdash a.m \Leftarrow \varsigma(x)if(a)b : c \leftrightarrow [m_j = \varsigma(x_j)if(a_j)b_j : c_j, m_i = \varsigma(x_i)if(a_i)b_i : c_i \ ^{i\in 1\ldots m-\{j\}}]} \tag{4.19}$$

La manera de analizar los procesos de reducción se encuentran expresados mediante la semántica operacional que está dado por los siguientes sistemas:

Reducción de un objeto donde: $a \equiv [l_i = \varsigma(x_i)b_i \ ^{i\in 1\ldots n}]$

$$\frac{}{\vdash v \leftrightarrow v} \tag{4.20}$$

Reducción de selección donde: $v' \equiv [l_i = \varsigma(x_i)b_i\{x\} \ ^{i\in 1\ldots n}]$

$$\frac{\vdash a \rightsquigarrow v' \ \vdash b_j \langle v' \rangle \rightsquigarrow v \ j \in 1 \ldots n}{\vdash a.l \rightsquigarrow v} \tag{4.21}$$

Reducción de actualización:

$$\frac{\vdash a \rightsquigarrow [l_i = \varsigma(x_i)b_i \ ^{j\in 1\ldots n}] \ j \in 1 \ldots n}{\vdash a.l \leftrightharpoons \varsigma(y)b \rightsquigarrow [l_j = \varsigma(x_j)b_j, l_i = \varsigma(x_i)b_i \ ^{i\in 1\ldots n-\{j\}}]} \tag{4.22}$$

4.5. Definición de $\beta\varsigma - cálculo$

Los lenguajes imperativos son una abstracción subyacente a la máquina de Von Neumann [62], en el sentido que ellos conservan las partes esenciales básicas sin detalles superfluos. Una vista jerárquica es que los lenguajes de bajo nivel proveen un limitado nivel de abstracción, mientras un lenguaje de alto nivel puede ser visto como una máquina virtual. En esta última de manera general, se pueden encontrar algunas manipulaciones sobre la memoria dada por algunas entradas y salidas. Estas por lo regular son expresadas de manera independiente al hardware en que se pretenda implantar.

Los lenguajes orientados a objetos son naturalmente imperativos [6, 74], con métodos que realizan alguna operación dentro o fuera de los objetos. En esta sección se analizará la definición imperativa de $\beta\varsigma - cálculo$ mediante la formalización basada en la semántica operacional.

Se desarrollará un pequeño, pero expresivo lenguaje imperativo $\beta\varsigma - cálculo$; que es una variante del lenguaje funcional presentado en la sección anterior. Este lenguaje

imperativo será el núcleo para el intérprete desarrollado y llamado *Pro-Object*, con lo que es posible experimentar con aspectos importantes de la programación orientada a objetos, el cual incluye definiciones, reducciones y vistas. En el área de definiciones se encuentran algunos aspectos relevantes como son: *Variables*, *Objetos*, *métodos de actualización* y *activadores*.

a, b y c	Términos para representar a objetos.
$\varsigma(x)b$	Método $Self$ ς con parámetros x y cuerpo b.
$[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}$, $m_j = \varsigma(x_j)if(c_j)a_j : b_j\ ^{j\in n+1..m}]$	Un objeto con n métodos etiquetados de $l_1 \dots l_n$ y con condiciones de activación de $m_{n+1} \dots m_m$.
$o.l$	Invocación de un método l del objeto o.
$o.l \leftleftharpoons \varsigma(x)b$	Actualización del atributo l del objeto o con el objeto $\varsigma(x)b$, y reduce las condiciones de activación m de los objetos registrados en el contexto σ.
$o.m \Leftarrow \varsigma(x)if(a)b : c$	Actualización del método m del objeto o por el método if.
$clone(a)$	Clona el objeto a en profundidad.

Tabla 4.5: Lenguaje expresivo para denotar al imperativo-$\beta\varsigma - cálculo$

Como se describió en el apartado de la parte funcional de $\beta\varsigma - cálculo$; un objeto con la estructura $[l_i = \varsigma(x_i)b_i\ ^{i\in 1\dots n}, m_j = \varsigma(x_j)if(c_j)a_j : b_j\ ^{j\in n+1\dots m}]$ es una colección de componentes $l_i = \varsigma(x_i)b_i$, donde l_i es el nombre que se le da al método y $\varsigma(x_i)b_i$ es el cuerpo del método. También se encuentra una forma de activación basada en una sentencia if, que se describe de la manera $m_j = \varsigma(x_j)if(c_j)a_j : b_j$. En donde m_j es el nombre que se le da a dicho mecanismo de activación y $\varsigma(x_j)if(c_j)a_j : b_j$ es el cuerpo de la condición de activación. Los dos cuerpos poseen una asociación a una variable de la forma $\varsigma(x_i)$ y $\varsigma(x_j)$ respectivamente. Éstos pueden ser vistos como una relación interna al objeto; también conocida como $Self$.

Un método es invocado si se indica el nombre de la etiqueta l con relación al nombre del objeto. Un ejemplo de este caso puede ser $a.l$; esto dice que se ejecutará el método l del objeto a. Esta ejecución se encuentra dada por $\varsigma(x)b$, en donde x se encuentra relacionado con a.

La actualización de un componente se realiza mediante $o.l \leftleftharpoons \varsigma(x)b$. Esto buscará

el método l del objeto o para luego reemplazarlo por el lado derecho que es $\varsigma(x)b$. Una vez realizada la actualización, la máquina abstracta procede a buscar todos los mecanismos de activación aif los cuales verifican la operación $o.m \Leftarrow \varsigma(x)if(a)b : c$. Este último indica que la etiqueta m del objeto o será actualizado por $\varsigma(x)if(a)b : c$. Como información, una diferencia con la actualización $\leftrightharpoons$, es que esta última activa a los objetos que se encuentran dentro del mismo contexto. La actualización $\Leftarrow$ únicamente hace el reemplazo de la reducción de la parte izquierda por la parte derecha de dicha operación.

El método $clone(a)$ es la función que se encarga de realizar una copia en profundidad de un objeto. Esta copia se podría realizar de tres formas: La primera es llamada copia superficial; esta no copia referencias o instancias internas al objeto. La segunda es denominada copia en profundidad; ésta hace una copia de todas las referencias o instancias internas al objeto. El tercero y último es conocida como clonación mixta en métodos, en la que intervienen las dos primeras formas; clonación superficial y en profundidad.

En la definición se observa en la Tabla 4.6 que se aplica conjuntamente el manejo de métodos como el de atributos. En este cálculo se hará la diferencia entre métodos y atributos. La siguiente cláusula puede mostrar la manera de generar la definición de un atributo.

Atributo:	$[\ldots, l = b, \ldots] \stackrel{\triangle}{=} [\ldots, l = \varsigma(x)b, \ldots]$	para $x \notin FV(b)$
Selección:	$o.l \stackrel{\triangle}{=} o.l$	
Actualización:	$o.l \leftrightharpoons b \stackrel{\triangle}{=} o.l \leftrightharpoons \varsigma(x)b$	para $x \notin FV(b)$

Tabla 4.6: Definición de un atributo

En la implantación de este paradigma se ha establecido que los métodos no activen a ninguna sentencia aif, por lo que se usará un cálculo para definir un método y otro para un atributo.

4.5.1. Análisis de procedimientos

En la especificación de $\beta\varsigma - cálculo$ de manera funcional todo es expresado por medio de funciones. En $\beta\varsigma - cálculo$ imperativo se puede expresar procedimientos. Para ello, se considerará el uso de llamadas por valor con efectos laterales. Esto puede expresarse en función de $\lambda - cálculo$.

$a.b :=$	Términos
x	Variable
$x := a$	Asignación
$\lambda(x)b$	Abstracción
$b(a)$	Aplicación

Tabla 4.7: La sintaxis de un imperativo $\lambda - cálculo$

4.5.2. Semántica operacional

La semántica operacional es expresada en función de una relación. Esta relación se encuentra dada por un sistema de almacenamiento σ (Store) y una pila S (stack) y con un término b que se reduce a v, este último se coloca en σ'.

$$\sigma \cdot S \vdash b \rightsquigarrow v \cdot \sigma'$$

La intención de realizar esto es que se inicie con el *store* σ (que juega el papel de *heap*) y el *stack* S, el término a reduce a un resultado v, cediendo una actualización al *store* σ' y dejando el *stack* S sin cambios. Las siguientes entidades implicadas en la semántica pertenecen a las clases definidas en la Figura 3:

$\iota \in Nat$	localización de almacenamiento
$v ::= [l_i = \iota_i{}^{i\in 1..n}, m_j = \iota_j{}^{j\in 1..n}]$	resultado (l_i y m_j distintas)
$S ::= (x_i \mapsto v_i)^{i\in 1..n}$	pila (*stack*) (x_i distintas)
$\sigma ::= (\iota_i \mapsto \langle \varsigma(x_i)b_i, S_i\rangle^{\ i\in 1..n}, \iota_j \mapsto \langle \varsigma(x_j)if(c_j)a_j : b_j\rangle^{\ j\in 1..n})$	almacenamiento (*store*) (ι_i y m_j distintas)
$\sigma \vdash \diamondsuit$	juicio para almacenamiento (store)
$\sigma \cdot S \vdash \diamondsuit$	juicio para la pila (stack)
$\sigma.S \vdash a \rightsquigarrow v.\sigma'$	juicio para la reducción de términos

Figura 4.2: Definición de Almacenamiento y Pila

Un resultado v representa un objeto el cual muestra una colección de nombres de métodos, junto con la localización correspondiente en la que son colocados los métodos cerrados. También se puede apreciar una colección de nombres de *condiciones de activación*, que al igual que los métodos presentan una ubicación donde están almacenadas las condiciones de activación cerradas.

Un método cerrado se encuentra construido por $\varsigma(x_i)b_i$ y una pila S_i, tal que $FV(\varsigma(x_i)b_i) \subseteq dom(S_i)$. Finalmente, esta representación se encuentra asociada a la localización de memoria.

Asimismo, una *condición de activación* cerrada se encuentra definida por $\varsigma(x)if(c_i)a_i : b_i$ y una pila S_i, tal que $FV(\varsigma(x_i)if(c_i)a_i : b_i) \subseteq dom(S_i)$, y a su vez asociada a una localización de memoria.

A continuación se describirán algunos aspectos para el almacenamiento y sustitución realizados por la máquina abstracta, con base a las siguientes expresiones:

- Representación de la relación de almacenamiento entre ι_i y su término cerrado para $i \in 1...n$:
  - $\iota_i \mapsto \langle \varsigma(x)b, S \rangle_i \ ^{i \in 1..n}$
  - $\iota_i \mapsto \langle \varsigma(x)if(c)a : b, S \rangle_i \ ^{i \in 1..n}$
  - $\iota_i \mapsto \langle true | false, S \rangle_i \ ^{i \in 1..n}$
- Representación de la relación de colocar el resultado del término cerrado en la localidad ι_i de σ para $i \in 1...n$:
  - $\sigma \cdot \iota_i \mapsto \langle \varsigma(x)b, S \rangle$
  - $\sigma \cdot \iota_i \mapsto \langle \varsigma(x)if(c)a : b, S \rangle$
  - $\sigma \cdot \iota_i \mapsto \langle true | false, S \rangle$
- Representación de la relación que reemplaza el contenido de la localización ι de σ con su término cerrado:
  - $\sigma \cdot \iota \leftarrow \langle \varsigma(x)b, S \rangle$
  - $\sigma \cdot \iota \leftarrow \langle \varsigma(x)if(c)a : b, S \rangle$
  - $\sigma \cdot \iota \leftarrow \langle true | false, S \rangle$

Estructura de reducción bien-formada para Store:

$$\frac{\text{Store } \phi\text{: } \dfrac{}{\phi \vdash \diamondsuit}}{\text{Store } \iota\text{:} \dfrac{\sigma \cdot S \vdash \quad \iota, \iota' \notin dom(\sigma)}{\sigma, (\iota \mapsto \langle \varsigma(x)b, S \rangle, \iota' \mapsto \langle \varsigma(x)if(c)a : b, S \rangle) \vdash \diamondsuit}}$$

Semántica operacional para $fun\beta\varsigma$ y para $imp\beta\varsigma$:

Red x: $$\frac{\sigma \cdot (S', x \mapsto v, S'') \vdash \diamondsuit}{\sigma \cdot (S', x \mapsto v, S'') \vdash x \rightsquigarrow v.\sigma}$$

Red constants: $$\frac{\sigma \cdot S \vdash \diamondsuit}{\sigma \cdot S \vdash true \rightsquigarrow true \cdot \sigma} \quad \frac{\sigma \cdot S \vdash \diamondsuit}{\sigma \cdot S \vdash false \rightsquigarrow false \cdot \sigma}$$

Red object: $$\frac{\sigma \cdot S \vdash \diamondsuit \quad \iota_i \notin dom(\sigma) \quad \forall i, j \in 1..n}{\begin{array}{l}\sigma \cdot S \vdash [l_i = \varsigma(x_i)b_i\,^{i\in 1..n}, m_j = \varsigma(x)if(c_j)a_j : b_j\,^{j\in 1..n}] \rightsquigarrow \\ \qquad [l_i = \iota_i\,^{i\in 1..n}, m_j = \iota_j\,^{j\in 1..n}]. \\ \qquad (\sigma, \iota_i \mapsto \langle \varsigma(x_i)b_i, S\rangle^{i\in 1..n}, \iota_j \mapsto \langle \varsigma(x_j)if(c_j)a_j : b_j, S\rangle^{j\in 1..n})\end{array}}$$

Red select: $$\frac{\begin{array}{c}\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\,^{i\in 1..n}, m_j = \iota_j\,^{j\in 1..n}] \cdot \sigma' \quad k \in 1..n \\ \sigma'(\iota_k) = \langle \varsigma(x_k)b_k, S'\rangle \quad x_k \in dom(S') \\ \sigma' \cdot (S', x_k \mapsto [l_i = \iota_i\,^{i\in 1..n}, m_j = \iota_j\,^{j\in 1..n}] \vdash b_k \rightsquigarrow v \cdot \sigma''\end{array}}{\sigma \cdot S \vdash a.l_j \rightsquigarrow v \cdot \sigma''}$$

Red let: $$\frac{\sigma \cdot S \vdash a \rightsquigarrow v' \cdot \sigma' \quad \sigma' \cdot (S, x \mapsto v') \vdash b \rightsquigarrow v'' \cdot \sigma''}{\sigma \cdot S \vdash let\ x = a\ in\ b \rightsquigarrow v'' \cdot \sigma''}$$

Red cloning: $$\frac{\begin{array}{c}\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\,^{i\in 1..n}, m_j = \iota_j\,^{j\in 1..n}] \\ \iota_i, \iota_j \in dom(\sigma') \quad \iota'_i, \iota'_j \notin (\sigma') \quad \forall i, j \in 1..n\end{array}}{\begin{array}{l}\sigma \cdot S \vdash clone(a) \rightsquigarrow \quad [l_i = \iota_i\,^{i\in 1..n}, m_j = \iota_j\,^{j\in 1..n}]. \\ \qquad\qquad (\sigma', \iota'_i \mapsto \sigma'(\iota_i)^{i\in 1..n}, \iota_j \mapsto \sigma'(\iota_j)^{j\in 1..n})\end{array}}$$

Semántica operacional para $fun\beta\varsigma$

Red Update$_I$: $$\frac{\begin{array}{c}\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i\,^{i\in 1..n}, m_j = \iota_j\,^{j\in 1..n}] \cdot \sigma' \\ \iota_k \notin dom(\sigma') \quad k \in 1..n \\ \sigma_\beta(\iota_r) = \langle \varsigma(x)if(c)a : b, S\rangle \quad x_r \in dom(S') \quad \forall r \in 1..n\end{array}}{\begin{array}{c}\sigma \cdot S \vdash a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow \\ [l_i = \iota_i\,^{i\in 1..n}, m_j = \iota_j\,^{j\in 1..n}]. \\ (\sigma', \iota_k \leftarrow \langle \varsigma(x)b, S\rangle)\end{array}}$$

Red conditional$_{true}$: $$\frac{\sigma \cdot S' \vdash c \rightsquigarrow true \cdot \sigma' \quad \sigma' \cdot S \vdash a \rightsquigarrow v \cdot \sigma''}{\sigma_\beta \cdot S \vdash \varsigma(x)if(c)a : b \rightsquigarrow v'.\sigma''}$$

Red conditional$_{false}$: $$\frac{\sigma \cdot S' \vdash c \rightsquigarrow false \cdot \sigma' \quad \sigma' \cdot S \vdash b \rightsquigarrow v \cdot \sigma''}{\sigma_\beta \cdot S \vdash \varsigma(x)if(c)a : b \rightsquigarrow v'.\sigma''}$$

$$\text{Red updateConditional}_f\text{:}\ \frac{\begin{array}{r}\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i{}^{i\in 1..n}, m_j = \iota_j{}^{j\in 1..n}] \cdot \sigma' \\ \iota'_k \notin dom(\sigma') \quad k \in 1..n\end{array}}{\begin{array}{r}\sigma \cdot S \vdash a.m_k \Leftarrow \varsigma(x)if(c)a : b \rightsquigarrow \\ [l = \iota_i{}^{i\in 1..n}, m_j = \iota'_k{}^{j\in 1..n-\{k\}}] \\ \cdot(\sigma', \iota'_k \mapsto \langle \varsigma(x)if(c)a : b, S\rangle)\end{array}}$$

4.5.3. Ejemplos de reducciones

Para el siguiente ejemplo se empleará la mayoría de las reducciones definidas en la sección anterior y se establecerá de manera formal la aplicación de cada regla de derivación. Primero se presenta la descripción de cuatro objetos que se encuentran definidos en el mismo ambiente (A, B, C y D).

- $A = [l_1 = \varsigma(x)false, l_2 = \varsigma(x)true]$
- $B = [l_1 = \varsigma(x)if(A.l_1)A.l_2 \rightleftharpoons \varsigma(x)true : A.l_2 \rightleftharpoons \varsigma(x)false, l_2 = clone(A)]$
- $C = [l_1 = \varsigma(x)B.l_1 \Leftarrow \varsigma(x)if(A.l_2)A.l_1 \rightleftharpoons \varsigma(x)true : A.l_2 = \varsigma(x)false]$
- $D = [l_1 = \varsigma(x)A.l_1 \rightleftharpoons \varsigma(x)true]$

$$\begin{array}{lll} let & A \equiv & [l_1 = \varsigma(x)false, l_2 = \varsigma(x)true]\ in \\ & B = & [l_1 = \varsigma(x)if(A.l_1)A.l_2 \rightleftharpoons \varsigma(x)true : A.l_2 \rightleftharpoons \varsigma(x)false, l_2 = clone(A)]\ in \\ & D = & [l_1 = \varsigma(x)A.l_1 \rightleftharpoons \varsigma(x)true]\ in\ D.l_1 \end{array}$$

El primer ejemplo en donde se muestra la reducción del Objeto A:

Reducción del objeto:

$\phi \cdot \phi \vdash [l_1 = \varsigma(x)false, l_2 = \varsigma(x)true] \rightsquigarrow [l_1 = \iota_1, l_2 = \iota_2] \cdot (\iota_1 \mapsto \langle \varsigma(x)false, \phi\rangle, \iota_2 \mapsto \langle \varsigma(x)true, \phi\rangle)$

Reducción de *true*:

$(l_1 \mapsto \langle \varsigma(x)false, \phi\rangle, l_2 \mapsto \langle \varsigma(x)true, \phi\rangle) \cdot (x \mapsto [l_2 = \iota_2] \vdash true \rightsquigarrow true \cdot (\iota_1 \mapsto \langle \varsigma(x)false, \phi\rangle, \iota_2 \mapsto \langle \varsigma(x)true, \phi\rangle)$

Reducción de *false:*

$(l_1 \mapsto \langle \varsigma(x) false, \phi \rangle , l_2 \mapsto \langle \varsigma(x) true, \phi \rangle) \cdot (x \mapsto [l_1 = \iota_1] \vdash false \rightsquigarrow false \cdot (\iota_1 \mapsto \langle \varsigma(x) false, \phi \rangle , \iota_2 \mapsto \langle \varsigma(x) true, \phi \rangle)$

Se presenta la reducción con localidad de memoria:

$A \hookrightarrow \sigma_0 = (\iota_1 \mapsto \langle \varsigma(x) false, \phi \rangle , \iota_2 \mapsto \langle \varsigma(x) true, \phi \rangle)$

$\phi \cdot \phi \vdash [l_1 = \varsigma(x) false, l_2 = \varsigma(x) true] \rightsquigarrow \sigma_0$

Ejemplo donde se muestra la Reducción del Objeto B:

Reducción de un *objeto*

$B \hookrightarrow \sigma_1 =$

$(l_1 \mapsto \langle \varsigma(x) if(A.l_1) A.l_2 \rightleftharpoons \varsigma(x) true : A.l_2 = \varsigma(x) false, \phi \rangle , l_3 = \langle \varsigma(x) clone(A), \phi \rangle)$

$\phi \cdot \phi \vdash$

$[l_1 = \varsigma(x) if(A.l_1) A.l_2 \rightleftharpoons \varsigma(x) true : A.l_2 \rightleftharpoons \varsigma(x) false, l_2 = clone(A)] \rightsquigarrow [l_1 = \iota_1, l_2 = \iota_2] \cdot \sigma_1$

Reducción de *clone*

$\sigma_1 = (l_1 \mapsto \langle \varsigma(x) if(A.l_1) A.l_2 \rightleftharpoons \varsigma(x) true : A.l_2 = \varsigma(x) false, \phi \rangle , l_3 = \langle \varsigma(x) clone(A), \phi \rangle)$

$\sigma_1 . (x \mapsto [l_2 = \iota_2]) \vdash \varsigma(x) clone(A) \rightsquigarrow [l_1 = \iota_1, l_2 = \cdot \sigma_0$

Ejemplo donde se muestra la Reducción del Objeto C:

Reducción de un *objeto*

$C \hookrightarrow \sigma_3 = (l_1 \mapsto \langle \varsigma(x) B.l_1 \Leftarrow \varsigma(x) if(A.l_2) A.l_1 \rightleftharpoons \varsigma(x) true : A.l_2 = \varsigma(x) false, \phi \rangle)$

$\phi \cdot \phi \vdash [l_1 = \varsigma(x) B.l_1 \Leftarrow \varsigma(x) if(A.l_2) A.l_1 \rightleftharpoons \varsigma(x) true : A.l_2 = \varsigma(x) false] \rightsquigarrow [l_1 = \iota_1] \rightsquigarrow \sigma_3$

Reducción de actualización *condición de activación* (aif)

$\sigma_3 \cdot (x \mapsto [l_1 = \iota_1]) \vdash \varsigma(x) B.l_1 \Leftarrow \varsigma(x) if(A.l_2) A.l_1 \rightleftharpoons \varsigma(x) true : A.l_2 = \varsigma(x) false \rightsquigarrow [l_1 = \iota_1] \cdot \sigma_1$

Reducción del Objeto D:

Se tomará la secuencia de los objetos A, B y D.

$$
\begin{array}{lll}
let & A \equiv & [l_1 = \varsigma(x) false, l_2 = \varsigma(x) true] \; in \\
 & B = & [l_1 = \varsigma(x) if(A.l_1) A.l_2 \rightleftharpoons \varsigma(x) true : A.l_2 \rightleftharpoons \varsigma(x) false, l_2 = clone(A)] \; in \\
 & D = & [l_1 = \varsigma(x) A.l_1 \rightleftharpoons \varsigma(x) true] \; in \; D.l_1
\end{array}
$$

4.6. Función secuencial de activación

Es importante hacer mención que existen dos maneras de analizar el problema de las activaciones. Una de ellas es secuencial y la otra es de manera concurrente. Se procederá a explicar la forma de analizar la *condición de activación* y algunas de sus posibles consecuencias en el diseño de software.

El análisis se iniciará bajo la suposición de que se tiene una secuencia de objetos. Para ello, es necesario crear una máquina que realice una función de búsqueda e introspección en los objetos. El objetivo de este ejemplo es el de mostrar las reglas de la *función de activación* bajo $\beta\varsigma - cálculo$. En particular, se hará uso de un Tipo de Dato Abstracto (TDA), como es el de Pila (*Stack* en inglés). Este ejemplo 4.23 puede ser corroborado con el intérprete que se describe en el Apéndice A.

$$\begin{aligned} &let\ Stack = [\\ &\qquad x = 0, \\ &\qquad set = \varsigma(s)\lambda(y)s.x \rightleftharpoons y, \\ &\qquad get = \varsigma(s)s.x \\ &\qquad]; \end{aligned} \tag{4.23}$$

Para ejecutar el objeto Stack descrito en 4.23 se deben realizar algunas reducciones como las que se presentan en 4.24:

$$\begin{aligned} &red\ a := a.set(4); \\ &red\ b := a.get; \\ &view\ d; \\ &red\ d := a.set(7).set(8).set(67).get; \\ &view\ d; \end{aligned} \tag{4.24}$$

Se realizaron algunas reducciones del objeto descrito en (4.23), en las que se obtienen las vistas de 4 y 67; cabe recordar que no existen tipos de datos, por lo que los números mostrados son tomados como objetos.

Para localizar las funciones de activación, se define la función $FindIF(\sigma_n, Stack, m)$, que se encarga de realizar una búsqueda de todos los objetos que se encuentran definidos

dentro de σ_n y *Stack* que contengan los métodos de la forma m. Un ejemplo simplificado de lo anterior se puede observar en 4.25:

$$\begin{aligned} stack :=& [a_1 := [n = 0], \\ & a_2 := [i = \varsigma(x) if(a_1.n) b : c], \\ & a_3 := [i = \varsigma(x) if(r.n == true) b : c]] \\ & red\ FindIF \end{aligned} \tag{4.25}$$

La función secuencial $FindIF$ es la que se encarga de realizar la introspección en los objetos para localizar las activaciones y transferir el control. Si un objeto posee más de dos condiciones de activación y la transferencia de control es secuencial, se deberá tener cuidado con los efectos laterales. Si se realiza de manera concurrente, se deberá proponer un mecanismo más complejo para lograr un estado de aceptación. Esto será explicado a detalle en el Capítulo 5. Por el momento se presenta en 4.26 un ejemplo secuencial de este caso:

$$\begin{aligned} & let\ a := [n = 0] \\ & let\ b := [l_1 = \varsigma(x) if(= a.n\ 1) a.n \rightleftharpoons +a.n\ 1 : []] \\ & let\ c := [l_1 = \varsigma(x) if(= a.n\ 1) a.n \rightleftharpoons +a.n\ 1 : []] \\ & red\ d := a.n \rightleftharpoons 1 \end{aligned} \tag{4.26}$$

Si se realiza la reducción de 4.26 según lo indicado por la regla *red* y visualizamos el resultado del objeto denominado a, el resultado aparente sería $n = 3$, aunque el resultado real será el de $n = 2$. Para comprobar esto se recomienda el uso del intérprete Pro-Object que se encuentra en el disco adjunto que acompaña este documento. Sin embargo, es posible proponer otra forma semántica para obtener el resultado de 3.

Se presentará un Algoritmo para sincronizar tres semáforos, esto nos dará una visión de la programación con activadores cerrados al objeto, ver 4.2.

Algoritmo 4.2 Semáforos

```
let Semaforo1 := [ px=20, py=20, Color = 1 , sin = @(x) if ( == Semaforo2.Color 1 )
x.Color <= 1 : x.Color <= 0 ]
let Semaforo2 := [ px=10, py=10, Color = 1, sin = @(x) if ( == Semaforo1.Color 1 )
x.Color <= 1 : x.Color <= 0 ]
let Semaforo3 := [ px=30, py=30, Color = 0, sin = @(x) if ( == Semaforo1.Color 1 )
x.Color <= 0 : x.Color <= 1 ]
```

4.7. Función $FindIF$ concurrente

Como se había mencionado en la sección 4.6, la función $FindIF$ podría ser concurrente para dar alternativas de solución a los efectos laterales. Sin embargo, recordando que una Máquina de Turing Abstracta con varias cintas puede ser emulada por otra Máquina de Turing Abstracta de una sola cinta [42, 53, 38], no habría necesidad de realizar este tipo de funciones, pero si se requiere optimizar el tiempo de solución de un problema se deberá proponer una alternativa concurrente.

Algunos puntos importantes en la construcción de paradigmas de programación concurrentes son:

- La semántica no debe desviar su concepto inicial, algunos lenguajes agregan instrucciones que al transcurso del tiempo llegan a cambiar totalmente.

- La sintaxis que se proponga deberá ser intuitiva para el programador; ya que existen reglas que complican la comprensión de un programa.

Para que una función $FindIf$ sea concurrente y evite efectos laterales, se deberá establecer que las operaciones de la condiciones de activación sean cerradas, de esta manera se evitarán los problemas de estados no alcanzables o problemas de paro (*halting*). Otra de las características que se deberá tomar en cuenta es la condición de activación, que también deberá ser cerrada o con un sólo operador por verificar; ya que la misma concurrencia presenta características especiales con algunos operadores lógicos (conjunción y disyunción). En este trabajo no se profundizará en este tema, debido a que el objetivo planteado es el de establecer las bases de un paradigma que permita simplificar el diseño de software.

4.8. Un intérprete basado en $\beta\varsigma - cálculo$ con tipos

En la sección 4.6 se observó la definición de $\beta\varsigma - cálculo$, ahora se mostrará una posible implantación con tipos y posteriormente, en el Capítulo 5 se realizarán algunos ejemplos y soluciones para diversos problemas, así como sus implicaciones en el diseño. La sintaxis que se define se describe en la Tabla 4.12.

$a, b, c ::= x$	Variables
$\mid INT$	Números enteros
$\mid true$	True
$\mid false$	False
$\mid [l_i = \varsigma(x_i)b_i, m_i = \varsigma(x_i)if(a_i)b_i : c_i]$	Pro-Object
$\mid a.l$	Seleccionar
$\mid a.l \rightleftharpoons \varsigma(x)b$	Actualizar
$\mid \lambda(x)b$	Abstracción
$\mid (a)$	Paréntesis
$\mid + \ a \ b$	Adición
$\mid - \ a \ b$	Sustracción
$\mid * \ a \ b$	Multiplicación
$\mid / \ a \ b$	División
$\mid > \ a \ b$	Mayor que
$\mid < \ a \ b$	Menor que
$\mid == \ a \ b$	Igualdad
$\mid not \ a$	Negación
$\mid and \ a \ b$	y lógico
$\mid or \ a \ b$	o lógico
$\mid clone(a)$	Clonar

Tabla 4.12: Sintaxis básica de Pro-Object

Las reglas de $\beta\varsigma - cálculo$ sugieren un algoritmo de reducción, en donde los algoritmos toman términos cerrados y si estos convergen producen un resultado o un identificador de problemas (*token wrong*); los cuales representan un error en cálculo. Se escribirá una función llamada *salida* para identificar el resultado de alguna ejecución en función de una entrada, se asume que el algoritmo termina o encuentra un estado de aceptación. Este último puede ser definido recursivamente con la excepción de los métodos aif.

$Salida(INT) \stackrel{\triangle}{=} INT$

$Salida(true) \stackrel{\triangle}{=} true$

$Salida(false) \triangleq false$

$Salida([l_i = \varsigma(x_i)b_i, m = \varsigma(x_j)if(a_j)b_j : c_j]^{i,j\in 1..n}) \triangleq$
$[l_i, = \varsigma(x_i)b_i,$
$m_j = \varsigma(x_j)if(a_j)b_j : c_j]^{i,j\in 1..n}$

$Salida(a.l_k) \triangleq let\ o = Salida(a)\ in$
$if\ o\ es\ de\ la\ forma$
$[l_i = \varsigma(x_i)b_i, m = \varsigma(x_j)if(a_j)b_j : c_j]^{i,j\in 1..n}\ con\ k \in i$
$then\ Salida(b\,\langle o\rangle)$
$else\ error$

$Salida(a.l_k \rightleftharpoons \varsigma(x)b) \triangleq let\ o = Salida(a)\ in$
$if\ o\ es\ de\ la\ forma$
$[l_i = \varsigma(x_i)b_i, m = \varsigma(x_j)if(a_j)b_j : c_j]^{i,j\in 1..n}\ con\ k \in i$
$then$
$\left[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n-\{k\}}, m = \varsigma(x_j)if(a_j)b_j : c_j\right]^{i,j\in 1..n}$
$else\ error$

$Salida(a.m_k \rightleftharpoons \varsigma(x)b) \triangleq let\ o = Salida(a)\ in$
$if\ o\ es\ de\ la\ forma$
$[l_i = \varsigma(x_i)b_i, m = \varsigma(x_j)if(a_j)b_j : c_j]^{i,j\in 1..n}\ con\ k \in i$
$then$
$\left[l_i = \varsigma(x_i)b_i, m = \varsigma(x_j)if(a_j)b_j : c_j\ ^{j\in 1..n-\{k\}}\right]^{i,j\in 1..n}$
$else\ error$

$Salida((a)) \triangleq ([l_i = \varsigma(x_i)b_i, m = \varsigma(x_j)if(a_j)b_j : c_j]^{i,j\in 1..n})$

$Salida(clone(a)) \triangleq a'\ en\ otra\ localidad\ de\ memoria$

Se puede observar que $\vdash c \rightsquigarrow v$ si y solamente si $Salida(c) \equiv v$ y que v no genere algún error. Durante la descripción de la función $Salida$, se describieron la mayoría de las reglas sintácticas omitiendo algunas operaciones aritméticas y lógicas. Estas operaciones aritméticas pueden ser expresadas en términos de $\lambda - cálculo$.

Funciones lógicas

$true$	$\stackrel{\triangle}{=}$	$\lambda x.\lambda y.x$
$false$	$\stackrel{\triangle}{=}$	$\lambda x.\lambda y.y$
not	$\stackrel{\triangle}{=}$	$\lambda z.z\ true\ false$
and	$\stackrel{\triangle}{=}$	$\lambda x.\lambda y.(x\ (y\ true\ false) false)$
or	$\stackrel{\triangle}{=}$	$\lambda x.\lambda y.(x\ true (y\ true\ false)$
if	$\stackrel{\triangle}{=}$	$\lambda z.\lambda x.\lambda y.z\ x\ y$

Funciones aritméticas

Succ	$\stackrel{\triangle}{=}$	$\lambda n.\lambda f.\lambda x.n\ f(f\ x)$
Pred	$\stackrel{\triangle}{=}$	$\lambda x.\lambda y.\lambda z.x(\lambda p.\lambda q.q(p\ y))(\lambda y.z)(\lambda x.x)$
Add	$\stackrel{\triangle}{=}$	$\lambda n.\lambda m.\lambda f.\lambda x.n\ f(m\ f\ x)$
Mult	$\stackrel{\triangle}{=}$	$\lambda n.\lambda m.\lambda f.n(m\ f)$
Exp	$\stackrel{\triangle}{=}$	$\lambda n.\lambda m.m\ n$
Monus	$\stackrel{\triangle}{=}$	$\lambda a.\lambda b.b\ Pred\ a$
isZero	$\stackrel{\triangle}{=}$	$\lambda n.n(\lambda x.False)True$
Equals	$\stackrel{\triangle}{=}$	$\lambda a.\lambda b.And(isZero(Monus\ a\ b))(isZero(Monus\ b\ a))$
Leq	$\stackrel{\triangle}{=}$	$\lambda a.\lambda b.isZero(Monus\ a\ b)$
Ged	$\stackrel{\triangle}{=}$	$\lambda a.\lambda b.isZero(Monus\ b\ a)$

Algunos ejemplos:

Probar que:$And\ true\ false \rightsquigarrow False$

$And\ true\ false \stackrel{\triangle}{=} (\lambda y.true(y\ true\ false)false)$

$\rightsquigarrow true(false\ true\ false)false$

$\rightsquigarrow(\lambda y.false\ true\ false)false$

$\rightsquigarrow false\ true\ false$

$\rightsquigarrow (\lambda y.y)false$

$\rightsquigarrow false$

Probar que: $Not \quad true \rightsquigarrow false$

$Not\ true \stackrel{\triangle}{=} true\ false\ true$

$\rightsquigarrow (\lambda y.false)true$

$\rightsquigarrow false$

En el disco adjunto a este documento se puede encontrar un programa para realizar las pruebas anteriores, el cual tiene el nombre físico de $lambda.jar$.

Ahora bien, recordemos que se está trabajando con una definición de objetos por lo que las expresiones aritméticas y lógicas deberán ser colocadas en esta forma. Es posible incorporar los términos de $\lambda - cálculo$ a $\beta\varsigma - cálculo$ a esta transformación la denominamos $\lambda\beta\varsigma - cálculo$.

$\lceil x \rceil \triangleq x$

$\lceil b(x) \rceil = \lceil b \rceil .arg \bullet \lceil a \rceil .val$

$\lceil \lambda x \rightarrow b \rceil = [arg = \varsigma(x)x.arg, val = \varsigma(x)(\lceil b \rceil [x \rightleftharpoons x.arg])]$

Si se desea probar algunas conversiones, favor de utilizar el programa $FOB.jar$ que viene en el disco adjunto a este documento. Este programa permite experimentar con transformaciones de expresiones λ a ς.

4.9. Una propuesta para modelar $\beta\varsigma - cálculo$ con diagramas UML

En la programación y modelado con UML (Unified Modeling Language) se propone el uso de algunos instrumentos que permiten tener una visión precisa de lo que sucede dentro de un sistema. UML, es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. Este tipo de diagramas incluye aspectos conceptuales, tales como: procesos de negocios y funciones del sistema.

Una implicación del uso de éstos dentro del paradigma de los objetos proactivos, es que se requiere de la definición de un diagrama de clases/secuencia, en el que el diseñador pueda observar la interacción entre los objetos. Una disquisición que impera en estas definiciones, es que no se debe pensar que se está modelando a los objetos de manera dinámica, ya que la definición de proactividad está dada por de facto en el paradigma propuesto.

Para proponer una forma de diagramar los objetos proactivos, se analizaron los *diagramas de secuencia*, que son uno de los más efectivos para modelar interacción entre objetos en un sistema. Por otra parte, también se estudiarán las interacciones con el *diagrama de casos de uso*, el cual permite el modelado de una vista de negocios "*business*" del escenario.

Para representar los objetos proactivos en un diagrama, se pueden usar los diagramas de clase, aunque éstos únicamente muestran un momento estático del sistema. Otra opción, es el de usar un diagrama de secuencia con alguna nueva relación para indicar la interacción entre los objetos. Estos dos diagramas, el de clase y el de secuencia son usados de esta manera, sin embargo, en los objetos proactivos no queda representado el problema de las activaciones, ya que se busca en un momento estático esquematizar estas relaciones y dejar a un lado el diagrama de secuencia, esto se debe a que los objetos proactivos por sí mismos establecen estas condiciones de activación.

Debido a lo explicado en el párrafo anterior se propone otro tipo de diagrama, el cual tendrá el nombre de *diagrama proactivo*. En este diagrama únicamente se establecerá las relaciones de activación entre los objetos, delegando la responsabilidad del orden de las secuencias de activación a los diagramas de secuencia.

Un *diagrama proactivo* establece la relación entre dos objetos, para observar el diseño de esta relación, se puede ver la Figura 4.3, en donde se muestran dos objetos o clases, la relación y la condición de activación. Esta relación se da entre el *Atributo1* del *Objeto1* y los métodos de un *Objeto2*.

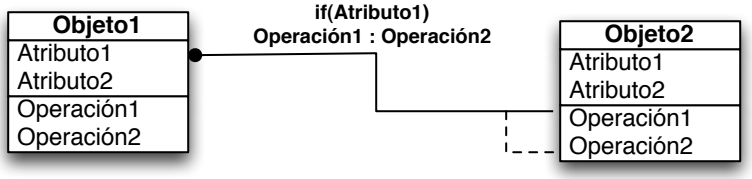

Figura 4.3: Relación de activación con un único atributo

Se puede observar en la Figura 4.3, que se establece una relación con una línea con un punto • en uno de sus extremos, y en el otro se encuentran dos líneas, la primera es completa y la segunda es punteada. El extremo de la línea con un punto indica el atributo involucrado en la operación; en este caso *Atributo1*. En el otro extremo se aprecian dos líneas, la primera es completa e indica qué operación se ha de realizar si la condición if es verdadera; en este caso es *Operación1*. La segunda línea que es punteada muestra qué operación se realiza en caso de que la condición de activación no se cumpla; en este caso es *Operación2* del *Objeto2*.

Con el principio básico de relación mostrado en el diagrama proactivo de la Figura 4.3 se pueden realizar otros ejemplos importantes de analizar. Uno que veremos aquí en la Figura 7, muestra la importancia de utilizar los diagramas de secuencia. Se inicia con un diagrama proactivo con dos clases y el mismo atributo.

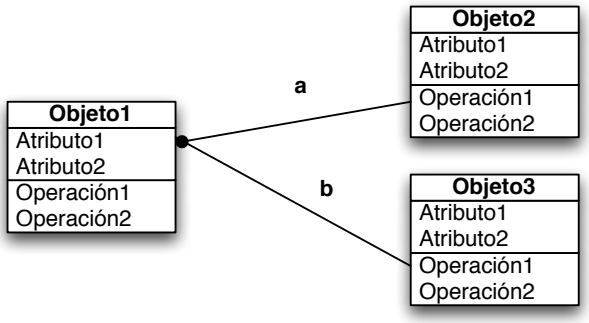

Figura 4.4: Diagrama proactivo con activación de dos objetos

En la Figura 4.6 se tiene la existencia de tres objetos *Objeto1*, *Objeto2* y *Objeto3*, los cuales se encuentran relacionadas por dos funciones de activación o señales de activación. Este diagrama puede ser expresivo para indicar solamente la lógica de los objetos que intervienen, más la relación de cómo ellos se activan. Los diagramas de Secuencia indicarán cual de las dos señales de activación se ejecutan primero sí a o b , en el caso de que a y b tengan la misma condición de activación.

En el momento en que se incorpora en un objeto la condición de activación y con alguna lógica de primer orden cero ésta puede presentar algunos caso interesantes de estudio.

4.9.1. Activación con disyunción

Se puede suponer que existe una condición de la siguiente forma:

$$if(Objeto_1.Atributo_1 = true \ \| \ Objeto_2.Atributo_1 = true) \ Operación_1$$

Donde el símbolo $\|$ significa disyunción. Se puede ver en Figura 4.7 que es posible expresar dicha sentencia dentro del *diagrama proactivo*.

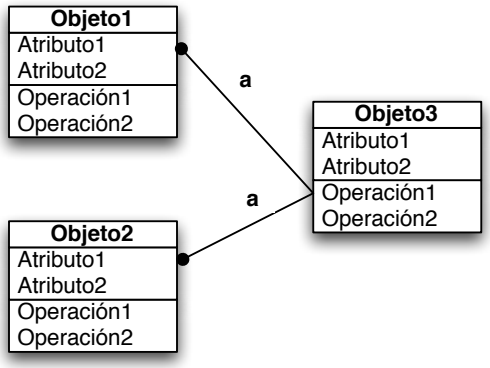

Figura 4.5: Diagrama proactivo para la disyunción

En la Figura 4.7 se observan tres objetos: *Objeto1*, *Objeto2* y *Objeto3*, el *Objeto3* se activará si alguna de las dos opciones son verdaderas ($Objeto_1.Atributo_1 = true$ o $Objeto_2.Atributo_2 = true$). Esto dependerá del análisis que realice el compilador, si se requiere la evaluación de las dos expresiones o solamente realiza el análisis suficiente para aceptar que la expresión es perezosa con la primera opción que sea verdadera.

4.9.2. Activación bajo la conjunción

Se tiene una condición de la siguiente forma:

$if(Objeto_1.Atributo_1 = true \;\&\&\; Objeto_2.Atributo_1 = true)\; Operación_1$

Donde el símbolo && significa conjunción. Se puede ver en Figura 4.6 que es posible expresar dicha sentencia dentro del *diagramaproactivo*.

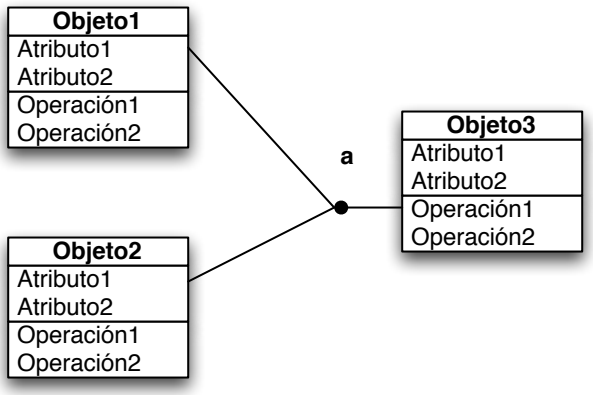

Figura 4.6: Diagrama proactivo con conjunción

En la Figura 4.6 se observan tres objetos: $Objeto1$, $Objeto2$ y $Objeto3$, el $Objeto3$ se activará si alguna de las dos opciones son verdaderas ($Objeto_1.Atributo_1 = true$ y && $Objeto_2.Atributo_1 = true$) .

4.9.3. Activación bajo la negación

Se tiene una condición de la siguiente forma:

$if(!Objeto_1.Atributo_1 = true)\ Operación_1$

Donde el símbolo ! significa negación. Se puede ver en Figura 4.7 que es posible expresar dicha sentencia dentro del diagrama proactivo.

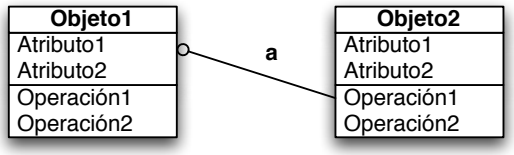

Figura 4.7: Diagrama proactivo para la negación

En la Figura 4.7 se observan dos clases: *Objeto1* y *Objeto2*, en el *Objeto2* se activará si la condición es falsa.

Capítulo 5

Disquisiciones experimentales

5.1. Resumen

En este capítulo se presentan algunos experimentos desarrollados con el intérprete Pro-Object.

5.2. Objetivos del capítulo

- Solucionar algunos de los problemas comunes en el ámbito de la programación orientada a objetos.
- Presentar problemas experimentales con el paradigma propuesto.
- Enlistar algunas recomendaciones para un diseño bajo el paradigma propuesto.
- Mostrar algunas disquisiciones con la evolución de los objetos proactivos.

5.3. Introducción

Existen diferentes métodos para implantar un compilador o intérprete. En este capítulo se mostrarán los experimentos realizados con el software llamado Proactive Object Interpreter o por su acrónimo (*Pro-Object*) denominado pobj. El diseño de dicho intérprete se encuentra especificado en el Apéndice A.

La sintaxis completa de $\beta\varsigma - cálculo$ con BNF (Backus–Naur Form) se encuentra indicada en el Apéndice A, por lo que en la Tabla 5.2 se dará una breve introducción a su sintaxis.

BeginParser	::=	(Assignation | Reduction | ViewObject)* <EOF>
Assignation	::=	<LET> JDefine
JDefine	::=	<IDENTIFIER> “:=“ expression
expression	::=	Object|Variable|Integer|Lambda|Clone
PApplication	::=	“(“ (expression)* “)” (SelectUpdate)?
Clone	::=	<CLONE> “(“ expression “)” (SelectUpdate)?
Lambda	::=	<LAMBDA> “(“ <IDENTIFIER> “)” expression (expression)*
SelectUpdate	::=	Update|UpdateIf|Select
UpdateIf	::=	“.” <IDENTIFIER> <UPDIF> FunctionSigma If (“-” Select)?
Select	::=	“.” <IDENTIFIER> (“(“ Parameter “)”)? (SelectUpdate)?
Parameter	::=	expression
Update	::=	“.” <IDENTIFIER> “<=“ (FunctionSigma)? expression (“-” Select)?
Variable	::=	<IDENTIFIER> (SelectUpdate)?
Integer	::=	<INTEGER>
Object	::=	“[“ ((aIf | Labels) (“,” (aIf | Labels))*)* “]” (SelectUpdate)?
aIf	::=	<IDENTIFIER> “=“ FunctionSigma If
If	::=	<IF> “(“ expression “)” expression “:” expression
Labels	::=	<IDENTIFIER> “=“ (FunctionSigma)? expression
FunctionSigma	::=	<SIGMA> “(“ <IDENTIFIER> “)”

Tabla 5.2: Introducción a la sintaxis de $\beta\varsigma - cálculo$

En la Figura 5.2 se observa parte de la sintaxis BNF de $\beta\varsigma - cálculo$ lo que se omitió fueron las operaciones lógicas y las operaciones aritméticas. El primer ejemplo que se mostrará es el uso de una calculadora ver Figura 2.

Algoritmo 5.1 Calculadora básica

```
let calc := [
      tmp = 0,
      total = 0,
      enter = @(x)\(y) x.tmp <= y,
      sum = @(x)\(y) ( x.tmp <= x.total ).total <= + x.tmp y,
      min = @(x)\(y) ( x.tmp <= x.total ).total <= - x.tmp y,
      mul = @(x)\(y) ( x.tmp <= x.total ).total <= * x.tmp y,
      div = @(x)\(y) ( x.tmp <= x.total ).total <= / x.tmp y,
      clear = @(x) ( x.tmp <= 0 ).total <= 0
]
```

En el Algoritmo 5.1 se ve la implantación de una calculadora con las cuatro operaciones básicas: suma, resta, multiplicación y división (el resultado es expresado en el dominio de los enteros). Esta calculadora es sugerida por un grupo de investigadores para probar el paradigma orientado a objetos [18].

Un ejemplo de reducción del código calculadora mostrado con el código [18] es:

$red\ v = calc.sum(1).sum(5).sum(4).min(2).mul(2).div(4).total$

$view\ = v \rightsquigarrow 4$

En el ejemplo anterior se realizó un cómputo descrito por $\lambda - cálculo$, pero basado en el paradigma orientado a objetos de $\beta\varsigma - cálculo$. Esto no presenta diferencia alguna con $\varsigma - cálculo$ por el momento, lo que se pretende mostrar es que la teoría presentada se mantiene. En la siguiente sección se mostrará algunas diferencias importantes que se definen en $\beta\varsigma - cálculo$.

5.4. Propuesta de solución al problema del estanque

En el Capítulo 1 sección 1.4 se planteó el problema del Estanque y de los Sensores. Una manera de solucionar dicho problema mediante $\beta\varsigma - cálculo$ es el mostrado en la Figura 5.1:

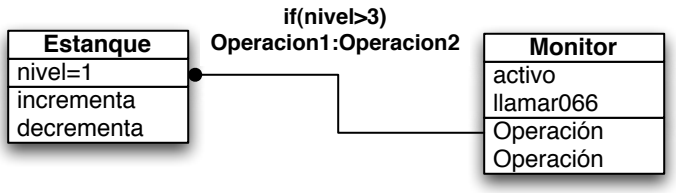

Figura 5.1: Diagrama proactivo a la posible solución del estanque

En el diagrama de la Figura 5.1 se muestran dos objetos. El primero es el que posee el nombre de *Estanque* y el segundo el de *Monitor*. El primer objeto se denominará *Objeto Activador* y el segundo se denomina *Objeto Activado*. La manera de implantar dicho diagrama de la Figura 5.1 dentro del lenguaje *Pro-Object* es la que se puede observar en el Algoritmo 5.2:

Algoritmo 5.2 Código de los objetos para el problema del estanque

```
let Estanque:= [
   nivel=0,
   inc = @(x) x.nivel <= + x.nivel 1,
   dec = @(x) x.nivel <= - x.nivel 1
]
let Monitor1:= [
   active = false,
   numActivaciones = 0,
   aif1 = @(x) if ( > Estanque.nivel 1 ) (x.active <= true).numActivaciones <= +
x.numActivaciones 1 : []
]
```

En el Algoritmo 5.2 se observan dos objetos (*Estanque* y *Monitor*1) y la condición de activación que se encuentra dentro del objeto *Monitor1*. Las funciones de *op1* y *op2* de *Monitor1* fueron optimizadas sintácticamente. Esto se realizó para reducir el número de pasos en la ejecución de dichos procesos. Otro de los detalles que se puede observar es que las operaciones se realizan en infijo, esto es $(> \ \ Estanque.nivel\ 1)$ lo que es equivalente a $(Estanque.nivel1 > 1)$ en prefijo.

Se procede a realizar algunas reducciones para analizar el comportamiento:

$red\ alarm := Monitor_1.active \rightsquigarrow false$

$red\ callEmergency := Monitor_2.callEmergency \rightsquigarrow NULL$

$red\ c := Estanque.inc \rightsquigarrow 1$

$red\ alarm := Monitor_1.active \rightsquigarrow false$

Se agrega un objeto más a la definición del Algoritmo 5.2, ésta se puede observar en el Algoritmo 5.3:

Algoritmo 5.3 Código de un objeto proactivo llamado Monitor2

```
let Monitor2:= [
    callEmergency = false,
    aif1 = @(x) if (> Monitor1.numActivaciones 2 ) x.callEmergency <= true : []
]
```

Se aplicará una segunda reducción a los tres puntos anteriores:

$red\ c := Estanque.inc \rightsquigarrow 2$

$red\ alarm := Monitor1.active \rightsquigarrow false$

$red\ c := Estanque.inc \rightsquigarrow 3$

$red\ alarm := Monitor1.active \rightsquigarrow true$

$red\ callEmergency := Monitor2.callEmergency \rightsquigarrow false$

$red\ c := Estanque.inc \rightsquigarrow 4$

$red\ alarm := Monitor1.active \rightsquigarrow true$

$red\ nac := Monitor1.numActivaciones \rightsquigarrow 3$

$red\ callEmergency := Monitor2.callEmergency \rightsquigarrow true$

Se observa en estas últimas reducciones que el *Monitor1* se *activa*, por lo que el objetivo planteado al principio de este trabajo se resuelve con los principios de activación, mostrando la sintaxis y semántica mínima para un lenguaje.

5.5. Ventajas y desventajas en el paradigma proactivo propuesto

Todo paradigma permite dar ciertas ventajas para resolver algunos problemas en particular. En esta sección se hará uso de algunas singularidades que se presentan al hacer uso de este trabajo.

En el llamado cómputo proactivo se menciona que el uso y la interconexión de varios dispositivos seguirán en aumento [89]. Por lo que deberá existir un paradigma de programación, que permita proponer una solución que sea de manera intuitiva para realizar

dichas tareas. Es por ello que con este paradigma se propone una manera de desarrollar estas formas de interacción.

Existen varias ventajas al momento de usar este cómputo. El objetivo no es describir todas las posibles ventajas que propone el paradigma propuesto, pero si nombrar las de mayor relevancia.

El objeto activador delega la responsabilidad a la máquina abstracta o sistema en que trabaje para localizar todos los objetos que se encuentren dentro de su contexto. Los objetos activados son los que contienen la *función de activación,* y al momento en que ésta sea verdad se procede a lo descrito por la definición. Esto muestra que no se delega responsabilidad a otro objeto o interfaz, ya sea de manera directa o indirecta. Esto hace que el principio de simple responsabilidad se mantenga de una manera transparente.

Los objetos activados poseen una simple manera de "negociar" con otros objetos, debido a que su comunicación se basa en lógica de primer orden.

El programador solamente tiene que tener una orientación en cómo buscar la manera en que los objetos reaccionen conforme a su entorno. Se pueden agregar dentro de un contexto *objetos activados* y *objetos activadores* en tiempo de ejecución, lo que permite analizar desde otra perspectiva la evolución de los objetos [75].

Es posible solucionar problemas distribuidos de una manera intuitiva, aunque el paso de mensajes deberá ser resuelto mediante otro cálculo, el cual se ha explicado en el Capítulo 3.

Al hablar de las desventajas del paradigma proactivo orientado a objetos, uno podría suponer equívocamente que existen problemas con la definición o que existen efectos laterales en el uso de dicho paradigma. Por esa razón, se ha decidido dividir dichas ventajas en dos secciones, la primera de ellas hablará sobre algunas disquisiciones que posee el paradigma presentado y en la segunda, se enumerarán algunas limitaciones y desventajas que presenta dicho paradigma al momento de hacer uso de él frente a ciertos problemas de cómputo.

5.5.1. Disquisiciones en el paradigma proactivo orientado a objetos

La función llamada *función de activación*, puede hacer introspección para localizar objetos que se encuentren dentro de otros objetos o únicamente localizar los objetos que estén referenciados por alguna definición o variable. Esto conlleva a un costo de

cómputo que podría traer implicaciones de determinismo, si la definición se realiza de la siguiente manera:

Se posee un objeto activador de la forma $a := [b := [n = 0]]$ y un objeto activado de la forma $c := [aif = \varsigma(x)if(b.n == 0)d : n \rightleftharpoons +n\ 1$. Se podría pensar que se ejecutaría d debido a que la función de activación realiza una búsqueda con introspección. Una solución a esta ambigüedad es realizar ésta de manera explicita $c := [aif = if(a.b.n == 0)d : e]$, lo que elimina la búsqueda de activaciones en profundidad por así llamarlo, si eso es lo que se requiere.

Otra funcionalidad es el uso del método *clone*, este método no surte efecto en la activación hasta que se encuentre referenciado por alguna variable. Se podría suponer que en la clonación existe dicho objeto para ser activado; pero por la definición de activación es importante que exista alguna referencia.

En el Capítulo 4 se muestran ejemplos bajo la definición de objetos. Es posible utilizar la definición de clases, debido a que la única diferencia entre objetos y clases es que se utiliza el método *clone* para crear objetos; mientras que una clase por lo regular emplea el método *new* o alguna otra función de creación [5, 4, 6].

5.5.2. Disquisiciones en el uso del paradigma proactivo

En este paradigma se recomienda realizar un pensamiento proactivo, esto es, todo objeto puesto dentro de un contexto tiende a reaccionar con los demás objetos que se encuentran en el mismo contexto, con lo que se da la posibilidad de que exista sinergia.

Para buscar que un objeto no tenga efectos laterales, se propone una pequeña secuencia de pasos para hacer uso de este paradigma. Lo primero que se debe realizar es la definición de todos los objetos importantes en el dominio de discurso. Posteriormente, realizar cada activación y condición de activación e iniciar la ejecución. Algunas recomendaciones en su uso se describen a continuación.

Se pueden realizar algunas estructuras de control de los lenguajes imperativos. Cabe recordar que el objetivo del paradigma no es mostrar el uso de estos problemas, sino el buscar cómo hacer la interacción y no qué hacer con la interacción entre objetos.

Algoritmo 5.4 Código para representar un ciclo *for* imperativo

```
let beginIf := [ numero = 0 ]
let procIf := [aif = @(x) if ( < beginIf.numero 10 ) beginIf.numero <= + beginIf.numero
1 : [] ]
red tmp := beginIf.numero <= 0
```

En el Algoritmo 5.5 se presenta una forma de implantar el ciclo *for* de los lenguajes imperativos. El paradigma puede simular estas funciones, aunque no es recomendable que el objeto activado modifique al objeto activador, ya que esto puede suponer que se encuentran dentro del mismo contexto. Adicionalmente, cada uno de ellos fue diseñado en el mismo momento; en otras palabras, el diseño mostrado en el Algoritmo 5.5 se puede realizar con una función imperativa directa, y no tener que hacer uso de las activaciones para solucionar un problema que bien puede ser resuelto de manera simple. Esto es como lo hacen los lenguajes imperativos tradicionales (C o Pascal).

Otra de las observaciones realizadas a este paradigma, es el uso de los operadores que permiten un diseño en el que se dé una operación recursiva y sin paro. Si se procede con el mismo diseño del Algoritmo 5.5 y se utiliza alguno de estos operadores lógicos, se puede incurrir en ciclos infinitos o con algún problema de paro [42].

Un ejemplo de esto, se puede observar en 5.5:

Algoritmo 5.5 Código para representar el problema de paro

```
let a:=[v=0]
let b:=[m=@(x)if (> a.v 0 ) a.v <= + a.v 1 : a.v <= + a.v 1 ]
red c:= a.v <= 1
```

Si se realiza la reducción de 5.5, el programa puede no terminar satisfactoriamente. Pero al igual que en 5.5 una de las causas de este comportamiento, se debe a una interacción entre los dos objetos de una manera poco controlada. Una posible solución sería el de crear otra forma de visibilidad, esto es, en los lenguajes orientados a objetos tales como C++, Java, entre otros; existen ciertos aspectos de visibilidad para los miembros de clase. Una propuesta es generar alguna que puede ser llamada "*proactive*"; la cual impida que otros objetos activados modifiquen sus propiedades. Esto podría tener otras implicaciones que por el momento salen del objetivo del presente trabajo, ya que principalmente lo que se busca es analizar la forma en que los objetos interactúan, así como el desarrollo incremental.

Algunas de las ventajas en el uso de este paradigma son el de incorporar objetos al instante de la ejecución, y no necesariamente al momento del diseño. En los lenguajes orientados a objetos como C++, Java, C#, entre otros, se diseña la jerarquía de clases. Este diseño genera alguna rigidez al momento de realizar un cambio importante en la clase base. Si existe algún cambio en alguna clase base por la manera en que se encuentra definida en los lenguajes ya mencionados, las clases derivadas tendrán que sufrir cambios significativos. En este paradigma esos problemas pueden presentar algunas ventajas.

Para analizar esta solución se tomará el código que se muestra en el Algoritmo 5.2 y las reducciones descritas en la misma sección.

Se puede observar la siguiente reducción:

$red\ c := Estanque.inc \rightleftharpoons 2$

$red\ alarma := Monitor1.active \rightleftharpoons true$

Después de haber realizado la reducción anterior, se agrega al contexto otro objeto llamado *Monitor2*. Ahora existen tres objetos dentro del contexto (*Estanque*, *Monitor1* y *Monitor2*).

En la definición de *Monitor2* se observa que al momento en que el número de activaciones descrita por la variable *numActivaciones* sea mayor a 2, en este caso se llamará a un "número de emergencia" descrito por una variable *callEmergency*. Al agregar dicho objeto al contexto, la ejecución de los demás objetos no se ve afectada por ningún motivo. Hay que tener precaución al momento de implantar la función de activación dentro de sistemas concurrentes. Esto es debido a que dicha función debe tener algún mecanismo que no permita la referencia a objetos, hasta que no se encuentre una instancia por medio de una variable o atributo.

Otra de las ventajas que presenta el diseñar un programa con este paradigma es la facilidad de construcción, ya que es similar al juego de LEGO, donde uno piensa dónde encajan o se unen las piezas. Este paradigma tiene el mismo principio que es el de unir piezas de una forma simple.

En el momento en el que se habla de evolución existen diferentes formas de representar y darle sentido a dicho concepto [75]. Muchos hablan de que existe la manera Aristotélica de clasificación intrínseca a las definiciones; pero si analizamos la naturaleza, ésta no tiene un diseño tan rígido como los lenguajes basados en clases. El diseño en el software se da al momento de modelar el proceso y ajustarlo a alguna definición. Esto nos lleva a pensar que es importante que todos los procesos de activación se ajusten y puedan evolucionar en el mismo sentido que lo han realizado los objetos en los modelos físicos. Una manera de hacer esto, es por medio de la definición de actualización de un *Activador m*. Un ejemplo de este caso se observa en el Algoritmo 5.6, donde se hace uso del operador $\Leftarrow$.

Algoritmo 5.6 Actualización de una función de activación

```
let a:= [ numero = 0]
let b:= [ nuk=0, act=@(x)if ( > a.numero 5) nuk <= a.numero : [] ]
let c:= [ t=b.act <= @(x)if ( > a.numero 6) nuk <= a.numero [] ]
red d:= c.t
view b
```

En el ejemplo del Algoritmo 5.6 se observa que el objeto *c* hace una actualización de la función de activación de *b*. La reducción se define en la Semántica Operacional descrita en el Capítulo 4. Hay que decir que la operación de actualización no activa ningún objeto, debido a que podría darse un estado de inconsistencia o de inestabilidad por así llamarlo.

Capítulo 6

Conclusiones y trabajos futuros

6.1. Resumen

En este capítulo se presentan las conclusiones y trabajos futuros tanto del modelo como de la herramienta desarrollada.

6.2. Objetivos del capítulo

- Presentar las conclusiones.
- Listar los trabajos futuros.

6.3. Conclusiones

En este trabajo se buscó la descripción y formalización de un paradigma proactivo orientado a objetos. Esto se realizó para fundamentar las bases del funcionamiento de los objetos bajo el concepto de *activaciones*.

Dentro de la búsqueda por resolver problemas de cómputo en el ámbito de lo proactivo. Se formalizó y desarrolló una especificación orientada a objetos, la cual permite dar una solución a los problemas de cómputo de esta naturaleza de manera incremental. Asimismo, se describió una solución evolutiva que permite que los objetos se desarrollen en su contexto. También se presentaron algunas soluciones en el diseño de objetos mediante los diagramas del tipo UML (Lenguaje Unificado de Modelado).

En el desarrollo del lenguaje e interprete, se realizaron algunos estudios e implicaciones en el área de la computación. Aunque el cómputo proactivo es un área relativamente nueva, es imperante mostrar ciertas preeminencias que hacen de este paradigma proactivo orientado a objetos, una forma simple de resolver algunos problemas pertenecientes a este campo. De la misma manera es importante mencionar que se buscó la solución de algunos problemas de cómputo no proactivo con el uso del paradigma propuesto (El problema del estanque, semáforos, entre otros).

6.4. Trabajos Futuros

Durante el desarrollo del presente trabajo se encontraron varios aspectos que podrían ser desarrollados como trabajos futuros con el uso de este paradigma. A continuación se enumeran algunos de ellos:

1. Realizar algún desarrollo o adecuación de un lenguaje orientado a objetos con el paradigma propuesto. Se pretenderá llevar esta propuesta a un lenguaje ya existente con la creación de un compilador basado en una nueva especificación. Este desarrollo se hará en función del problema que se requiera resolver.

2. La formalización del paradigma proactivo orientado a objetos mediante el cálculo π. Lo que se buscará es establecer este paradigma para manejar la concurrencia.

3. Promover un estudio dentro de $\beta\varsigma - cálculo$ para la utilización de ciertos operadores de visibilidad, que eviten conflictos con los auto-llamados o la recursividad no controlada.

4. Impulsar un estudio con el uso de la herencia y se analizará la evolución de los objetos bajo estos mecanismos de clasificación.

5. Efectuar un estudio sobre la manera eficiente de implantar este paradigma bajo el concepto de Máquinas de Turing comparado con algún otro modelo de cómputo.

6. Analizar el paradigma con los conceptos de Clases y Prototipos. Lo que se buscará con este análisis es indicar las ventajas de una y otra implementación. También se hará uso de métodos empotrados y delegados en dicho análisis.

7. Desarrollar un sistema de transacciones mediante este paradigma. El objetivo general es el de resolver la restauración de valores si llegase a suceder alguna acción posterior.

8. Implantar algunos problemas pertenecientes al campo de la Inteligencia Artificial con este paradigma. Por dar algunos ejemplos: redes neuronales, algoritmos genéticos, agentes, entre otros.

9. Buscar otros mecanismos de activación en los objetos. Este mecanismo deberá ayudar a resolver otros problemas de diseño e implantación en el área de computación.

Apéndice A

Pro-Object

A.1. Resumen

En este Apéndice se presentan algunos aspectos sobre la construcción de la herramienta Pro-Object, que fue creada y usada durante la elaboración de este documento.

A.2. Objetivos del apéndice

- Construir bajo LL(n) un lenguaje denominado Pro-Object.
- Mostrar algunas consideraciones particulares en la implantación.

A.3. Diseño de Pro-Object

Durante el presente trabajo de tesis se elaboró una herramienta llamada Pro-Object, construida con base a las especificaciones realizadas en el Capítulo 4. Diseñada en dos fases, compilar y posteriormente interpretar el código de $\beta\varsigma - cálculo$.

Pro-Object se encuentra desarrollado en el lenguaje Java bajo el proyecto JavaCC *(Java Compiler Compiler*[1]), el cual permite definir la sintaxis de un lenguaje. *JavaCC* es un generador sintáctico del tipo LL(n) y algunas de sus características son:

- JavaCC crea programas de análisis sintáctico de arriba hacia abajo (Top-Down[2])

[1] https://javacc.dev.java.net/

[2] Es una palabra de uso común dentro del ámbito de los compiladores. Significa que es de arriba hacía abajo.

de una manera recursiva. Esto permite el uso de gramáticas más generales, aunque no se permite el uso de recursividad por la izquierda.

- Se permite definir especificaciones sintácticas y semánticas mediante LOOKAHEAD[3]. *JavaCC* genera por omisión un analizador del tipo LL(1), aunque es posible modificarlo para aceptar LL(n), esto permite reducir los conflictos de ambigüedad dentro del analizador.

A.4. BNF de Pro-Object

Se presenta una tabla en donde se encuentra la especificación BNF del lenguaje Pro-Object:

BeginParser	::=	(Assignation | Reduction | ViewObject)* <EOF>
Assignation	::=	<LET> JDefine
JDefine	::=	<IDENTIFIER> “:=“ expression
expression	::=	Object | Variable | Integer | Lambda | POperations | POperationr | POperationm | POperationd | BOperationIgual | BOperationMayor | BOperationMenor | Clone | PApplication | BTrue | BFalse | BAnd | BOr | BNot
BOperationMenor	::=	“<“ expression expression
BOperationMayor	::=	“>“ expression expression
BAnd	::=	“&&” expression expression
BOr	::=	“||” expression expression
BNot	::=	“!” expression
BTrue	::=	<TRUE>
BFalse	::=	<FALSE>
BOperationIgual	::=	“==“ expression expression
POperations	::=	“+” expression expression
POperationr	::=	“-” expression expression
POperationm	::=	“*” expression expression
POperationd	::=	“/” expression expression
PApplication	::=	“(“ (expression)* “)” (SelectUpdate)?
Clone	::=	<CLONE> “(“ expression “)” (SelectUpdate)?

[3]Instrucción usada para predecir el siguiente símbolo dentro del análisis sintáctico.

Lambda	::=	<LAMBDA> “(“ <IDENTIFIER> “)” expression (expression)*
SelectUpdate	::=	Update|UpdateIf|Select
UpdateIf	::=	“.” <IDENTIFIER> <UPDIF> FunctionSigma If (“-” Select)?
Select	::=	“.” <IDENTIFIER> (“(“ Parameter “)”)? (SelectUpdate)?
Parameter	::=	expression
Update	::=	“.” <IDENTIFIER> “<=“ (FunctionSigma)? expression (“-” Select)?
Variable	::=	<IDENTIFIER> (SelectUpdate)?
Integer	::=	<INTEGER>
Object	::=	“[“ ((aIf | Labels) (“,” (aIf | Labels))*)* “]” (SelectUpdate)?
aIf	::=	<IDENTIFIER> “=“ FunctionSigma If
If	::=	<IF> “(“ expression “)” expression “:” expression
Labels	::=	<IDENTIFIER> “=“ (FunctionSigma)? expression
FunctionSigma	::=	<SIGMA> “(“ <IDENTIFIER> “)”
ViewObject	::=	<VIEW> <IDENTIFIER>
Reduction	::=	<RED> JDefine

La definición BNF de Pro-Object mostrada en la Tabla anterior es una implantación simple, la cual permite experimentar con la especificación de $\beta\varsigma - cálculo$. Aunque en esta parte solamente se muestra la definición sintáctica y en el Capítulo 4 se especificó la semántica. En el siguiente apartado se analizarán algunos puntos relevantes durante la creación de dicho lenguaje.

A.5. Consideraciones para Pro-Object

En esta sección se hablará de algunos aspectos que son relevantes para el desarrollo de Pro-Object que fueron tomados en cuenta.

Se resolvió la mayoría de las ambigüedades del tipo LL(n) que podría generar el compilador. Aunque es importante destacar algunas propiedades que fueron incluidas para resolver estos problemas.

Dentro de $\beta\varsigma - cálculo$ se encuentra la definición de Update que es la de actualizar algún método y que sintácticamente quedó de la siguiente manera:

Update::=“.” <IDENTIFIER> “<=“ (FunctionSigma)? expression (“-” Select)?

Se observa en la definición de Update que existe un símbolo “-” en la sección (” − ”*Select*)?. Este símbolo fue agregado para evitar la ambigüedad conocida como la “*ambigüedad del else*” o “*dangling*”. Este problema consiste en resolver el Algoritmo mostrado en A.1.

Algoritmo A.1 Problema de ambigüedad con la sentencia else

```
if a then
if b then
s1
else
s2
```

En el Algoritmo A.1 se observa que el compilador tendría una ambigüedad al tratar de establecer, cuál de las dos sentencias if se relaciona con la sentencia $else$. En otras palabras, el $else$ a cuál de los dos if pertenece.

Con *Update* la ambigüedad se resuelve con el uso del símbolo “-”.

Algoritmo A.2 Solución al problema de la ambigüedad

```
let a:= [l1=5,l2=7]
red b:= a.l1 <= a.l2 - .l1
red c:= a.l1 <= a.l2.l1
```

Se muestra en el Algoritmo A.2, una forma simple del porqué se agrega el símbolo “-”, el cual evitará que la expresión muestre dicha ambigüedad.

A.6. Uso del intérprete

La presente tesis viene con un disco el cual contiene el intérprete Pro-Object versión 1. En su interior se encontrarán los siguientes programas ubicados en el directorio *bin*.

- POBJ.EXE que es un archivo ejecutable para plataformas Windows.
- POBJ.JAR que es un archivo que funciona con la Máquina Virtual de Java.

También se pueden hallar varios ejemplos que ayudan al uso del intérprete. Estos están localizados en la carpeta *example*.

Dentro de las primeras tres definiciones importantes que se colocaron en este intérprete, se encuentran: *let*, *red* y *view*. Donde *let* se utiliza para hacer referencia a cualquier objeto, *red* se utiliza para aplicar reglas de reducción y *view* se utiliza para ver el contenido de algún objeto. Esto se realiza mediante variables que contengan alguna referencia.

Se recomienda hacer algunas prácticas con los primeros ejemplos para involucrarse en la sintaxis y la semántica del intérprete. Para ejecutar cualquier ejemplo usar:

Para plataformas con el sistema operativo Windows:

>Pobj example1.jp

Para plataformas compatibles con Java (Unix, Linux):

>java –jar pobj example1.jp

Para MacOs X:

>./run.sh /examples/example1.jp

NOTA: Para cualquiera de las tres formas antes mencionadas se debe tener instalada la Máquina Virtual de Java. La versión utilizada es la 1.5. Aunque funciona correctamente con 1.6, pero deberá compilarse el código fuente para un correcto funcionamiento.

A.7. Ejemplo del uso del Intérprete

En esa sección se analizará un ejemplo del uso de la herramienta Pro-Object. El ejemplo que se ha escogido es el de los Numerales de Church. El cual se encuentra indicado en el Algoritmo A.3:

Algoritmo A.3 Código que representa a los Numerales de Church

```
let zero := [
      iszero = true,
      pred = @(x)x,
      succ = @(x)( clone(x).iszero <= false).pred <= clone(x)
]
red uno := zero.succ
red dos := uno.succ
view zero
view uno
view dos
```

En el Algoritmo A.3 se observa que existe un objeto llamado *zero* y que contiene tres métodos: El método *iszero* es utilizado para determinar cuál es la base de la inducción. El método *pred* es usado para guardar el objeto anterior; en el caso inicial o por definición es *zero*. Por último, el método *succ* que es el encargado de generar el siguiente objeto y mantener la referencia de si mismo (*self*) con el anterior.

La aplicación de la regla de reducción *zero.succ* da como resultado a un objeto definido como *one*; su estructura sintáctica se puede ver en el Algoritmo A.4:

Algoritmo A.4 Código que representa al numeral uno

```
one:= [ iszero=false ,
   pred= [
          iszero=true ,
          pred=@(x)x ,
          succ=@(x)(clone (x.iszero<=false).pred<=clone (x))
     ] ,
succ=@(x)(clone (x.iszero<=false).pred<=clone (x))
]
```

En el Algoritmo A.4 se presenta el objeto *one*, el cual contiene los mismos tres métodos que el objeto *zero*; con la salvedad de que el método *pred* ha sido actualizado, y en este momento contiene al objeto *zero*.

En la segunda reducción es la de *one.succ*, que presenta el Algoritmo A.3. Se toma como referencia al objeto *one* del Algoritmo A.4 para generar la reducción que se presenta en el Algoritmo A.5.

Algoritmo A.5 Código que representa al numeral dos

```
two:= [ iszero=false ,
     pred= [ iszero=false ,
          pred= [ iszero=true ,
               pred=@(x)x , succ=@(x)(clone (x.iszero<=false).pred<=clone (x))
          ] ,
     succ=@(x)(clone (x.iszero<=false).pred<=clone (x))
     ] ,
     succ=@(x)(clone (x.iszero<=false).pred<=clone (x))
]
```

En el Algoritmo A.5 el objeto numeral *two* contiene los mismos tres métodos que el numeral *one*, con la diferencia de que el método *pred* contiene al objeto numeral *one*, que a su vez contiene el objeto numeral *two*.

Como se observó en el ejemplo del Algoritmo A.3 existe en el cuerpo del método el uso de la función o método *clone*, la cual permite generar una copia del objeto x. Esta última función de copia se realiza en profundidad, esto es, se copian todas las referencias que contengan el objeto x.

Apéndice B

Propuesta para diagramar Pro-Object

B.1. Resumen

En este Apéndice se tratará de dar una introducción a algunas propuestas para realizar diagramas del tipo UML. Estos diagramas nos deben permitir definir a los objetos proactivos de una manera estática y dinámica.

B.2. Objetivos del capítulo

- Presentar diagramas estáticos para $\beta\varsigma - cálculo$.
- Presentar diagramas dinámicos para $\beta\varsigma - cálculo$.

B.3. Propuesta de un diagrama estático

En la propuesta que presenta UML existen varios diagramas que permiten modelar cualquier sistema. Dentro de estos dos tipos de diagramas existen: los estáticos y los dinámicos. En este apartado se describe una propuesta para diagramar objetos proactivos mediante diagramas estáticos.

Uno de los diagramas importantes en UML es el diagrama de clases, el cual permite tener una visión general de los objetos y sus relaciones. Estas relaciones no indican cómo se comunican los objetos, pero sí las posibles interacciones que existen entre ellos.

Existen varios tipos de relaciones dentro de los diagramas de clases. Los tres más importantes son:

- Asociación: Es la conexión entre clases.
- Dependencia: Se refiere a la relación de uso.
- Generalización/Especialización: Es una relación de herencia.

Los tres puntos antes mencionados, no presentan una clara visión del modelado ante los objetos proactivos definidos en el presente trabajo. Una forma de establecer o de representar dicha relación es proponer un diagrama, al cual se le llamará diagrama proactivo.

El diagrama proactivo tiene como objetivo mostrar la relación que existe entre objetos. Esta relación se encuentra dada por el aif que está definida en el Capítulo 4. La cual establece una condición que se realiza entre dos objetos. El primero se le llamará *activador* y el segundo será nombrado *activado*.

El objeto activador será el encargado de enviar una señal a todos los objetos que tengan una relación del tipo aif y que se encuentre dentro de su contexto.

El objeto activado ejecutará una de las dos opciones que se encuentran definidas en la función if.

Esta forma de realizar activaciones se puede realizar dentro de ambientes secuenciales, concurrentes o distribuidos. Estas soluciones fueron descritas en el Capítulo 4 y en el Capítulo 5 del presente trabajo.

Se utilizarán los mismos estereotipos que presenta UML, con la excepción de agregar uno nuevo que represente la relación de activación aif.

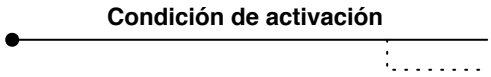

Figura B.1: Estereotipo de relación para objetos proactivos

Esta relación que se presenta en la Figura B.1 indica la manera en que dos objetos se encuentran relacionados. En un extremo se puede apreciar un • que indica el atributo o el área de atributos que intervienen en la condición, del otro lado se observan dos líneas, una completa y otra punteada. La línea continua indica qué método del objeto se ejecutará en caso de que la condición sea verdadera. La línea punteada indica qué método se ejecutará en caso que la condición de activación sea falsa.

Muchos de los diagramas UML, en especial el *diagrama de clases* muestran información sobre Clases e interacción estática. La dicotomía Objeto/Clase es factible que se dé, por lo que el tipo de diagramas que se propondrá será uno llamado *Diagrama de Objetos o Clases,* y cada uno de ellos representará su propio diseño. El diagrama de objetos tiene un parecido con el objeto de clases, pero a diferencia, muestra parte de la relación dinámica que éstos podrían presentar. La definición de un objeto mediante diagramas se puede ver en la Figura B.2.

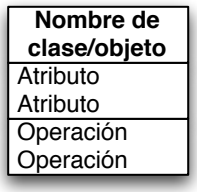

Figura B.2: Diagrama de objetos

Algunos aspectos fueron discutidos en el Capitulo 4, como son los operadores relacionales y lógicos que permiten que un aif se active. Sin embargo, a este tipo de diagramas no se les ha dotado de una visibilidad debido a dos razones principales.

1.- Si se llega a utilizar la visibilidad (*private*, *protected*, *public* y *default*) estos deberán manejar algún tipo de jerarquía lo que provocaría que existiera otro manejo en la cuestión de tipos y en especial el polimorfismo.

2.- Proponer otro mecanismo de visibilidad para que las activaciones se den con el manejo de cuantificadores. Aunque este punto se encuentra fuera del objetivo de la tesis, sí se hace una mención en el Capítulo 6.

B.4. Propuesta de un diagrama dinámico

Dentro de los diagramas UML se puede representar una manera dinámica de representar a los objetos. Este diagrama es conocido como de secuencia.

El diagrama de secuencia es uno de los diagramas para modelar la interacción entre los objetos de un sistema. Un diagrama de secuencia muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada caso de uso.

Para modelar a los objetos proactivos propuestos de manera dinámica, es necesario definir el contexto en el que se encuentran los objetos mencionados. Éstos pueden ser en un ambiente Secuencial, Concurrente o Distribuido.

Si se trata de crear un diagrama de secuencia en un ambiente secuencial, es importante señalar qué objeto tendrá prioridad ante otros objetos. Estos problemas se pueden apreciar en el diagrama de la Figura B.3 . Se podría suponer que los dos objetos se deben ejecutar al mismo tiempo (*Objeto2* y *Objeto3*), pero por tratarse de una máquina secuencial deberá ejecutarse uno primero y luego el otro. En este argumento no queda expresado cuál es el primer objeto que debe tener esa prioridad, si el *Objeto2* o el *Objeto3*. Es por este simple proceso que debe indicarse mediante un diagrama de secuencia proactiva, cuál de las dos clases debe tener prioridad sobre la otra.

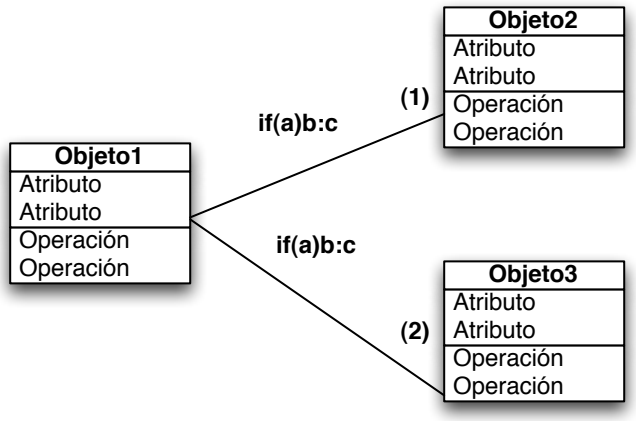

Figura B.3: Diagrama de secuencia y objetos proactivos

Este mismo fenómeno ocurre en los sistemas concurrentes de una máquina. Este sistema simula la ejecución de dos procesos, aunque solamente se ejecute uno en un tiempo dado. Por lo que se deberá definir cuál es el proceso que tiene mayor prioridad. Esto puede ser expresado en los diagramas de secuencia de UML.

En los sistemas distribuidos la ejecución es independiente, y por lo tanto existen varias máquinas ejecutándose al mismo tiempo para resolver el problema. Aunque cada objeto proactivo llamado Activado deberá registrarse ante la máquina que ejecute al objeto Activador; de esta manera, se puede mantener la referencia y resolver mediante algún mecanismo de distribución de objetos estos problemas.

Apéndice C
Ejemplos en Pro-Object

C.1. Resumen

En este Apéndice se presentarán una serie de ejemplos los cuales pueden ser experimentados con el interprete Pro-Objects. Este software se encuentra en el disco adjunto a este documento.

C.2. Objetivos del capítulo

- Presentar ejemplos básicos para la comprensión de $\beta\varsigma - cálculo$.
- Presentar ejemplos con activadores en $\beta\varsigma - cálculo$.
- Presentar ejemplos con clonación en $\beta\varsigma - cálculo$

C.3. Ejemplo básicos de reducciones en $\beta\varsigma - cálculo$

Se iniciará una serie de ejemplos básicos para conocer los alcances del paradigma propuesto. El símbolo de ς (sigma) por el de @ (arroba), y el símbolo λ (lambda) por el de \ (diagonal). El primer ejemplo es denominado transitividad, donde se presentan las tres principales directrices: let, red y view, estos lo puede observar en C.1.

Algoritmo C.1 Transitividad

```
let a := b
let b := c
red d := a
view d
//Resultado
//d:=c
```

El primer esquema de reducción se denomina selección y se puede observar en C.2.

Algoritmo C.2 Selección

```
let Objeto := [ valor = 3 ]
red resultado := Objeto.valor
view resultado
//Resultado
//3
```

El siguiente esquema de reducción se denomina uso de métodos y se puede observar en C.3.

Algoritmo C.3 Método

```
let Objeto := [ propiedad = 3, metodo = @(x) x.propiedad ]
red resultado := Objeto.metodo
view resultado
//Resultado
//3
```

El siguiente ejemplo muestra el uso de parámetros, estos son usados mediante la referencia de λ que en el código se representa por medio de una \ diagonal, observar el Algoritmo C.4.

Algoritmo C.4 Paso de parámetros

```
let Objeto := [ propiedad = 3, setValor = @(x) \(y) x.propiedad <= y ]
red Objeto := Objeto.setValor (5)
view Objeto
// Resultado
//Objeto:= [ propiedad = 5 , setValor = @(x)\(y)x.propiedad <= y ]
```

C.4. Ejemplos con activadores

En esta sección se analizará el uso de los activadores, el primer ejemplo que se observa es el uso de la activación de manera cerrada, esto es; que el resultado de la condicional modificará propiedades internas al objeto. Para analizar esta reducción se puede observar el Algoritmo C.5.

Algoritmo C.5 Activador cerrrado

```
let Objeto := [ valor =0 ]
let OtroObjeto := [ Activo = false, aif = @(x) if ( == Objeto.valor 2) x.Activo <=
true : [] ]
red Objeto := Objeto.valor <= 2
view OtroObjeto
//Resultado
//OtroObjeto:= [ Activo=true , aif=@(x) if ( == Objeto.valor 2 ) x.Activo<=true : [
] ]
```

El siguiente ejemplo muestra el uso de activadores, con la variante de modificar los valores de otros objetos, esto se observa en el Algoritmo C.6.

Algoritmo C.6 Activador libre

```
let Objeto := [ valor =0 ]
let OtroObjeto := [ Activo = false, aif = @(x) if ( == Objeto.valor 2) Objeto.valor <=
1 : [] ]
red Objeto := Objeto.valor <= 2
view Objeto
//Resultado
//Objeto:= [ valor=1 ]
```

Con los elementos y ejemplos antes mencionados se puede dar una solución al problema del Estanque basándose en activadores; recordando que lo que se busca es la programación incremental. Para ver esta solución observar el Algortimo C.7.

Algoritmo C.7 Una solución simple al problema del Estanque

```
let Estanque:= [ nivel=0 ]
let Sensor1:= [ active = false, aif1 = @(x) if ( == Estanque.nivel 1 ) x.active <= true
: [] ]
let Sensor2:= [ active = false, aif1 = @(x) if ( == Estanque.nivel 2 ) x.active <= true
: [] ]
let Sensor3:= [ active = false, aif1 = @(x) if ( == Estanque.nivel 3 ) x.active <= true
: [] ]
let Alarma:= [ sonido = false, aif1 = @(x) if ( == Sensor2.active true ) x.sonido <=
true : [] ]
red Estanque:= Estanque.nivel <= 2 view Alarma
//Resultado
//Alarma:= [ sonido=true , aif1=@(x) if ( == Sensor2.active true ) x.sonido<=true :
[ ] ]
```

C.5. Ejemplos con clonación

La clonación en $\beta\varsigma - cálculo$ por omisión de una clonación en profundidad, se le incorpora una clonación innata o inicial del objeto. El siguiente ejemplo muestra la funcionalidad de las dos, esto se puede observar en el Algoritmo C.8 .

Algoritmo C.8 Clonación en $\beta\varsigma - cálculo$

```
let Oveja:= [ Peso = 5, Color = blanco ]
red Dolly:= clone(Oveja)
red Dolly:= Dolly.Peso <= 7
view Oveja view Dolly
red DollyOriginal := iclone(Dolly)
view DollyOriginal
//Resultado
//Oveja:= [ Peso=5 , Color=blanco ]
//Dolly:= [ Peso=7 , Color=blanco ]
//DollyOriginal:= [ Peso=5 , Color=blanco ]
```

El Algoritmo C.9 muestra el uso de clonación superflua y en profundidad, observe que se agregó un carácter # al atributo para indicar que la clonación es superflua, ya que como se comentó los objetos proactivos realizan clonación en profundidad.

Algoritmo C.9 Clonación superficial y en profundidad

```
let Objeto1:= [ Comp# = 7 , Simp = 9 ]
red Objeto2:= clone(Objeto1)
red tmp:= Objeto2.Simp <= 3
red tmp:= Objeto2.Comp# <= 6
view Objeto1 view Objeto2
//Resultado
//Objeto1:= [ Comp#=6 , Simp=9 ]
//Objeto2:= [ Comp#=6 , Simp=3 ]
```

El siguiente ejemplo muestra otra propuesta a la solución del problema del Estanque, en esta opción intervienen la reducción de clonación. Esto se puede observar en el Algoritmo C.10.

Algoritmo C.10 Solución al problema del estanque con el uso de clonación

```
let Estanque:= [ nivel=0 ]
let Sensor1:= [ active = false, aif1 = @(x) if ( == Estanque.nivel 2 ) x.active <= true
: [] ]
let Alarma:= [ sonido = false, aif1 = @(x) if ( == Sensor1.active true ) x.sonido <=
true : [] ]
red Sensor2:= iclone(Sensor1) red Alarma2:= iclone(Alarma)
red Alarma2:= Alarma2.aif1 <<= @(r)if ( && (== Sensor1.active true) (== Sen-
sor2.active true) ) r.sonido <= true :[]
red Estanque:= Estanque.nivel <= 2
view Alarma view Alarma2
//Resultado
//Alarma:= [ sonido=true , aif1=@(x) if ( == Sensor1.active true ) x.sonido<=true :
[ ] ]
//Alarma2:= [ sonido=true , aif1=@(r) if ( && ( == Sensor1.active true)( == Sen-
sor2.active true) ) r.sonido<=true : [ ] ]
```

C.6. Disquisiciones en Pro-Objects

En esta sección se presentan ejemplos de algunas disquisiciones experimentales. La primera de ellas muestra el problema del Paro, éste es derivado por $\lambda - cálculo$. El Algoritmo se puede observar en C.11.

Algoritmo C.11 Problema de Paro

```
let a:= [ l1=@(x) x.l1 ]
red b:= a.l1
view b //No se ejecuta
```

El segundo Algoritmo C.12 muestra la *inestabilidad* de los objetos cuando las operaciones no son cerradas, esto es; cuando se tienen dos objetos A y B, y el objeto A se activa por alguna acción que sucede en B y viceversa, cuando el objeto B se activa por un suceso en A, el sistema presenta una recursividad, por lo que el sistema modelado se comporta de manera inestable.

Algoritmo C.12 Inestabilidad entre objetos

```
let a:=[ n=0, l1=@(x)if( > b.sn 0) a.n <= + a.n 1:[] ]
let b:=[ sn = 0, l=@(x)if( > a.n 0) x.sn <= + x.sn 1:[] ]
red d:= a.n<=1
view a
view b
//Resultado
//a:= [ n=3 , l1=@(x) if ( > b.sn 0 ) a.n<= + a.n 1 : [ ] ]
//b:= [ sn=2 , l=@(x) if ( > a.n 0 ) x.sn<= + x.sn 1 : [ ] ]
```

Como se puede observar en el Algoritmo C.12 los resultados de $a.n$ es igual a 3 y de $b.sn$ igual a 2.

Por último se presenta un ejemplo más amplio, recordando que únicamente se podrá hacer uso de las reglas básicas de $\beta\varsigma - cálculo$. Observar el algorítmo C.13.

Algoritmo C.13 Moviles en la calle

```
let Stack := [ isempty = true, top = 0, pop = @(s)s, push = @(s)\(x) ((s .pop <=
clone(s) ).isempty <= false).top <=x ]
let Semaforo1 := [ px#=20, py#=20, Color# = 1 , sin = @(x) if ( == Semafo-
ro2.Color# 1 ) x.Color# <= 1 : x.Color# <= 0 ]
let Semaforo2 := [ px#=10, py#=10, Color# = 1, sin = @(x) if ( == Semaforo1.Color#
1 ) x.Color# <= 1 : x.Color# <= 0 ]
let Semaforo3 := [ px#=30, py#=30, Color# = 0, sin = @(x) if ( == Semaforo1.Color#
1 ) x.Color# <= 0 : x.Color# <= 1 ]
red Stack := Stack.push ( Semaforo1 ).push ( Semaforo2 ).push ( Semaforo3 )
let beginIf := [ start = false, valor = Stack ]
let procIf := [ ifLabel = @(x)if( && (== beginIf.start true ) (== beginIf.valor.isempty
false)) (beginIf.valor <= beginIf.valor.pop) : [] ]
let Movil := [ mpx = 0, mpy = 0, incx = @(x) ((x.mbx <= x.mpx).mpx <= + x.mpx
1).Stat <= 0, incy = @(x) ((x.mby <= x.mpy).mpy <= + x.mpy 1).Stat <= 1, movx
= @(x) \(p) ((x.mbx <= x.mpx).mpx <= p).Stat <= 0, movy = @(x) \(p) ((x.mby
<= x.mpy).mpy <= p).Stat <= 1, mbx = 0, mby = 0, Stat = 0, return = false, re-
tx = @(x) (x.mpx <= x.mbx), rety = @(x) (x.mpy <= x.mby), reta = @(x) (x.mpx
<= x.mbx).mpy <= x.by, ifl = @(x)if( &&(&&(&&(== beginIf.start true) (== begi-
nIf.valor.isempty false)) (== beginIf.valor.top.px# x.mpx )) (== beginIf.valor.top.py#
x.mpy ) ) beginIf.start <= false : [], a1 = @(x)if( || (!(== x.Stat 0)) (!(== x.Stat 1))
) (beginIf.start <= true).valor <= Stack : [], if2 = @(x)if( &&(&&(&&(&&(&&(== be-
ginIf.start false) (== beginIf.valor.isempty false)) (== beginIf.valor.top.px# x.mpx ))
(== beginIf.valor.top.py# x.mpy )) (== beginIf.valor.top.Color# 1 ) ) x.return <=
true : [] , a2 = @(x)if ( && (== x.return true) (==x.Stat 0) ) (x.return <=false).retx
: [], a3 = @(x)if ( && (== x.return true) (==x.Stat 1) ) (x.return <=false).rety : [] ]
red Movil2 := iclone(Movil).movx(29).movy(30)
red Semaforo1 := Semaforo1.Color# <= 2
red Movil2 := Movil2.incx
red Movil := Movil.movx(19) red Movil := Movil.movy(20) red Movil := Movil.incx
red posActualX := Movil2.mpx red posActualY := Movil2.mpy
view posActualX view posActualY
```

Bibliografía

[1] Book review: Abstraction and specification in program development by barbara liskov and john guttag (mit press/mcgraw-hill, 1988, 469 pages, isbn 0-262-12112-3). *SIGSOFT Softw. Eng. Notes*, 11(3):79–81, 1986. Reviewer-Dan Conde.

[2] M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1995.

[3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[4] Martin Abadi and Luca Cardelli. A theory of primitive objects. draft, October 1993.

[5] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In Donald Sannella, editor, *Proceeding of ESOP '94 on Programming Languages and Systems*, volume 788, pages 1–25. Springer Verlag, 1994.

[6] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[7] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Inf. Comput*, 125(2):78–102, 1996.

[8] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *J. Funct. Program*, 5(1):111–130, 1995.

[9] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *POPL*, pages 396–409, 1996.

[10] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst*, 28(2):207–255, 2006.

[11] Gregory R. Andrews and Fred B. Schneider. Corrigenda: "Concepts and notations for concurrent programming". *ACM Computing Surveys*, 15(4):377–377, December 1983. See [**?**, **?**, **?**].

[12] Michael Backes, Christian Cachin, and Reto Strobl. Proactive secure message transmission in asynchronous networks. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 223–232, New York, NY, USA, 2003. ACM.

[13] John W. Backus. The IBM 701 speedcoding system. *J. ACM*, 1(1):4–6, January 1954.

[14] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1981 (1st ed) revised 84.

[15] H. P. Barendregt. Functional programming and lambda calculus. pages 321–363, 1990.

[16] Henk P. Barendregt. Introduction to lambda calculus. *Nieuw Archief voor Wiskunde*, 4(2):337–372, 1984.

[17] Emery D. Berger. FP+OOP=Haskell. Technical Report TR-92-30, The University of Texas at Austin, Department of Computer Science, December 1991.

[18] Andrew Black and Jens Palsberg. Foundations of object-oriented languages. *SIGPLAN Not.*, 29(3):3–11, 1994.

[19] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[20] Gary C. Borchardt. Event calculus. In *IJCAI*, pages 524–527, 1985.

[21] A. H. Borning. Classes versus prototypes in object-oriented languages. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 36–40, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[22] T. Bruckschlegel. Microbenchmarking C++, C#, and Java. *C/C++ Users Journal*, 23(7):14–21, 2005.

[23] Eric J. Byrne. A conceptual foundation for software re-engineering. In *Proceedings of the 1992 Conference on Software Maintenance (CSM '92),* (Orlando, Florida; November 9-12, 1992). IEEE Computer Society Press (Order Number 2980), November 1992.

[24] J. A. Campbell, editor. *Implementations of Prolog*. Series in Artificial Intelligence. Ellis Horwood, 1984.

[25] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[26] Cardelli, Ghelli, and Gordon. Types for the ambient calculus. *INFCTRL: Information and Computation (formerly Information and Control)*, 177, 2002.

[27] L. Cardelli. Types for data-oriented languages. *Lecture Notes in CS*, 303:1, April 1988.

[28] Luca Cardelli. The functional abstract machine. *Polymorphism*, 1(1), January 1983.

[29] Luca Cardelli. The functional abstract machine. *Bell Laboratories*, 1(1):1–45, 1983.

[30] Luca Cardelli. Compiling a functional language. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 208–217, New York, NY, USA, 1984. ACM.

[31] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[32] K. Mani Chandy, Michel Charpentier, and Agostino Capponi. Towards a theory of events. In Hans-Arno Jacobsen, Gero Mühl, and Michael A. Jaeger, editors, *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems, DEBS 2007, Toronto, Ontario, Canada, June 20-22, 2007*, pages 180–187. ACM, 2007.

[33] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*, pages 200–208, 1990.

[34] Fintan Culwin. Object imperatives! In Daniel Joyce, editor, *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, volume 31.1 of *SIGCSE Bulletin*, pages 31–36, N. Y., March 24–28 1999. ACM Press.

[35] J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors. *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[36] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[37] Mehmet Dincbas. Constraint programming. *ACM Computing Surveys*, 28(4es):62, December 1996.

[38] Eugene Eberbach and Peter Wegner. Beyond turing machines. *Bulletin of the EATCS*, 81:279–304, 2003.

[39] Fisher, Honsell, and Mitchell. A lambda calculus of objects and method specialization. In *Nordic Journal of Computing*, volume 1. 1994.

[40] Man's Search for Meaning. *Viktor E.Frankl*. USA, 2006.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[42] Goldin and Wegner. The church-turing thesis: Breaking the myth. In *Conference on Computability in Europe (CiE), LNCS*, volume 1, 2005.

[43] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1979.

[44] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with Uml,Volume 1*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[45] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP*, pages 235–246, 2002.

[46] J. Guttag. Notes on type abstraction. *IEEE Transactions on Software Engineering*, SE-6(1):13–23, January 1980.

[47] Paul Hankin. A study of objects, October 03 1999.

[48] M. Hennessy. *The Semantics of Programming Languages: an elementary introduction using structural operational semantics.* Wiley, 1990.

[49] Hoare. Corrigendum: Communicating sequential processes. *CACM: Communications of the ACM*, 21, 1978.

[50] C. A. R. Hoare. Hints on programming language design. In Anthony I. Wasserman, editor, *Tutorial Programming Language Design*, pages 43–52. IEEE, October 1980.

[51] Ian M. Holland and Karl J. Lieberherr. Object-oriented design. *ACM Comput. Surv.*, 28(1):273–275, 1996.

[52] Jan Rune Holmevik. Compiling SIMULA: A historical study of technological genesis. *IEEE Annals of the History of Computing*, 16(4):25–??, Winter 1994.

[53] R. C. Holt, P. A. Matthews, J. A. Rosselet, and J. R. Cordy. *The Turing Programming Language, Design and Definition.* Prentice-Hall, 1988.

[54] P. Hudak, J. Peterson, and J. Fasel. A gentle introduction to haskell, version 98., September 1999.

[55] D. H. H. Ingalls. Design principles behind smalltalk. *BYTE*, 6(8):286–298, August 1981.

[56] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[57] Bjørn Kirkerud. *Object-Oriented Programming with Simula.* Addison-Wesley, Reading, MA, 1989.

[58] Charles W Krueger. Software reuse. *Computing Surveys*, 24:131–83, June 1992.

[59] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[60] René Meier and Vinny Cahill. Location-aware event-based middleware: A paradigm for collaborative mobile applications? In *8th CaberNet Radicals Workshop*, October 2003.

[61] Bertrand Meyer. Genericity versus inheritance. *SIGPLAN Not.*, 21(11):391–405, 1986.

[62] M.Fernandez. *Programming Languages And Operational Semantics: An Introduction*, volume 1. King's College Publications, 2007.

[63] G. Michaelson. *An Introduction to Functional Programming Through Lambda Calculus*. Addison-Wesley, 1989.

[64] Fiona Morgan, Computer Science Tripos, and Part Ii. Implementation of an imperative object calculus, August 03 2000.

[65] Mosses. A practical introduction to denotational semantics. In *Neuhold & Paul (Eds.), Formal Description of Programming Concepts, Springer-Verlag*. 1991.

[66] Bruce J. Neubauer and Dwight D. Strong. The object-oriented paradigm: more natural or less familiar? *J. Comput. Small Coll.*, 18(1):280–289, 2002.

[67] Niehren, Schwinghammer, and Smolka. A concurrent lambda calculus with futures. *TCS: Theoretical Computer Science*, 364, 2006.

[68] Joachim Niehren, David Sabel, Manfred Schmidt-Schauß, and Jan Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures, May 11 2007.

[69] Jeremiah S. Patterson. An object-oriented event calculus. Technical Report 02-08, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, July 2001.

[70] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL'96 — Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg, Florida, January 1996. ACM.

[71] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

[72] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.

[73] Alessandro Provetti. Hypothetical reasoning about actions: From situation calculus to event calculus. *Computational Intelligence*, 12:478–498, 1996.

[74] Aarne Ranta. Type theory and the informal language of mathematics. In Hendrik Barendregt and Tobias Nipkow, editors, *Selected papers from TYPES'93: Int.*

Workshop on Types, Nijmegen, The Netherlands, volume 806 of *LNCS*, pages 352–365. Springer-Verlag, 1994.

[75] Derek Rayside and Gerard T. Campbell. An aristotelian understanding of object-oriented programming. In *OOPSLA*, pages 337–353, 2000.

[76] Regev, Panina, Silverman, Cardelli, and Shapiro. Bioambients: An abstraction for biological compartments. *TCS: Theoretical Computer Science*, 325, 2004.

[77] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

[78] C. Reinke. Towards a Haskell/Java connection. *Lecture Notes in Computer Science*, 1595:200–215, 1999.

[79] J. C. Reynolds. Polymorphic lambda-calculus: Introduction. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, Reading, MA, 1990.

[80] R.O.Gandy and C.E.M.Yates. Collected works of a.m. turing : Mathematical logic. 1(294), 1994.

[81] W. A. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[82] Davide Sangiorgi and David Walker. *The Pi-Calculus — A Theory of Mobile Processes*. Cambridge University Press, 2001.

[83] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today*, pages 409–430. 1999.

[84] Mary Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Softw. Eng. Notes*, 20(1):27–38, 1995.

[85] Ryan Stansifer. Imperative versus functional. *SIGPLAN Notices*, 25(4):69–72, 1990.

[86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 1986.

[87] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[88] Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[89] David L. Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000.

[90] A. Thayse, editor. *From Standard Logic to Logic Programming.* John Wiley & Sons, 1988.

[91] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison Wesley, July 1996.

[92] Walker. A theory of aspects. *SPNOTICES: ACM SIGPLAN Notices*, 38, 2003.

[93] P. Wegner. Research paradigms in computer science. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 322–330. IEEE Computer Society Press, October 1976.

[94] P. Wegner. Models and paradigms of interaction. *Lecture Notes in Computer Science*, 791:1–??, 1994.

[95] Peter Wegner. Programming language semantics. In R. Rustin, editor, *Formal Semantics of Programming Languages*, pages 149–248. Prentice-Hall, 1972.

[96] Peter Wegner. Classification in object-oriented systems. *SIGPLAN Notices, ACM*, 21(10), October 1986.

[97] Peter Wegner. The object-oriented classification paradigm. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming.* The MIT Press, 1987.

[98] Peter Wegner. Paradigms of interpretation and modeling. Technical Report CS-91-09, Brown University, 1991.

[99] Ashley Williams. Assessing prototypes' role in design. In *SIGDOC '02: Proceedings of the 20th annual international conference on Computer documentation*, pages 248–257, New York, NY, USA, 2002. ACM.

[100] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

[101] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Rice University, February 1993.