Universidad Nacional del Centro de la Provincia de Buenos Aires

Análisis y diseño de algoritmos I
Ciencias de la computación II

Una herramienta para una introducción a la verificación formal de programas

Autores
Tomás Juárez
Guillermo Pacheco

Directores
Mg. Virginia Mauco
Dra. Laura Felice

1 TABLA DE CONTENIDOS

1 MOTIVACIÓN

El siguiente informe da cuenta del proyecto final en conjunto de las materias análisis y diseño de algoritmos I y ciencias de la computación II, dictadas en la facultad de Ciencias Exactas de la Universidad Nacional del Centro de la provincia de Buenos Aires.

El objetivo de este trabajo es realizar una herramienta grafica que aplique conceptos vistos en el área de las ciencias informáticas, modelando e implementando la misma aplicando los conocimientos adquiridos en la materia de Análisis y Diseño de Algoritmos I. En este caso, el tema a aplicar es la *verificación formal de programas Iterativos*. Esta temática no fue vista durante el transcurso de la cursada de la materia Ciencias de la Computación, pero algunos de los conceptos que se aplican, si lo son. Este es el caso de los árboles de refutación, uno de los temas centrales de la materia.

El desarrollo de la herramienta se realizó haciendo énfasis en los temas centrales de la materia de Diseño y Análisis de algoritmos I, es por eso que se han tenido en cuenta la *complejidad temporal*, los *tipos de datos abstractos* y el concepto de *clase*.

Una de las principales motivaciones que nos llevó a realizar este trabajo, es el hecho de que por primera vez en nuestra carrera universitaria, ponemos en práctico lo aprendido para desarrollar una aplicación compleja con entorno gráfico. Otra razón, fue la posibilidad de poder desarrollar una herramienta que puede servir de ayuda a futuros estudiantes que cursen la materia, ya que una herramienta visual es de gran ayuda para la comprensión y entendimiento del alumno.

Durante la codificación del trabajo nos hemos encontrado con problemas que se escapaban de nuestros conocimientos. Es por ello que muchos de éstos se podrían haber resuelto de una manera más elegante e incluso eficiente. En muchos casos el tratamiento de estos problemas se ve a finales de la carrera, como es el caso del *parser*.

2 BASES TEÓRICAS

En esta sección se cubren algunos temas claves en nuestro trabajo. Es muy importante conocerlos para entender cómo se llevó a cabo la construcción del software.

2.1 LÓGICA PROPOSICIONAL

2.1.1 Introducción

La lógica proposicional estudia la verdad o falsedad de las proposiciones y de cómo la verdad se transmite de unas proposiciones (premisas) a otras (conclusión). Una proposición es la unidad mínima de significado susceptible de ser verdadera o falsa.

2.1.2 Sintaxis

El alfabeto de la lógica proposicional para la definición de fórmulas puede definirse como $A\_prop = VAR \cup \{\neg, \rightarrow, \wedge, \vee\} \cup \{(,)\}$ en donde VAR representa al conjunto de variables o símbolos proposicionales, $\{\neg, \rightarrow, \wedge, \vee\}$ representa los conectivos proposicionales (NOT, IMPLICACIÓN, AND y OR respectivamente) y $\{(,)\}$ es el conjunto de símbolos auxiliares.

El lenguaje de la lógica proposicional puede definirse como sigue:

```
<form_log>   ::= <form_log> → <form_or> | <form_or>
<form_or>    ::= <form_or>  ∨ <form_and> |<form_and>
<form_and>   ::= <form_and> ∧ <factor_log> | <factor_log>
<factor_log> ::= (<form_log>) | ¬<factor_log> | <var_prop>
<var_prop>   ::= a | b | c | ... | z
```

De estas reglas obtenemos una fórmula $F_m \subseteq \mathrm{A}_{\mathrm{prop}}$.

2.1.3 Semántica

Para cada $F_m \subseteq \mathrm{A}_{\mathrm{prop}}$ bien definido, tenemos que cada una de ellas puede interpretarse su valor de verdad teniendo en cuenta el significado de los conectivos que contienen. Los conectivos se interpretan como sigue:

Conjunción: $p \wedge q = 1$ si $p = q = 1$. En cualquier otro caso, $p \wedge q = 0$

Disyunción: $p \vee q = 1$ si $p = 1$ o $q = 1$. En cualquier otro caso, $p \vee q = 0$

Implicación: $p \rightarrow q = 0$ si $p = 1$ y $q = 0$. En cualquier otro caso, $p \rightarrow q = 1$

Negación: $\neg p = 0$ si $p = 1$, y $\neg p = 1$ si $p = 0$

2.1.4 Valuaciones

Una valuación es una función $v: F_m \rightarrow \{0,1\}$ tal que, para todo $A, B \in F_m$ cumple con las siguientes propiedades:

- $v(\neg A) = \neg v(A)$
- $v(A \wedge B) = v(A) \wedge v(B)$
- $v(A \vee B) = v(A) \vee v(B)$
- $v(A \rightarrow B) = v(A) \rightarrow v(B)$

2.1.5 Definiciones importantes

- Una *tautología* es una fórmula A que es verdadera bajo toda valuación $(v(A) = 1)$. En símbolos $\models A$.
- Una *contradicción* es una fórmula A que es falsa bajo toda valuación $(v(A) = 0)$.
- Una *contingencia* es una fórmula A en donde algunas de sus valuaciones son falsas y otras verdaderas.

2.2 Lógica de Predicados de Primer Orden

2.2.1 Introducción

Los principales objetivos de la lógica proposicional se basan en modelar el proceso de razonamiento y formalizar la noción de demostración. Los agentes basados en el conocimiento funcionan razonando con una representación del conocimiento sobre el mundo y sus acciones. En algunos casos el poder expresivo de la lógica proposicional es limitado, por lo que necesitamos introducir una lógica más expresiva.

2.2.2 Lenguaje de Primer Orden

En la lógica de primer orden existen sentencias representando hechos y términos que representan objetos. Se hace uso de símbolos de *constante*, *predicado*, *función*, *variables*, *conectivas lógicas* y *cuantificadores*.

Los símbolos de constante específica a qué objeto se refieren sabiendo que una constante se refiere a un solo objeto, puede haber dos constantes haciendo referencia al mismo objeto y no es necesario nombrar todos los objetos del universo. Se representan como $C = \{c_1, c_2, \dots, c_m\}$.

Los símbolos de predicado especifican a qué relación hacen referencia. Formalmente toda relación se define mediante el conjunto de tuplas que la satisfacen. Se representan como $R = \{R_1, R_2, \dots R_k\}$.

Los símbolos de función establecen la relación funcional a la que hacen referencia. Una relación funcional también puede definirse mediante un conjunto de tuplas. Se representan como $F = \{f_1, f_2, \dots, f_n\}$. Tanto para las funciones como para las relaciones, definimos la *aridad* como el número de argumentos que estas posean, en donde siempre serán mayores que 0.

La semántica de los conectivos lógicos es el mismo que en la lógica proposicional y en el caso de los cuantificadores, estos son derivados de las sentencias con conectivas. Los conectivos lógicos son ¬, →, ∧, v y los cuantificadores son ∃, ∀ *existe* y *para todo*, respectivamente. Estos dos últimos son muy importantes ya que juntos con las relaciones nos da ese poder de expresión que la lógica proposicional no tiene. Con estos dos cuantificadores podemos indicar cuántos o qué tipo de elementos de un conjunto (universo) cumplen con cierta propiedad.

El lenguaje de primer orden se define como $L = \langle R, F, C \rangle$. Un ejemplo de fórmulas bien formadas en el lenguaje de primer orden sería $\forall(x, y)\left(Q(x) \rightarrow R\big(x, f_1(y)\big)\right)$.

2.3 Árboles de Refutación

2.3.1 Introducción

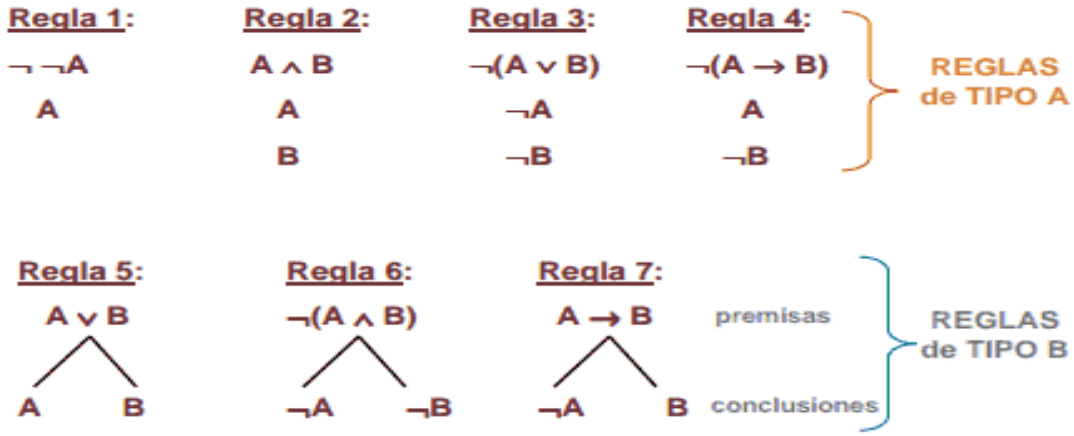
El método de *demostración por contradicción* o *reducción al absurdo* nos permite utilizar las llamadas tablas semánticas para comprobar si un argumento es válido o no. Éste permite saber si una proposición es una contradicción basándose en la estrategia de refutación, es decir, $\nvDash B$ sí y sólo sí $\neg B$ es contradicción y $A \nvDash B$ sí y sólo sí $\{A, \neg B\}$ es insatisfacible $(A = \{A1, A2, \dots, An\})$.

Los árboles de refutación son la base de nuestra aplicación ya que la verificación de programas utiliza esta técnica.

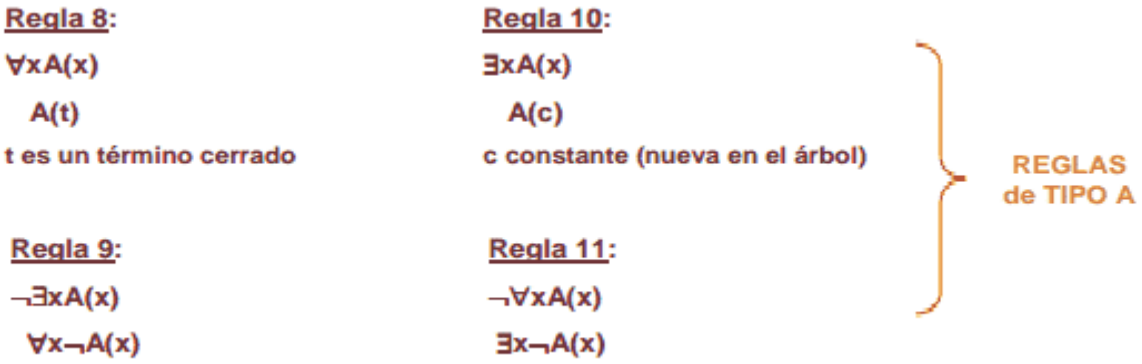
2.3.2 Reglas

Dada una formula F, su árbol de refutación se obtiene haciendo un número finito de aplicaciones inmediatas de las reglas, partiendo del árbol cuyo único nodo es F. Sean A, B formulas de la lógica de

predicados de primer orden, definimos las siguientes reglas:



Reglas que involucran cuantificadores (todas son reglas de tipo A):



En el caso de que se quiera obtener un árbol de refutación de un conjunto finito S de fórmulas, este sería el árbol de refutación de la conjunción de todas las formulas pertenecientes a S.

2.3.3 DEFINICIONES

Al analizar un árbol de refutación surgen los conceptos de *rama abierta*, *rama cerrada*, *rama saturada*, *árbol cerrado* y *árbol completo*.

- Una rama de un árbol de fórmulas se dice cerrada si contiene a la vez una fórmula y su negación. En caso contrario, se dice abierta.
- Un árbol se dice cerrado si todas sus ramas son cerradas.
- Una rama se dice saturada si no es cerrada y no se puede expandir utilizando las reglas.
- Un árbol se dice completo si cada una de sus ramas es cerrada o saturada.
- Un conjunto C de fórmulas se dice saturado sí y sólo sí satisface las siguientes condiciones:

- Una fórmula y su negación no pueden estar simultáneamente en C.
- Si una fórmula de tipo A está en C, entonces sus dos conclusiones están en C, excepto para la regla 1 que es una sola conclusión.
- Si una fórmula de tipo B está en C, entonces al menos una de sus dos conclusiones están en C.

- Una rama de un árbol de fórmulas se dice saturada si el conjunto formado por las fórmulas de sus nodos es un conjunto saturado.

Con estas definiciones, obtenemos las siguientes deducciones:

- Toda fórmula F tiene por lo menos un árbol de refutación completo.
- Si F es una fórmula satisfacible, entonces todo árbol de refutación de F tiene al menos una rama abierta.
- Si $\neg F$ tiene un árbol de refutación cerrado, entonces F es lógicamente válida.
- Si un conjunto finito $S = \{H_1, H_2, \dots, H_n\}$ de lógica de Predicados de primer orden, tiene un árbol de refutación cerrado, entonces S es insatisfacible.
- Si $\{H_1, H_2, \dots, H_n, \neg C\}$ tiene un árbol de refutación cerrado, entonces $H_1, H_2, \dots, H_n \models C$.

2.4 BNF

2.4.1 INTRODUCCIÓN

La notación *Backus-Naur form* (BNF) es un metalenguaje usado para expresar gramáticas libres de contexto o una manera formal de describir lenguajes formales. Esta notación es sumamente necesaria para describir un analizador sintáctico, que es el pilar de este trabajo y se comentará más adelante.

2.4.2 SINTAXIS

Una especificación de BNF es un sistema de reglas de derivación. Se hace uso de símbolos terminales $T = \langle t_1, t_2, \dots, t_m \rangle$ y símbolos no terminales $N = \langle n_1, n_2, \dots, n_r \rangle$, corchetes para describirlos y una barra vertical | para introducir opciones entre símbolos. Se escribe de la siguiente manera:
$\langle t_i \rangle ::= \langle t_j \rangle | \langle t_k \rangle | \dots | \langle n_l \rangle$ con $i, j, k \in \{0,1, \dots m\}$ y $l = 0,1, \dots r$.

La recursión en caso de que un símbolo no terminal tenga símbolos no terminales como opción representa un problema en nuestra aplicación que se comentará luego.

2.5 COMPLEJIDAD TEMPORAL

2.5.1 INTRODUCCIÓN

La *teoría de la complejidad computacional* es una rama de las ciencias de la computación que se centra en la clasificación de los problemas computacionales de acuerdo a su dificultad inherente, y en la relación entre dichas clases de complejidad. En otras palabras, se estudia la eficiencia de los algoritmos de forma teórica, independientemente del ordenador en el cual se ejecuten. Un algoritmo es una especificación rigurosa de un conjunto de operaciones o instrucciones ejecutadas sobre un autómata en un tiempo finito. Una vez que se plantea un algoritmo para la resolución de un problema, un paso importante es determinar, en lo que a recursos respecta, cuánto espacio y tiempo requerirá. Podemos obtener un tiempo de ejecución teórico de un algoritmo determinado y a esto lo llamamos *eficiencia asintótica*, es decir, observamos como aumenta la cantidad de trabajo realizada por un algoritmo a medida que se incrementa el tamaño de la entrada n con valores suficientemente grandes.

2.5.2 DEFINICIONES

Podemos representar la eficiencia asintótica de un algoritmo T con una notación conveniente en función de la longitud de la estructura de entrada N, dividiéndose ésta en 3 posibles casos: el mejor caso $\Omega(g(n))$, el caso promedio $\Theta(h(n))$, y el peor caso $O(f(n))$. A estas notaciones, se las conoce como *Big-Omega*, *Big-Tetha* y *Big-Oh* respectivamente y su definición formal se da a continuación:

$$T(N) = \begin{cases} O(f(n)) \iff (\exists m, c \in \mathbb{R}^+, N \geq m)[T(N) \leq c \cdot f(N)] \\ \Omega(g(n)) \iff (\exists m, c \in \mathbb{R}^+, N \geq m)[T(N) \geq c \cdot g(N)] \\ \Theta(h(n)) \iff T(N) = O(h(n)) = \Omega(h(n)) \end{cases}$$

Particularmente, nos interesa obtener el peor caso del algoritmo. Para ello debemos tener en cuenta las siguientes fórmulas, dependiendo de la estructura del algoritmo.

En el caso de un algoritmo iterativo tenemos las siguientes fórmulas

$$\text{WhileLoop} = \text{T}_{\text{condicion}} + \sum_{i=1}^{\text{N}} (\text{T}_{\text{cuerpo}} + \text{T}_{\text{condicion}})$$

$$\text{ForLoop} = \sum_{i=1}^{\text{N}} (\text{T}_{\text{cuerpo}} + \text{T}_{\text{condicion}})$$

$$\text{IfElse} = \text{T}_{\text{condicion}} + \max(\text{T}_{\text{if}}, \text{T}_{\text{else}})$$

En el caso de un algoritmo recursivo podemos reconocer dos casos diferentes. Si el algoritmo subdivide el problema en partes más pequeñas, utilizando el *teorema Maestro:*

$$\text{T(N)} = \begin{cases} \text{N} = k \Rightarrow \text{T}_{\text{funcion}} \\ \text{N} > k \ \Rightarrow a \cdot \text{T}\left(\frac{\text{N}}{b}\right) + \ \text{T}_{\text{funcion}} \end{cases}$$

Siendo $a, b, k \in \mathbb{N}, a \geq 1$ la cantidad de veces que se hace la llamada recursiva y $b > 1$ el número en el cual se subdivide el problema. Ahora bien, si el algoritmo recorre de forma exhaustiva la estructura de entrada:

$$\text{T(N)} = \begin{cases} \text{N} = k \Rightarrow \text{T}_{\text{funcion}} \\ \text{N} > k \ \Rightarrow \ \text{T}(\text{N} - 1) + \ \text{T}_{\text{funcion}} \end{cases}$$

Siendo $k \in \mathbb{N}$.

En general, $k = 0 \vee k = 1$ *y* $\mathrm{T}_{\mathrm{funcion}} = O(1)$.

El tiempo en operaciones lógicas, aritméticas, declaraciones, inclusiones de bibliotecas y asignaciones es siempre $O(1)$, y son las llamadas *operaciones elementales.*

2.6 Verificación Formal de Programas

2.6.1 Introducción

La prueba formal de programas es una técnica que se basa en el cálculo de predicados. Primero, se debe describir formalmente el comportamiento de cada instrucción del lenguaje, es decir, se debe definir la semántica de un lenguaje de programación en términos de fórmulas lógicas. Para probar un programa se debe expresar su semántica en términos de fórmulas lógicas y luego probar que el programa significa lo mismo que su especificación. Una prueba formal de un programa asegura que el programa es correcto con respecto a una especificación para todas las entradas.

Una variable se usa en un programa para describir una posición de memoria que puede contener valores diferentes en diferentes estados. Una manera de describir un conjunto de estados es utilizando fórmulas del cálculo de predicados. Estas fórmulas se denominan aserciones, son verdades cada vez que el programa pase por ese punto. La pre-condición P y post-condición Q son aserciones que cumplen con la siguiente definición: $\{P\}S\{Q\}$ siendo S un fragmento del programa. Si la ejecución de S empieza en un estado caracterizado por P y S termina, entonces terminará en un estado caracterizado por Q.

Para todo fragmento de programa S y fórmula Q, se define la pre-condición más débil $R = wp(S, Q)$ tal que $\{R\}S\{Q\}$ es verdadero. Es decir, $wp(S, Q)$ representa todos los estados tal que la ejecución de S que comenzó en cualquiera de ellos, si termina, termina en un estado que satisface Q. Este también se denomina predicado transformador, ya que para cualquier fragmento de programa define una transformación de un predicado post-condición en un predicado pre-condición. Es decir, en vez de describir cómo un programa transforma un conjunto de estados iniciales en un conjunto de estados finales, describe cómo un programa transforma un predicado post-condición, que caracteriza el conjunto de estados finales, en un predicado pre-condición que caracteriza el conjunto de estados iniciales.

La metodología que se aplica en este proyecto permite verificar algoritmos a partir de su especificación de una manera totalmente constructiva a través de la obtención de un árbol de refutación de la forma $P \nvdash \neg wp(A, Q)$, donde $\neg wp(A, Q)$ es la denominada precondición más débil del algoritmo A con respecto a la post-condición Q, la cual se define como el predicado menos restrictivo que asegura que si un estado lo satisface entonces después de ejecutar el algoritmo A ,el predicado Q se verifica.

En este proyecto nos limitamos a verificar las sentencias *if*, *if-else* y *while.*

2.6.2 SENTENCIA IF

Si C_1 es una parte de un programa y B es una condición, entonces la sentencia interpreta de la siguiente forma: *si B es verdadero se ejecuta C_1*.

Para demostrar la corrección de una sentencia *if* con una precondición $\{P\}$ y una postcondición $\{Q\}$ tendremos dos posibilidades; si el estado inicial satisface a B además de P entonces se ejecutará C_1 y por tanto la verificación equivaldrá demostrar que $\{P \wedge B\}C_1\{Q\}$ es correcto o bien, si el estado inicial no satisface B entonces esto equivaldrá demostrar que $\{P \wedge \neg B\} \rightarrow \{Q\}$.

2.6.3 SENTENCIA IF-ELSE

Si C_1 y C_2 son dos partes de un programa y si B es una condición, entonces la sentencia se interpreta de la siguiente forma: *si B es verdadero se ejecuta C_1 sino C_2*.

Para demostrar la correctitud de una sentencia *if-else* con una precondición $\{P\}$ y una postcondición $\{Q\}$ tendremos dos posibilidades; si el estado inicial satisface B además de P entonces se ejecutará C_1 y por tanto la verificación equivaldrá demostrar que $\{P \wedge B\}C_1\{Q\}$ es correcto, o bien, si el estado inicial no satisface B entonces se ejecutará C_2 y por tanto la verificación equivaldrá demostrar que $\{P \wedge \neg B\}C_2\{Q\}$ es correcto.

2.6.4 SENTENCIA WHILE

Si B es la condición del bucle y S es parte del programa, entonces para demostrar la corrección de esta sentencia se deberán cumplir las siguientes pruebas:

- El invariante se cumple antes de la primera iteración del bucle, $\{P\}A\{I\}$.
- Mientras se ejecuta el cuerpo del bucle A, el invariante se mantiene, $\{I \wedge b\}A\{I\}$.
- El invariante se sigue cumpliendo al salir del bucle, $I \wedge \neg B \rightarrow Q$.
- C será una función dependiente de las variables del bucle que garantice que este termina. En este caso mientras se cumpla la condición B, C será mayor que cero, $I \wedge B \rightarrow C \geq 0$.
- C decrecerá al ejecutarse el cuerpo del bucle, $\{I \wedge B \wedge C = T\}A\{C < T\}$.

3 TIPOS DE DATOS ABSTRACTOS

3.1 INTRODUCCIÓN

Una confusión habitual es considerar que el concepto matemático de conjunto es el más indicado para explicar qué es un tipo de datos; sin embargo, el concepto matemático más apropiado para explicar la naturaleza de un tipo de datos es el de álgebra. Un tipo de datos es un álgebra, es decir, un conjunto de valores, llamado dominio, caracterizados por el conjunto de operaciones que sobre ellos se pueden aplicar y por el conjunto de propiedades que dichas operaciones poseen y que determinan inequívocamente su comportamiento. Un valor perteneciente al dominio de un tipo de datos se denomina una instancia o ejemplar del tipo de datos. Un tipo abstracto de datos es un tipo de datos que se define mediante una especificación que es independiente de cualquier implementación. Por tanto, la especificación del tipo abstracto de datos es la definición del mismo. La idea principal que está detrás de la palabra "abstracto" es precisamente la separación e independencia de la especificación del tipo abstracto de datos de una implementación suya.

En esta oportunidad, nos corresponde representar los *TDA* en el lenguaje de especificación algebraica *Nereus*.

3.2 TDA DATA

Este tipo de dato abstracto es utilizados para almacenar los datos pertenecientes a cada uno de los nodos del árbol, especificando la formula a la que representa y su ubicación (en coordenadas) en el árbol de refutación generado.

```
CLASS Data
    IMPORTS
        Integer, String, Boolean
    BASIC CONSTRUCTORS
        Crear, setData
    EFFECTIVE
        TYPES Data
         FUNCTIONS
              Crear: → Data
              getCoordX: Data → Integer
              getCoordY: Data → Integer
              getFormula: Data → String
              isLiteral: Data → Boolean
END-CLASS
```

3.3 TDA COORDINATE

El tipo de dato Coordinate, como su nombre lo dice, almacena la ubicación de un nodo en el árbol de refutación mediante puntos, donde la coordenada *X* corresponde a la ubicación sobre el eje X y la coordenada *Y* corresponde al nivel donde se ubicaba el nodo en el árbol.

```
CLASS Coordinate
    IMPORTS Natural
    BASIC CONSTRUCTORS
        inicCoord, setCoord
    EFFECTIVE
        TYPES Coordinate
        FUNCTIONS
            inicCoord: → Coordinate
            setCoord: Coordinate × Natural × Natural → Coordinate
            getCoordX: Coordinate → Natural
            getCoordY: Coordinate → Natural
END-CLASS
```

3.4 TDA TEST

El tipo de dato abstracto Test, representa cada una de las pruebas lógicas (es decir un conjunto de fórmulas lógicas) que debe cumplir el algoritmo a la hora de su verificación formal. Esta estructura, dadas las distintas aserciones del algoritmo, genera y almacena las formulas lógicas que deben ser lógicamente validas, para la verificación del programa. Cabe aclarar que cuando hablamos de aserción, a la hora de la implementación, utilizamos una cola de cadenas de caracteres para representar a la misma.

```
CLASS Test
    IMPORTS
          Formula, Algorithm, Boolean, Natural
    BASIC CONSTRUCTORS Crear
    EFFECTIVE
          TYPES Test
    FUNCTIONS t:Test
          Crear: Formula× Formula× Formula× Formula× Formula× Algorithm → Test
          getTest: Test × Natural → Formula
                Pre: not finPruebas(t)
          finPruebas: Test → Boolean
END-CLASS
```

3.5 TDA WEAKESTPRECONDITION

Esta estructura hace referencia al transformador de predicados, o como normalmente es conocido, “pre-condición más débil”. Su constructor requiere de una aserción y una porción de programa para generar la precondición más débil.

```
CLASS WeakestPreCondition
    IMPORTS Asertion
```

```
    BASIC CONSTRUCTORS Crear
    EFFECTIVE
           TYPES WeakestPreCondition
    FUNCTIONS
        Crear: Asertion × Algoritmo → WeakestPreCondition
        getPreCondiciónMasDébil: WeakestPreCondition → Asertion
END-CLASS
```

3.6 TDA RefutationTree

RefutationTree representa, valga la redundancia, a un árbol de refutación. Este posee la función getTree es la generadora del árbol en cuestión. Para ello, esta estructura interactúa con los TDA más relevantes de la herramienta: Tree y Formula.

```
CLASS RefutationTree
    IMPORTS
        TernaryTree, Formula
    BASIC CONSTRUCTORS inicRefutationTree
    EFFECTIVE
        TYPES RefutationTree
        FUNCTIONS
            inicRefutationTree: → RefutationTree
            getTree: RefutationTree × Formula → TernaryTree
 END-CLASS
```

3.7 TDA Formula

Esta estructura abstracta representa fórmulas del cálculo de predicados. El método *getPrimerOperando* y *getSegundoOperando* que nos permiten poder obtener las conclusiones de la formula lógica representada.

```
CLASS Formula
    IMPORTS
          Natural, String, Queue [String]
    BASIC CONSTRUCTORS Crear
    EFFECTIVE
        TYPES Formula
        FUNCTIONS
            Crear: Queue [String] → Formula
            getTipo: Formula → Natural
            getPrimerOperando: Formula → Queue[String]
            getSegundoOperando: Formula → Queue[String]
END-CLASS
```

3.8 TDA TernaryTree

El árbol ternario es una de las estructuras más importantes de la herramienta, ya que a partir de la misma se construye el árbol de refutación con el que el usuario va a interactuar.

```
CLASS TernaryTree [elem]
    IMPORTS
    BASIC CONSTRUCTORS
        Crear, Agregar
    EFFECTIVE
        TYPES TernaryTree
        FUNCTIONS
             Crear: → TernaryTree
             Agregar:elem × TernaryTree ×TernaryTree ×TernaryTree→ TernaryTree
END_CLASS
```

Cabe aclarar que un árbol ternario podría tener muchas otras funcionalidades, sin embargo, en nuestro caso las funciones requeridas tienen que ver con una cuestión de implementación, razón por la cual no figuran en esta especificación.

4 ANALIZADOR SINTÁCTICO-LÉXICO

4.1 MOTIVACIÓN

El principal problema de esta aplicación es interpretar las fórmulas introducidas por el usuario, es decir, dada la formalización de un programa, ¿cómo puede la computadora verificarlo? La solución se basó en una analizador sintáctico-léxico para la interpretación y manipulación de las fórmulas introducidas.

4.2 ANALIZADOR LÉXICO

El analizador léxico, *lexer* o *scanner* es el encargado de buscar los componentes léxicos o *tokens* según unas reglas o patrones previamente definidos, generalmente utilizando *expresiones regulares*. Éste divide una secuencia de caracteres en palabras con significado propio y después lo convierte en una secuencia de terminales desde el punto de vista del analizador sintáctico, ya que este consume los tokens que el lexer le provee conforme los vaya necesitando para avanzar en la gramática.

Un *token* es un símbolo terminal de una gramática, una unidad mínima de información. Estos pueden contener información adicional que son de utilidad para el analizador sintáctico y semántico. Un *lexema* es un conjunto de caracteres que concuerdan con el patrón de un token.

4.3 ANALIZADOR SINTÁCTICO

La función del analizador sintáctico, *grammar* o *parser* es comprobar que la secuencia de componentes léxicos o una cadena de *tokens* cumplen las reglas de una gramática dada y genera el árbol sintáctico que lo reconoce.

Si el analizador tuviera que procesar sólo entradas correctas, su diseño e implantación se simplificaría, pero el usuario puede ingresar una cadena mal formada según nuestra gramática, por lo que el analizador debe detectar ese error y notificarlo, para que el usuario puede reingresar correctamente la cadena de texto.

Una gramática se define como $G = \langle N, T, P, S \rangle$, donde N, T, P y S representan a los *símbolos no terminales*, *símbolos terminales*, *reglas de producción* y *axioma inicial* respectivamente. Esto puede realizarse por medio de la notación BNF. Cabe destacar que los lenguajes habitualmente reconocidos por los analizadores sintácticos son los lenguajes libres de contexto. Existe una demostración formal que establece que los lenguajes libres de contexto son aquellos reconocibles por un autómata de pila, de modo que todo analizador sintáctico que reconozca un lenguaje libre de contexto es equivalente en capacidad computacional a un autómata de pila.

4.4 HERRAMIENTAS UTILIZADAS

Codificar los analizadores además de ser muy complejo es tedioso, por lo cual optamos por utilizar un generador para el analizador léxico y sintáctico. En el caso del parser utilizamos *Bison* y para el scanner *Flex*.

4.4.1 BISON

GNU Bison (versión libre de Yacc) es un programa generador de analizadores sintácticos que convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto, en un programa en C++ que realiza análisis sintáctico.

La sintaxis es una adaptación de la notación BNF sin corchetes. Es posible añadir porciones de código escritas en C++ para aportar cierto dinamismo al programa. La sintaxis básica es la siguiente:

```
%{
Declaraciones de c++.
%}

Declaraciones de Bison.
%token terminal.

%%
Reglas gramaticales.

NoTerminal:
        NoTerminal terminal     { acción semántica. }
      | terminal                { acción semántica. }
%%
Código c++ adicional.
```

4.4.2 FLEX

Flex (alternativa de Lex) es un generador de analizadores sintácticos escrito en C++ que utiliza un autómata finito determinístico representado como una matriz de estados para separar en tokens la entrada. Este proceso toma, en el peor de los casos n pasos, siendo n la longitud de la cadena de texto de entrada, es decir $O(n)$.

Su sintaxis se basa en expresiones regulares permitiendo al igual que Bison agregar porciones de código escritas en C++. La estructura básica es la siguiente:

```
%{
código C++.
%}

Declaraciones de flex.

%%
Expresiones regulares.

regEx  { acción semántica. }
```

```
%%
Código C++ adicional.
```

4.5 ESPECIFICACIÓN DEL ANALIZADOR SINTÁCTICO

En la implementación del parser, se separó en diferentes símbolos no terminales el bloque if-else para manipular más fácilmente el árbol del parsing en la recursión y detectar errores para informar al usuario de la malformación de las cadenas ingresadas. Si bien no era necesario separar los bloques previamente dichos en la notación BNF, en la implementación resultó más sencillo hacerlo de esta manera.

```
input:
        assignations block_statement
    |   block_statement

block_statement:
        if_statement
    |   while_statement
    |   formula

else_statement:
        /*En blanco.*/
    |   ELSE LEFT_BRACE assignations RIGHT_BRACE

if_statement:
        IF LEFT_PAR formula RIGHT_PAR LEFT_BRACE assignations RIGHT_BRACE
else_statement

while_statement:
        WHILE LEFT_PAR formula RIGHT_PAR LEFT_BRACE RIGHT_BRACE


assignations:
        VAR EQUAL arith_expr END_OPERATION
    |   assignations VAR EQUAL arith_expr END_OPERATION

formula:
        arith_expr
    |   CUANTIFIER formula
    |   LEFT_PAR formula RIGHT_PAR
    |   NOT formula
    |   DIST formula RIGHT_PAR
    |   formula LOGIC_OPERATOR formula

arith_expr:
        VAR
    |   LEFT_PAR arith_expr RIGHT_PAR
    |   arith_expr ARITH_OPERATOR arith_expr
```

4.6 ESPECIFICACIÓN DEL ANALIZADOR LÉXICO

```
"if" {
       return IF;
}
"else" {
       return ELSE;
}
"while" {
       return WHILE;
}
[a-zA-Z0-9]+  {
       return VAR;
}
[ \t\r\n\s] { /*NADA.*/ }
"=" {
       return EQUAL;
}
";" {
       return END_OPERATION;
}
"{" {
       return LEFT_BRACE;
}
"}" {
       return RIGHT_BRACE;
}
"&&" {
       return LOGIC_OPERATOR;
}

"||" {
       return LOGIC_OPERATOR;
}
"->"    {
       return LOGIC_OPERATOR;
}
"<->" {
       return LOGIC_OPERATOR;
}
"!" {
       return NOT;
}
"!(" {
       return DIST;
}
[ \0\0] {
       return END;
}
"("     {
       return LEFT_PAR;
}
")"     {
```

```
        return RIGHT_PAR;
}
[\^\*+-><]  {
        return ARITH_OPERATOR;
}
">=" {
        return ARITH_OPERATOR;
}
"<=" {
        return ARITH_OPERATOR;
}
"!=" {
       return ARITH_OPERATOR;
}
"@"  {
       return CUANTIFIER;
}
"#" {
       return CUANTIFIER;
}
```

5 DETALLES DE IMPLEMENTACIÓN

En esta sección se discute el diseño de las clases esenciales del software y el porqué de las herramientas elegidas.

5.1 DISCUSIÓN DEL DISEÑO DE CLASES

5.1.1 CLASE TEST

La clase *Test* es la encargada de generar cada una de las pruebas que se deben cumplir en la verificación formal de un algoritmo. A partir de un valor dado distinguirá entre la verificación de una sentencia *while*, *if*, o *if-else* generando sus respectivas pruebas.

El constructor de la clase recibe como parámetro las aserciones del programa y la estructura *parsedData* que almacena el algoritmo en cuestión. A partir de estas estructuras y algunos de sus métodos privados generara y almacenara cada una de las formulas lógicas que se deben satisfacer. La formación de las pruebas se realiza de forma dinámica, ya que dependiendo de las sentencias que contenga el programa, será el número de fórmulas generadas. Para ello fue necesaria una estructura auxiliar llamada *_PROOF* que es una lista de punteros donde cada nodo contiene los siguientes datos:

- *QQueue<string> test:* una cola de string, que almacena la prueba lógica.
- *bool view*: valor booleano que determina si la prueba fue verificada o aún no.
- *unsigned int num*: número natural que determina el número de prueba al que corresponde.
- *_proof* nextTest:* puntero a la siguiente prueba

Los métodos privados más importantes implementados son los encargados de la formación de las pruebas. Cada una de las sentencias tiene diferentes pruebas por en ende fue necesario tener un método por cada una de ellas. A continuación describiremos las formulas lógicas generadas por cada uno de esos métodos, dependiendo de las sentencias existentes:

- Sentencia "While":
  - _Test1_W: $\{\boldsymbol{P}\}\ \boldsymbol{A}\ \{\boldsymbol{I}\}$
  - _Test2_W: $\{\boldsymbol{I}\ \wedge\ \boldsymbol{b}\}\ \boldsymbol{A}\ \{\boldsymbol{I}\}$
  - _Test3_W: $\boldsymbol{I}\ \wedge\ \neg\boldsymbol{B}\ \Rightarrow\ \boldsymbol{Q}$
  - _Test4_W: $\boldsymbol{I}\ \wedge \boldsymbol{B}\ \Rightarrow\ \boldsymbol{C} \geq \boldsymbol{0}$
  - _Test5_W: $\{\boldsymbol{I} \wedge \boldsymbol{B} \wedge \boldsymbol{C} = \boldsymbol{T}\}\boldsymbol{A}\ \{\boldsymbol{C} < \boldsymbol{T}\}$

- Sentencia "If-Then":
  - _Test1_IF_T: $\{\boldsymbol{P} \wedge \boldsymbol{B}\}\ \boldsymbol{C1}\ \{\boldsymbol{Q}\}$
  - _Test2_IF_T: $\{\boldsymbol{P} \wedge \neg\boldsymbol{B}\}\ \Rightarrow\ \{\boldsymbol{Q}\}$

- Sentencia "If-Else":
  - _Test1_IF_E: $\{\boldsymbol{P} \wedge \boldsymbol{B}\}\ \boldsymbol{C1}\ \{\boldsymbol{Q}\}$
  - _Test2_IF_E: $\{\boldsymbol{P} \wedge \neg\boldsymbol{B}\}\ \boldsymbol{C2}\ \{\boldsymbol{Q}\}$

Esta clase interactúa con la clase *WeakestPreCondition*, ya que de ser necesario, la construcción de la prueba requerirá el cálculo de la precondición más débil.

5.1.2 CLASE REFUTATIONTREE

La implementación de esta clase tiene como objetivo construir un árbol de refutación. El constructor de la clase recibe como parámetro una cola de *tokens* con la fórmula de entrada y un objeto *Tree* vacío que será devuelto con el árbol de refutación completo.

Esta clase provee un método que es el encargado de generar el árbol de refutación, por lo que interactúa con la clase *Formula*, que es la encargada de distinguir y aplicar las reglas de formación del árbol. Esta a su vez también genera las conclusiones de cada formula y son agregadas al árbol de refutación. El método en cuestión se llama *getTree*, y el pseudocódigo del mismo es el siguiente:

```
getTree(Arbol & a, Queue<Formula> formulas) {
       Formula f = NULL;
       Formula conclusion = NULL;
    while ( ! formulas.empty() ) {
        Formula f = formulas.dequeue();
        if ( f.tipoA() || f.tipoB()  )  //sin incluir cuantificadores
            generarConclusion(conclusion, formula,tipo);
         else //Fórmula que involucra cuantificadores.
                if ( f.cuantificadorExistencial() )
                    generarConclusionConExistencial(conclusion, formula);
                else
                    for ( constante in constantes )
                        generarConclusionConstantes(conclusion, constante);
              /**
               * Se agrega al árbol de refutación teniendo en cuenta el tipo
               * de regla que se aplicó.
               */
              a.agregarConclusion(conclusion);
              formulas.enqueue(conclusion);
       }
}
```

5.1.3 CLASE TERNARYTREE

La clase *TernaryTree* implementa la estructura de un árbol ternario. Cada nodo de éste árbol tiene una cadena de caracteres que representa una fórmula lógica, un objeto *Coordinate* que contiene la ubicación en coordenadas cartesianas del nodo en la representación gráfica, un valor que determina si la fórmula es un literal o no y un puntero al nodo padre.

El hecho de que cada nodo posea un solo puntero al padre se debe a que la inserción de los nodos en el árbol comienza desde las raíces, y dependiendo del tipo de fórmula se le asignaran a los nodos el mismo padre (regla tipo B) o se enlazaran para posteriormente agregarlos al árbol (reglas del tipo A). La inserción de los nodos, por un lado, es de a pares, es decir, una premisa posee dos conclusiones, estas dos fórmulas son las que se agregan al árbol. En el caso de la existencia de cuantificadores, la inserción solo involucrara un solo nodo (dependiendo del cuantificador, será la inserción del mismo).

Como se ha visto en la teoría, no todas las formulas se agregan de la misma manera a la hora de construir un árbol de refutación. Los métodos utilizados para la formación del árbol, son *setTree* y *addFormulas*. El primero de ellos sólo inicializa el árbol, es decir, inserta la raíz del mismo (la fórmula a partir de la cual vamos a desarrollar el árbol de refutación) mientras que el segundo es el encargado de

agregar los nodos al árbol. Éste recibe como parámetros el tipo de la formula, las 2 formulas a agregar (en el caso de los cuantificadores, solo se agrega una, el otro parámetro es cargado como *NULL*), la premisa de esas conclusiones y un valor que determina si en la premisa existe un cuantificador existencial. De esta manera, en la inserción se distinguen tres casos: las formulas del tipo A, las del tipo B y las que involucran a cuantificadores, tanto existenciales como universales. El pseudocódigo del algoritmo base de los mismos es el siguiente:

```
addFormulas(Formula conclusion) {
    List<Hoja> hojasConPremisa = this.getHojasConPremisa(conclusion);
    if ( hojasConPremisa == NULL ) {
    /**
     * Si no existe la premisa en las hojas del árbol entonces inserto las
     * conclusiones en todas las hojas que contengan la premisa en su misma
     * rama.
     */
        List<Rama> ramasConPremisa = this.getRamasConPremisa(conclusion);
        for ( rama in ramasConPremisa )
            rama.insertarConclusion(conclusion);
    }
    else
        for ( hoja in hojasConPremisa )
            hoja.insertarConclusion(conclusion);
}
```

Esta metodología es la misma en las tres formas de inserción, pero cada uno de los métodos realiza su acción respetando las reglas mencionadas anteriormente. Es decir, en las reglas de tipo A (incluyendo fórmulas que involucran cuantificadores, pero en este caso, solo se agrega un solo nodo) los nodos se agregan en la misma rama de la premisa; en el caso de las reglas del B, los nodos se insertan expandiendo la rama correspondiente a su premisa. Además, en la implementación de las reglas que involucran cuantificadores, existe un método adicional al final del algoritmo, que se encarga de administrar y actualizar las constantes utilizadas anteriormente.

5.1.4 CLASE FORMULA

Esta clase recibe una cola de *tokens* y aplica las reglas básicas de la lógica de predicados, almacenando también el tipo de fórmula para que resulte más sencillo la formación del árbol de refutación.

Si bien es la clase más larga en cuanto a código, es la más fácil de entender debido a que es la encargada de notificar qué reglas se deben aplicar en la fórmula y de aplicar distribuciones de cuantificadores y los demás operadores lógicos con el fin de simplificar la fórmula en cada nodo del árbol de refutación.

5.1.5 CLASE PARSER

Es la encargada de interactuar con el parser implementado. Recibe un string por cada fórmula que se quiera transformar a un conjunto de tokens y en cada una de ellas se debe agregar al final un string, '\0\0' esto sirve para que el parser identifique el token de fin de fórmula, que es la definición explícita de un string vacío. Esto se utiliza ya que tuvimos que modificar el código fuente de Bison para que lea

directamente las entradas como cadenas de texto almacenadas en un buffer y no desde un archivo, ya que resultó más cómodo. Una vez que el lexer detecta el token *FIN,* finaliza el algoritmo.

El parser inserta las fórmulas en un *árbol de parsing*, en donde es más sencillo manipular los tokens. Éste retorna en estra estructura el algoritmo formalizado, pero para el resto de las fórmulas lógicas se utiliza una cola con sus respectivos tokens.

5.2 Qt

Debido a la complejidad que conlleva el desarrollo de esta aplicación, optamos por utilizar un framework, sobre todo para solucionar la interfaz gráfica y la comunicación entre la lógica del software y las entradas del usuario.

Qt provee varias funcionalidades que fueron de gran utilidad para nuestro trabajo, desde estructuras completamente implementadas hasta integración de tecnologías ajenas al lenguaje tales como *Canvas*. Las estructuras más utilizadas fueron *QQueue* (cola), *QList* (lista), *QString* (string), *QMap* (map) entre otras. El hecho de que se utilicen las estructuras que incluye Qt y no las de la librería estándar es que las últimas son muy pobres en cuanto a sus métodos, por el contrario, Qt ofrece para cada una de ellas una amplia funcionalidad que agilizó el desarrollo.

5.3 Qml

Inicialmente, optamos por realizar la interfaz gráfica con *HTML5*, ya que es una tecnología conocida para nosotros y muy agradable para utilizar. Si bien Qt permite la integración de *HTML5*, esto conllevaba la integración de un módulo nativo de Google Chrome, Opera y Safari llamado *WebKit*, el cual permite renderizar regiones del programa con contenido web dinámico. Ciertas cuestiones de implementación hacen que la integración de este módulo sea muy difícil por lo cual no era una solución viable. Una segunda opción surgió al darnos cuenta que Qml, un metalenguaje basado en *Javascript* para diseñar aplicaciones enfocadas en la interfaz de usuario. Qml tiene muchas similitudes con *HTML5* y *CSS3*, ya que cubre desde efectos gráficos como transiciones e integra etiquetas en formato *JSON* que se declaran con sus respectivos comportamientos y propiedades. La utilización de este lenguaje fue muy sencilla debido a su gran parecido con las tecnologías previamente nombradas y además resultó muy eficiente y le dio un aspecto diferente al software.

Toda la lógica de la interfaz gráfica, es decir, el comportamiento de cada elemento fue codificado en el lenguaje Javascript, y en general utilizamos eventos para permitir la interacción entre el usuario y el sistema.

5.4 Canvas

Para renderizar los árboles de refutación fue necesaria la integración de Canvas, un elemento de HTML5 que permite la generación tanto de gráficos estáticos y animaciones dinámicamente por medio del scripting.

Un ejemplo sencillo es el caso del gráfico de un cuadrado:

```
import QtQuick 2.0

Canvas {
    //Tamaño del elemento Canvas.
    width: 200; height: 200
    // handler to override for drawing
    onPaint: {
        //Obtener el contexto a graficar.
        var ctx = getContext("2d")
        ctx.lineWidth = 4
        ctx.strokeStyle = "blue"
        ctx.fillStyle = "steelblue"
        //Comenzar un nuevo camino para dibujar.
        ctx.beginPath()
        ctx.moveTo(50,50)
        //Linea superior.
        ctx.lineTo(150,50)
        //Linea derecha.
        ctx.lineTo(150,150)
        //Linea inferior.
        ctx.lineTo(50,150)
        //Finalizar el camino.
        ctx.closePath()
        //Pintar el cuadrado.
        ctx.fill()
        //Dibujar las lineas.
        ctx.stroke()
    }
}
```

Cabe destacar que en este caso, no es necesario dibujar un cuadrado ya que Qml provee el elemento Rectangle.

6 CONCLUSIÓN

Con los conocimientos que nos brindaron las materias Ciencias de la Computación I y Análisis y Diseño de Algoritmos I pudimos llevar a cabo gran parte del proyecto. Su finalidad es pura y exclusivamente educativa para los alumnos que la deseen utilizar permitiendo correcciones de sus algoritmos a medida que se va efectuando la verificación, y de esta manera poder adquirir los conceptos involucrados en la verificación formal de programas.

Si bien la verificación de sentencias anidadas o consecutivas no está automatizada en esta versión del programa se puede extender en un futuro para añadir una posible mejora para próximas implementaciones, es decir, la verificación se puede realizar de la misma manera pero de forma manual, involucrando un conocimiento más profundo sobre el tema por parte del usuario.

Por otro lado, es importante destacar que la realización de este trabajo involucró el aprendizaje de nuevos conceptos no vistos en las materias previamente nombradas tales como analizadores sintácticos y léxicos, programación de la interfaz de usuario con nuevas tecnologías y lenguajes y el uso de frameworks, aunque lo más importante fue la realización de un primer proyecto real en la carrera y la posibilidad de trabajar en equipo para resolver un problema dado, permitiéndonos utilizar las herramientas que hayamos creído necesarias.

7 BIBLIOGRAFÍA

- Kaldewaij, A.: *Programming: The Derivation of Algorithms*. Prentice-Hall International Series in Computer Science (1990).
- Levine, John: *Flex & Bison: Text Processing Tools*. O'reilly (2009)