Tabla de Contenido

0. INTRODUCCIÓN **1**

1. CONTEXTO DEL PROBADOR DE TEOREMAS (ManTa) **5**

1.1. **DESCRIPCIÓN DE MANTA** 5

1.1.1. **Sintaxis y Semántica del Lenguaje de Especificación de ManTa** 6

1.1.2. **Proposición de Expresiones para Rescribir** 8

1.1.3. **Proposición de Conjeturas** 9

1.2. **ANÁLISIS DEL PROBADOR ANTERIOR DE MANTA** 10

1.2.1. **Motor de Reescritura** 12

1.2.1.1. **Reescritura con Axiomas** 12

1.2.1.2. **Manejo del Elemento *ERROR*** 14

1.2.2. **Métodos de Prueba** 15

1.2.2.1. **Prueba por Inducción Estructural** 16

1.2.2.2. **Prueba Semántica y el Predicado EQUAL** 17

1.2.3. **Aplicación de Métodos de Prueba** 19

2. APORTES AL PROBADOR DE TEOREMAS DE MANTA **20**

2.1. **REESCRITURA** 20

2.1.1. **Manejo del Elemento *ERROR*** 20

2.1.2. **Reescritura con Lemas** 21

2.1.3. **Reescritura con Hipótesis** 25

2.2. **CONJETURAS CON HIPÓTESIS** 27

2.3. **MÉTODOS DE PRUEBA** 28

2.3.1. **Prueba por Casos** 28

2.3.2. **Prueba por Shannon** 29

2.3.3. **Generalización** 30

2.4. **APLICACIÓN DE LOS MÉTODOS DE PRUEBA** 32
2.5. **APOYO A LA VERIFICACIÓN DEL PREDICADO EQUAL** 34

3. CONCLUSIONES **36**
3.1. **PROBLEMAS CONOCIDOS DEL PROBADOR DE TEOREMAS** 36
3.2. **PROPUESTAS PARA IMPLEMENTAR EN EL PROBADOR DE TEOREMAS DE MANTA** 37

4. BIBLIOGRAFÍA **42**

Anexo A. MANUAL DE USO DEL PROBADOR DE TEOREMAS

Anexo B. MANUAL DE MANTENIENTO
ESTRUCTURAS DE DATOS
B−1
ESTRUCTURA DE ARCHIVOS DE *kernel\d_theo*
B−2
RELACIÓN DE FUNCIONES DEL MÓDULO DE TEOREMAS
B−3

Anexo C. EJEMPLOS DE DEMOSTRACIONES HECHAS POR EL PROBADOR DE TEOREMAS DE MANTA

Anexo D. DEMOSTRACIONES DE TEOREMAS USADOS EN ESTE DOCUMENTO

ÍNDICE ANALÍTICO

0 INTRODUCCIÓN

"Se hace la ciencia con hechos,
como una casa con piedras;
pero una acumulación de hechos
no es una ciencia, del mismo
modo que un montón de piedras
no es una casa."[1]
Henri Poincaré

Un gran aporte de las matemáticas es, sin lugar a dudas, haber abierto un espacio de expresión, que permite crear modelos científicos, con los cuales ha sido posible pensar la Naturaleza mediante razonamientos deductivos dentro de las teorías elaboradas. Se ha visto, por tanto, en la historia de la ciencia una coevolución entre las teorías científicas y las matemáticas.

No es, en modo alguno, una simple coincidencia la colaboración mutua entre la ciencia y las matemáticas, sino una manifestación de la sustentación recíproca existente. Si bien es cierto que los postulados de la ciencia habrían carecido de herramientas sin el pensamiento matemático, es igualmente válido afirmar que son las aplicaciones prácticas y la posibilidad de hablar de la realidad, tal como se propone la ciencia, las que hacen de las deducciones matemáticas mucho más que una gigantesca tautología.

Parecería, entonces, que las matemáticas no pueden alimentarse por sí mismas; sin embargo, son varios los avances que se conocen de ellas, en los que sólo ha tenido participación esta disciplina. ¿Tienen algo que decir las ideas desarrolladas internamente en la teoría matemática? la respuesta tiene que ser sí, pues de no ser así no tendría sentido poner la ciencia en lenguaje matemático.

Lo anterior es contradictorio tan sólo en apariencia, pues al utilizar el razonamiento matemático no ponemos en duda su potencialidad; pero no se trata de una ligereza en su uso, sino que al dar un significado real a las premisas que nutren las matemáticas, definimos las reglas con las que comunicamos el mundo abstracto y el tangible. Se trata, por tanto, de un puente de comunicación entre el modelo y el mundo, que se transita

[1] [Poi1907].

cuando se quieren hacer afirmaciones concretas; pero el pensamiento fuera del mundo concreto no se deshace, puesto que está ligado a la realidad desde su inicio.

Poca sería la utilidad del *software* si no estuviera en él plasmada la abstracción de situaciones concretas; realidades que se han podido plantear mediante modelos matemáticos, estableciendo, así, las reglas del desarrollo de la situación real. Es la elaboración de esta abstracción lo que se hace en la fase inicial del diseño del *software*, con lo cual los desarrollos matemáticos posteriores cobran sentido para la solución de un problema.

Probador de Teoremas

En la tesis que se explicará a lo largo de este documento se trabajó en el desarrollo del módulo de *prueba de teoremas* de ManTa (Manejador de Tipos Abstractos). Esta es una herramienta para apoyar la especificación de *Tipos Abstractos de Datos −TAD−*, que constituye una teoría muy utilizada para la definición formal de problemas abordables en informática.

La motivación para indagar sobre la prueba de teoremas se basa en la necesidad de comparar el comportamiento del modelo, resultado de la abstracción, y la realidad modelada en los aspectos que el diseñador considera relevantes.

Nuevamente, resalto ahora, debe darse su justo papel al desarrollo matemático que se hace en la prueba de teoremas; se trata del aprovechamiento de la exactitud con que fue planteada la realidad, para deducir propiedades de ésta mediante un método matemático, con el beneficio de poder hacerlo ayudado por un computador.

El trabajo de tesis que se expone en este documento consiste en revisar el *probador de teoremas* de ManTa e implementar nueva funcionalidad para éste. El desarrollo que se lleva a cabo se hace sobre plataforma DOS, en lenguaje C.

En el primer capítulo se hace un resumen del proyecto ManTa, enfocado en el módulo de *prueba de teoremas* y puntualizando los aspectos de los desarrollos anteriores sobre los que

se trabaja en esta tesis, ya que daban espacio para ser ampliados o tenían errores que intentan ser corregidos en el presente trabajo. El segundo capítulo corresponde al estudio de la teoría en la que se basan las mejoras implementadas en este trabajo, para presentar, en las conclusiones (*Capítulo 3*), sugerencias de lo que podría trabajarse en un futuro sobre este tema.

Notación Usada en este Documento

Dentro de este documento se usará la notación adoptada en ManTa, para ilustrar su funcionamiento; ésta es una notación funcional prefija, en la que todas las variables están universalmente cuantificadas (una mayor explicación se encuentra en la sección *1.1. Descripción de ManTa*).

ejemplo:

> `MULT(X, Y) = MULT(Y, X)` debe entenderse: *para todo `X`, `Y` perteneciente a los Naturales, `X` multiplicado por `Y` es igual a `Y` multiplicado por `X`.*

Se utilizará también una notación menos estricta para dar una explicación más asequible, en la que se incluyen los símbolos comunes para las funciones conocidas (v. gr. suma, multiplicación, cuantificación, etc.) Esta notación es tomada de [Gri1994], en la que se maneja la siguiente estructura:

($cuantificador_{variable\ cuantificada}$ | *rango de operación de la variable cuantificada* : *operando*)[2]

De esta forma se pueden escribir expresiones como:

$(\Sigma_i \mid 1 \leq i \leq 10 : i)$ que equivale a $\Sigma i = 1 + 2 + ... + 10$ o

$(\forall_{x, y} \mid x, y \in \mathrm{NAT} : x * y = y * x)$, que es equivalente al ejemplo dado para la notación de ManTa.

Finalmente, para expresiones en las que no se cuantifican las variables, se usa una notación informal en la que se incluyen los símbolos comunes y los rangos se dan con los operandos iniciales y finales, separados por puntos suspensivos. Se usan, también, para expresar sustituciones, subíndices y superíndices, con los que se indica la variable y el término que participan en la sustitución.

[2] El cuantificador es la función con que se opera.

Como ejemplos de esta notación podemos ver expresiones como las siguientes:

$A_1 \cup ... \cup A_n$, que significa la unión de los conjuntos A_1 a A_n y

E, que se debe entender como la expresión *E* con todas las ocurrencias de la variable *z* sustituidas por *true* (se supone que *z* es de tipo *booleano*).

1 CONTEXTO DEL PROBADOR DE TEOREMAS (ManTa)

ManTa es un proyecto del Grupo de Construcción de Software de la Universidad de los Andes, que se está llevando a cabo desde 1985 y ha contado con la colaboración de profesores y estudiantes de pregrado y magíster (una síntesis del desarrollo de ManTa se encuentra en [Tam1998]).

1.1. DESCRIPCIÓN DE MANTA

Una fase de gran importancia en el desarrollo de software es la de diseño, en la que se define un modelo de la realidad del problema a abordar. De la buena definición de este modelo depende, en gran medida, el desarrollo de las fases posteriores; por esta razón, es necesario contar con herramientas para verificar el modelo del mundo.

ManTa es una aplicación para la manipulación formal de *Tipos Abstractos de Datos*[1] — *TAD*[2] —, que ofrece una serie de herramientas para el modelaje con *TADs ecuacionales*[3], con las cuales se pueden especificar y verificar propiedades de estructuras de datos, que son luego implementadas de forma automática y certificada.

En este capítulo se pretende describir la funcionalidad de ManTa, desarrollada en trabajos anteriores a este, y definir algunos conceptos que se utilizarán a lo largo del documento; esto con el objeto de dar una idea sobre el contexto del probador de teoremas y familiarizarnos con la terminología. Las explicaciones que se dan a continuación no abordan a fondo los temas tratados, por lo que se encontrarán referencias a diferentes títulos incluidos en la bibliografía.

[1] Un tipo abstracto de datos es una estructura algebraica con propiedades particulares. Quien conoce la programación *Orientada a Objetos* puede pensar en un *TAD* como en una *clase*.
[2] Se usará, también, la palabra *tipo* para referirse a *tipos abstractos de datos*.
[3] Se entiende por *TAD ecuacional* una estructura algebraica de datos con sus propiedades descritas mediante axiomas ecuacionales ([Car1995]).

1.1.1. Sintaxis y Semántica del Lenguaje de Especificación de ManTa

En los inicios de ManTa se estableció un lenguaje para la especificación de *TADs*, la proposición de expresiones para rescribir y el planteamiento de conjeturas. A continuación se hace una breve exposición de la sintaxis y de la semántica de ManTa. Para una documentación completa pueden consultarse [Tam1998] y [ManDoc], donde se encuentra la descripción formal de ManTa (*Sintaxis Concreta*, *Árbol de Sintaxis Abstracta*, *Semántica Estática*, *Semántica Operacional*, *Semántica Denotacional* y *Semántica Traslacional y Esquema de Traducción*).

Especificación de TADs

Un *TAD* se compone de un conjunto de elementos, llamado el *tipo de interés* (*tdi*), y unas *funciones* u *operaciones*, que actúan sobre el *tdi* (ver el *Diagrama 1.1*). La definición del *tdi* y las *funciones*, en ManTa, se lleva a cabo de forma funcional recurrente o inductiva y las funciones se dotan de semántica mediante un conjunto de axiomas.

En Manta un *TAD* se define por medio de su *signatura* y sus *axiomas*. La *signatura* es la especificación de la sintaxis del tipo, en la que se da nombre y aridad a las funciones (vale la pena mencionar aquí que en ManTa el *tdi* de un *TAD*, al igual que las operaciones, es definido por medio de funciones). Los *axiomas* son un conjunto de igualdades, con el que se da semántica a las operaciones del *TAD*.

Para un **`TAD Y`** (con *parámetros*[4] $\mathbf{X_1, \ldots, X_n}$), podemos clasificar las funciones que lo especifican, de la siguiente manera ([Muñ1992], *sección 1.1.1*; [Car1993], *sección 6.4.3.*):

iniciales: Definen los elementos básicos del *TAD*. Una función inicial puede tener aridad cero, en cuyo caso se trata de una constante, o tomar como parámetros variables de los tipos $\mathbf{X_i}$.

(signatura. $\mathbf{i : X_{i1} * \ldots * X_{im} \not\in Y}$, $1 \le i_j \le n$).

[4] Los parámetros son *place holders* de tipos, usados para la creación de tipos genéricos o paramétricos ([ManDoc], documento *User's Manual*). Con la opción de tener tipos parámetros se puede, por ejemplo, especificar el *TAD Pila* sin necesidad de decir si se trata de una pila de papeles, de números, etc.

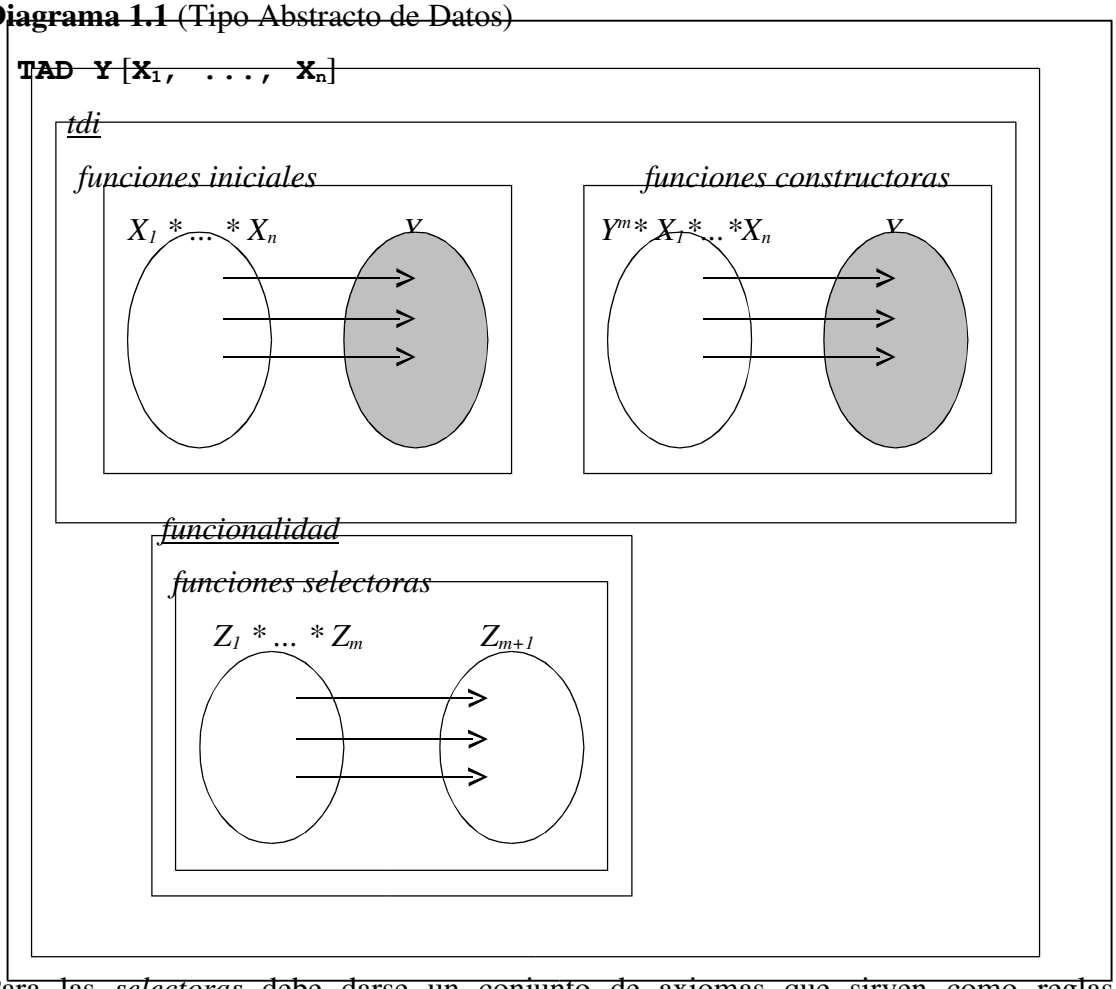
constructoras: Usadas para definir elementos de **Y**, con base en los ya definidos y los elementos de tipo $\mathbf{X_i}$.

(signatura. $\boldsymbol{c}$ **: Y * ... * Y * $\mathbf{X_{i1}}$ * ... * $\mathbf{X_{im}}$ ¢ Y**, $1 \le i_j \le n$).

selectoras: Funciones para manipular elementos de **Y**, junto con los de otros *TADs* ya definidos. Las *selectoras* son la funcionalidad del tipo y permiten relacionarlo con los demás *TADs*.

(signatura. $\boldsymbol{s}$ **: $\mathbf{Z_1}$ * ... * $\mathbf{Z_m}$ ¢ $\mathbf{Z_{m+1}}$**). Los tipos $\mathbf{Z_i}$ se utilizan para indicar que puede ser cualquier tipo del ambiente, que no guarda relación alguna con el ***TAD* Y** ni con los *parámetros* $\mathbf{X_i}$.

Diagrama 1.1 (Tipo Abstracto de Datos)



Para las *selectoras* debe darse un conjunto de axiomas que sirven como reglas de reescritura para los valores del *TAD*. Los *axiomas* determinan la igualdad entre dos expresiones, donde la izquierda debe ser de la forma $f(x_1, ..., x_m)$ (siendo f una selectora del

TAD y x_i una variable o valor del tipo especificado en la *signatura* para el i–ésimo parámetro de *f*) y la expresión derecha debe ser una expresión del tipo de *f* (esta expresión puede incluir la función *if–then–else* o ser el elemento *ERROR*[5]).

ManTa ofrece la opción de crear nuevos *TADs* de forma semi–automática, basándose en los tipos ya definidos. Se cuenta, además, con algunos tipos predefinidos, dentro de los que se encuentra BOOL (tipo primitivo), NAT y algunos *TADs* conocidos. En el *Cuadro 1.1* se presenta la especificación del *TAD Lista*, para ilustrar lo dicho hasta el momento.

Cuadro 1.1 (TAD Lista en sintaxis de ManTa)

```
TAD List [ X ]
INICIALES
nil          :                                  -> List
CONSTRUCTORAS
cons         : X * List                         -> List
SELECTORAS
head         : List(i)                          -> X
tail         : List(i)                          -> List
length       : List(i)                          -> NAT
concat       : List(i) * List                   -> List
memberList   : List(i) * X                      -> BOOL
isEmptyList  : List(i)                          -> BOOL
EQ_List  (c): List(i) * List(i)                 -> BOOL

AXIOMAS
head(nil) = ERROR
head(cons(X1,L2)) = X1
tail(nil) = ERROR
tail(cons(X1,L2)) = L2
length(nil) = ZERO
length(cons(X1,L2)) = SUC(length(L2))
concat(nil,L1) = L1
concat(cons(X1,L2),L3) = cons(X1,concat(L2,L3))
memberList(nil,X1) = FALSE
memberList(cons(X1,L2),X3) = if EQUAL(X1,X3)
                                then TRUE
                                else memberList(L2,X3)
                             fi
isEmptyList(nil) = TRUE
isEmptyList(cons(X1,L2)) = FALSE
EQ_List(nil,nil) = TRUE
EQ_List(nil,cons(X1,L2)) = FALSE
EQ_List(cons(X1,L2),nil) = FALSE
EQ_List(cons(X1,L2),cons(X3,L4)) = AND(EQUAL(X1,X3),EQ_List(L2,L4))
```

1.1.2. Proposición de Expresiones para Rescribir

Las expresiones en ManTa son funcionales (en la *Tabla 1.1* se encuentran las clases de expresiones) y se escriben de acuerdo a la sintaxis dada en la especificación de los *TADs*. Una expresión tiene, entonces, un árbol sintáctico asociado, como se ilustra con el siguiente ejemplo:

IMP

[5] A todos los *TADs* manejados en ManTa se les agrega un elemento *ERROR*, para permitir la especificación de selectoras para los valores en los que no están definidas.

Para la sintaxis se adoptó la notación prefija y se reservaron los paréntesis (’*(*’ y ’*)*’) para agrupar los parámetros de las funciones con aridad mayor a cero; las funciones constantes (que no tienen parámetros), son referidas por el nombre que se les dio. Adicional a los elementos y funciones de los *TADs*, una expresión tiene variables, a las que puede dárseles cualquier nombre que no entre en conflicto con las palabras usadas en los tipos.

Para las expresiones existen, además de las operaciones especificadas, las funciones *if−then−else* y *EQUAL* y el elemento *ERROR* (cuya utilización se describe en la *Tabla 1.1*). Estas operaciones toman diferentes tipos (es decir, son polimorfas) y su comportamiento no es modificable, pues está determinado por el motor de reescritura.

Tabla 1.1 (clases de expresiones en ManTa)

clase	***sintaxis***	***tipo***
Variable	<nombre de la variable>	*Se define de acuerdo al contexto*
Valor Inicial	<nombre de función inicial del TAD Z*>	Z
Valor Construido	<nombre de una función constructora del TAD Z> (<*parámetros*>)	Z
Selectora	<nombre de una selectora del TAD Z> (<*parámetros*>)	X
if−then−else	*if* <expresión de tipo BOOL> *then* <expresión de tipo Z> *else* <expresión de tipo Z> *fi*	Z
EQUAL	*EQUAL*(<expresión de tipo Z>, <expresión de tipo Z>)	BOOL
ERROR	*ERROR*	*Se define de acuerdo al contexto*

* Los tipos Z y X significan cualquier tipo, pero se mantiene coherencia entre la expresión cuando aparece más de una vez.

1.1.3. Proposición de Conjeturas

Un módulo de ManTa, para apoyar la labor de especificación, es el *probador de teoremas*, del cual se hablará a lo largo de este documento. Con esta herramienta se permite al usuario proponer *conjeturas*, que ManTa intenta probar de forma semi−automática.

Las conjeturas con que se trabaja son ecuacionales y tienen la siguiente forma:

$$Expresión_{izquierda} = Expresión_{derecha}$$ [6]

Estas expresiones tienen la estructura mencionada en el numeral anterior y en la conjetura está implícita una cuantificación universal para todas las variables mencionadas dentro de ella; así, si se propone la conjetura *ADD(X, Y) = ADD(Y, X)* se probaría el teorema $(\forall_{X,\,Y} \mid X, Y \in \text{NAT} \mid : X+Y = Y+X)$.

1.2. ANÁLISIS DEL PROBADOR ANTERIOR DE MANTA

"El carácter esencial del razonamiento por recurrencia es que contiene, condensados, por decirlo así, en una fórmula única, una infinidad de silogismos"[7]

Henri Poincaré

ManTa cuenta, desde 1987, con un módulo de *prueba de teoremas*[8], cuya finalidad dentro de la aplicación es ayudar, a quien define un *TAD*, a probar que la representación que escogió para un objeto de la realidad modelada, conserva, efectivamente, las propiedades[9] que se quieren tener dentro del modelo.

Por ejemplo, si se están modelando las calificaciones de un estudiante en un curso, se pueden haber escogido los números naturales, para representar cada calificación, y una lista de números naturales, para representar las calificaciones del estudiante en el curso.

calificación es de tipo NAT

calificaciones del curso es de tipo List[NAT]

[6] En el capítulo 2 (*Aportes al Probador de Teoremas de ManTa*) se tratará el tema de conjeturas con hipótesis, que no tienen esta estructura. Sin embargo, la forma de escribirlas es usando también igualdad entre dos expresiones, donde una de ellas es una implicación; la interpretación de que se trata de una conjetura de la forma $h \mid\!\!- E_{izq} = E_{der}$ la hace ManTa internamente.

[7] [Poi1907], p. 20

[8] En este documento se hablará de conjeturas, teoremas y lemas. Conjetura es una igualdad que se cree es verdadera; teorema es una igualdad que ya se ha probado que es verdadera; y lema es una igualdad, ya probada, que se usa como regla de reescritura, para probar que una conjetura es verdadera.

[9] Estas propiedades pueden ser del objeto como tal o de su relación con otros.

Suponiendo que todas las calificaciones tienen un mismo peso, se puede querer mantener la propiedad de que intercambiar el orden de dos calificaciones no afecta el resultado de sumarlas. Con este fin se propone la conjetura [10]

$$(\forall_{x,y,L} \mid x, y \in \text{NAT} \wedge L \in \text{List[NAT]} : sumaLista(x4y4L) = sumaLista(y4x4L)),$$

$$donde\ \ sumaLista(L) = (\Sigma_i \mid 0 \leq i < tamaño(L) : L[i])$$

En el ejemplo 1 (*Anexo C*) se encuentra el desarrollo en ManTa de este caso.

La primera tarea que se llevó a cabo en este trabajo de tesis fue la de evaluar el *probador de teoremas* ya implementado en ManTa, para empezar por solucionar los errores de implementación que se encontraran y tener, así, claridad sobre el punto de partida en la fase de desarrollo de la nueva funcionalidad.

Las pruebas realizadas fueron conjeturas clasificadas de acuerdo con un conjunto de características (esta clasificación se encuentra descrita en la *Tabla 1.2* junto con algunos ejemplos). Se tomaron conjeturas con estas características con la finalidad de obtener la información necesaria (ubicación dentro de la estructura del código fuente) de los errores que se hallaran.

Tabla 1.2 (Clasificación de las pruebas hechas al *probador de teoremas*)

Conjetura	Verdadera ?	Tipos Involucrados	Número de Funciones Involucradas	Funciones Recurrentes	Funciones con *if−then−else* en el cuerpo	Resultado
AND(X, Y) = AND(Y, X)	*verdadera*	*BOOL*	*1*	*No*	*No*	*Se cae el programa*
ADD(X, Y) = ADD(Y, X)	*verdadera*	*NAT*	*1*	*Sí*	*No*	*Probada (bien)*
concat(x, y) = concat(y, x)	*falsa*	*List*	*1*	*Sí*	*No*	*Probada (error)*

En los subcapítulos sucesivos se expone el funcionamiento del *probador de teoremas* con el que se empezó este trabajo. Sobre estos temas se pueden consultar [Muñ1992],

[10] El símbolo *4* es el *prepend* de listas, que agrega el elemento *x* a la lista *L*.

[Car1995], [Tam1998] y [ManDoc]. También se explican los problemas encontrados y las modificaciones que se hicieron.

1.2.1. Motor de Reescritura

Al especificar *TADs* en ManTa, bien sea de forma automática o por especificación inductiva, se verifica automáticamente la definición de un sistema de reescritura confluente[11]; también se verifica que los axiomas de las selectoras conformen un sistema *total*[12] y no sean redundantes o contradictorios sobre ningún valor[13]. De esta forma es posible asegurar que el sistema es convergente; es decir, toda expresión se puede reducir a una única forma normal (llamaremos a E^* la forma normal de la expresión E).

1.2.1.1.Reescritura con Axiomas

La propiedad del sistema de reescritura (determinado por el conjunto de axiomas de los *TADs* del ambiente[14]), de reducir toda expresión a su única forma normal, es de gran importancia para la demostración de conjeturas; esta característica permite usar como estrategia inicial, para la prueba de igualdad entre dos expresiones, la reducción a la forma normal de las expresiones que, si son sintácticamente iguales, se puede afirmar la veracidad de la conjetura ([Car1996]).

Para la reescritura con axiomas se adoptó el régimen *Leftmost − Outermost*, consistente en rescribir primero la expresión más a la izquierda de las que están más afuera (ver *ejemplo 2* en el *Anexo C*). Tomando el árbol sintáctico de la expresión, este régimen describe un recorrido en inorden (primero la raíz, luego los hijos de izquierda a derecha).

Para la reescritura de expresiones es necesario, primero, determinar la regla que se puede aplicar. Antes de abordar el algoritmo de *correspondencia de patrones* se hacen algunas definiciones ([Car1996], [Tam1998]).

1) Llamaremos *Var* al conjunto de variables.

[11] Esta verificación se hace revisando linealidad izquierda y confluencia local ([Tam1998], *Capítulo 3*).
[12] Las selectoras están definidas para todos los valores del tipo.
[13] Los algoritmos utilizados para apoyar la especificación de tipos y verificar las propiedades mencionadas no son descritos en este documento. Se pueden consultar [Man1979] y [Car1996].
[14] Entenderemos por *ambiente* el conjunto de *TADs* y *lemas* cargados en una sesión de ManTa.

2) Llamaremos *Term* al conjunto de términos, donde t ∈ *Term* si se cumple algunas de las siguientes condiciones:
   i. t ∈ *Var*.
   ii. t es valor inicial de algún *TAD* del ambiente.
   iii. t es de la forma $f(E_1, ..., E_n)$, donde $f : X_1 * ... * X_n \not\to X_{n+1}$ es una función (constructora o selectora) de un *TAD* del ambiente y $E_1, ..., E_n \in Term$ (siendo de tipos $X_1, ..., X_n$ respectivamente).
3) Una *sustitución* es una función $\sigma : Var \not\to Term$,
   tal que $\|\{x \mid x \in Term : \sigma(x) \neq x\}\| < \infty$.
   Una sustitución puede escribirse como conjunto así: $\sigma = \{x_1 \Rightarrow t_1, ..., x_n \Rightarrow t_n\}$, donde $x_i \in Var$, $x_i \neq t_i$ y $(\forall v : Var \mid \sigma(v) \neq v : (\exists_j \mid 1 \leq j \leq n : x_j \in Term \wedge x_j = v))$.
4) Llamaremos *Sust* al conjunto de sustituciones (*Sust* = $\{\sigma \mid : \sigma$ es sustitución$\}$).
5) Sean *P*, *O* ∈ *Term*, llamaremos *match* entre el *patrón P* y el *objetivo O* a una sustitución σ, si $\sigma(P) = O$. También se dice que *P corresponde* a *O*.

El algoritmo de *correspondencia de patrones* para rescribir una expresión *E* consiste, entonces, en determinar la *sustitución* σ que es *match* entre el lado izquierdo de un axioma *Ax* (sea éste el *patrón P*) y la expresión *E*, si existe tal correspondencia. Para esto se consideran los siguientes casos:

i. Si $P \in Var$ hay correspondencia y $\sigma = \{P \Rightarrow E\}$.
ii. Si *P* es una constante y *E* = *P* hay correspondencia y $\sigma = \{\}$.
iii. Si *P* es de la forma $f(P_1, ..., P_n)$ y *E* es de la forma $f(E_1, ..., E_n)$, donde *f* es una función (constructora o selectora), entonces hay correspondencia si
   $(\forall_i \mid 1 \leq i \leq n : (\exists \sigma_i \mid \sigma_i \in Sust : \sigma_i(P_i) = E_i))$.
   En caso de que lo anterior se cumpla $\sigma = \sigma_1 \cup ... \cup \sigma_n$.

Finalmente, la reescritura se efectúa reemplazando la expresión en cuestión por la expresión resultante de aplicar la sustitución hallada al lado derecho del axioma que corresponde:

$$si\ \sigma(Ax_{izquierdo}) = E \qquad entonces \qquad E \not\equiv \sigma(Ax_{derecho})$$

El conjunto de reglas de reescritura contiene, además de los axiomas de los *TADs* del ambiente, unas reglas especiales que determinan la semántica de la funciones *EQUAL* e *if−then−else* y son:

- if TRUE then X else Y fi ¢ X
- if FALSE then X else Y fi ¢ Y
- EQUAL(A, B) ¢ EQ_τ(A, B), donde A y B son de tipo T [15].

La forma normal de una expresión es obtenida, entonces, aplicando todas las reglas de reescritura posibles, siguiendo el régimen *Leftmost − Outtermost*, mediante un algoritmo que puede resumirse así:

Sea $E = f(E_1, ..., E_n)$ la expresión a reducir a su forma normal E^*.

1) *Se busca entre el conjunto de axiomas (completado con las reglas de reescritura), uno cuyo lado izquierdo corresponda a E.*
2) *Si existen Ax y* σ *: Var ¢ Term, tal que* $\sigma(Ax_{izquierdo}) = E$*, se reemplaza E por* $\sigma(Ax_{derecho})$ *y se reinicia el proceso con la expresión principal* $\sigma(Ax_{derecho})$*. Si no, se intenta rescribir alguna de las expresiones* $E_1, ..., E_n$*, en ese orden, mediante el mismo proceso.*

La reescritura termina, hallándose así E^**, cuando no es posible aplicar ninguna regla de reescritura.*

1.2.1.2.Manejo del elemento ERROR

Ya se ha mencionado en este capítulo el elemento *ERROR*, que es agregado al *tdi* de todos los *TADs*. La razón por la cual se adoptó el uso de este elemento fue la de permitir la especificación del significado de una función selectora en los valores para los cuales está indefinida.

[15] Esta última regla tiene sólo importancia sintáctica, pero se mencionó porque más adelante se hablará de otras reglas de *EQUAL*.

Este elemento no es considerado un valor y, por tanto, los axiomas no definen un comportamiento cuando un parámetro es *ERROR*. No obstante, el comportamiento en estos casos está dado por el hecho de que las funciones (salvo *if−then−else*) son *estrictas*[16].

Sin embargo, en la versión anterior del *probador de teoremas*, la evaluación de términos, cuando un parámetro reducía a *ERROR*, no estaba establecida ([Tam1998], *sección 3.4.3*).

Podemos ver, ahora, qué trato se le daba al *ERROR* dentro del *probador de teoremas*. La igualdad de las conjeturas ecuacionales, que se manejaba en ManTa, tomaba como verdadera la conjetura si las expresiones izquierda y derecha de la igualdad reducían ambas a *ERROR*. En el próximo capítulo se explica cómo se modifica este comportamiento en el *probador de teoremas*.

Por otro lado, existe un inconveniente con el tipo del *ERROR*, pues este identificador es utilizado igual para todos los *TADs*, lo cual crea inconsistencias en las pruebas, debido a que se puede llegar a demostrar la igualdad entre dos expresiones de diferente tipo (i.e. *head(nil) = top(emptyStack)*, ver ejemplo 3 en el *Anexo C*). El inconveniente que esto generaba se solucionó para la nueva versión, haciendo una revisión de consistencia de tipo, antes de empezar la prueba[17].

1.2.2. Métodos de Prueba

Para conjeturas que sólo tratan valores concretos de los tipos es suficiente reducir las expresiones izquierda y derecha a sus formas normales, para decidir la veracidad de la conjetura. El problema es más complejo cuando la conjetura involucra variables, razón por la cual, en el *probador de teoremas* se cuenta con otros métodos de prueba ([B&M1979]), que permiten decidir sobre un mayor número de conjeturas verdaderas[18].

---

[16] Se dice que una función *f* es *estricta* si *f(..., ERROR, ...) = ERROR*.

[17] Esta revisión incluye verificar que a una misma variable no se le entienda como de dos tipos diferentes dentro de la conjetura. Al hacer esta verificación se obtiene información sobre el tipo de una variable, cuando esta aparece más de una vez y de una de las apariciones no es posible deducir su tipo, pero de otra sí.

[18] Se aclara que las conjeturas sobre las que se puede decidir son las verdaderas, pues el sólo hecho de que fallen estos métodos no conduce a afirmar que la igualdad no se cumple.

De estos métodos es indispensable asegurar su *corrección* y *terminación* para que puedan ser empleados. A continuación se exponen los métodos que ya estaban implementados antes de empezar este trabajo. Aunque había algunos avances en los métodos de *prueba por casos* y *generalización*, estos se dejan para el próximo capítulo porque su comportamiento no fue estudiado mediante pruebas, sino directamente sobre el código fuente.

1.2.2.1.Prueba por Inducción Estructural

Este método se basa en el siguiente teorema:

Sea I_X el conjunto de las operaciones iniciales, C_X el conjunto de las operaciones constructoras y S_X el conjunto de las operaciones selectoras del **TAD *X***.

Sea ψ : **X** ¢ BOOL un predicado.

$(\forall x \mid x \in X : \psi(x)) \equiv$

$(\forall_\alpha \mid \alpha \in I_X : \psi(\alpha)) \wedge (\forall_\beta \mid \beta \in C_X : \psi(x_{i1}, ..., x_{in}) \Rightarrow \psi(\beta(x_{i1}, ..., x_{in})))$[19]

Es decir, para probar un teorema para toda variable x ∈ ***X***, basta probarlo para todos los valores iniciales de ***X*** y, luego, probarlo para un elemento construido con cada operación constructora (asumiendo que el teorema es cierto para todos los parámetros de tipo ***X*** de la constructora).

Descripción del Algoritmo:

Sea $M^*[\boldsymbol{w}] = N^*[\boldsymbol{w}]$ la conjetura que se quiere probar, en la que $M^*[\boldsymbol{w}]$ y $N^*[\boldsymbol{w}]$ son expresiones normalizadas[20] cuyas variables son las contenidas en el vector ***w***.

Para toda variable $v \in \boldsymbol{w}$ (siendo v de tipo ***T***):

Caso trivial: Se prueba $M^*[\boldsymbol{w}(v := i)] = N^*[\boldsymbol{w}(v := i)]$, para todo $i \in I_T$.

Caso Complejo: Para toda $c \in C_T$, se prueba

[19] La demostración del teorema no se incluyó en este documento. Pueden consultarse [Man1974] (Capítulo 5) y [Muñ1992] (Capítulo 2) sobre la corrección y terminación del método.

[20] *E* está normalizada si *E* es igual a su forma normal; es decir, no es posible reducirla con el sistema de reescritura.

$M^*[\boldsymbol{w}(v := c(u, \boldsymbol{z}))] = N^*[\boldsymbol{w}(v := c(u, \boldsymbol{z}))]$, agregando al sistema de reescritura la regla $M^*[\boldsymbol{w}(v := u)] = N^*[\boldsymbol{w}(v := u)]$ (hipótesis de inducción[21]), de modo que u y las variables contenidas en el vector ***z*** no aparecen en ***w***.

Al probar la implementación de este método en la versión anterior del *probador de teoremas* se encontraron algunos problemas (que fueron corregidos), que se explican a continuación:

- No se contemplaba el caso en que el *TAD*, al que pertenece la variable de inducción, no tuviera funciones constructoras; esto hacía que el programa abortara al plantear el *caso complejo* de la inducción, pues dicho caso no existe[22].
- El algoritmo utilizado para crear el vector ***z*** no aseguraba que sus variables no aparecieran ya en la conjetura. Esto conducía a un error de corrección en el método, al estar suponiendo propiedades falsas sobre las nuevas variables. Por ejemplo la conjetura *concat(x, y) = concat(y, x)* era demostrada para todo par de listas *x*, *y* (ver ejemplo 4, *Anexo C*).

1.2.2.2.Prueba Semántica y el Predicado *EQUAL*

Todo **`TAD T`** tiene definida una selectora ***`EQ_T`*** que determina, por medio de sus axiomas, si dos expresiones de tipo ***`T`*** son iguales (independiente de que sean literalmente iguales). En ManTa se reservó la palabra ***`EQUAL`*** para agrupar las funciones de igualdad de todos los *TADs*; de esta manera, se permite no explicitar el tipo de la función de igualdad, en una conjetura o expresión para rescribir (v. gr. al escribir ***`EQUAL`***(A, B) ManTa lo entiende como ***`EQ_NAT`***(A, B), si A y B son expresiones de tipo ***`NAT`***).

La prueba semántica consiste, entonces, en tratar de reducir la expresión ***`EQ_T`****(A, B)* a ***`TRUE`***, en lugar de reducir *A* y *B* a expresiones sintácticamente idénticas (ver ejemplo 5, *Anexo C*).

[21] En el próximo capítulo se explican algunas diferencias que existen entre la reescritura con axiomas y la reescritura con hipótesis.
[22] Al corregir este problema se obtuvo un probador completo para conjeturas que sólo incluyeran variables booleanas; porque, al no haber constructoras, los casos triviales equivalen a una prueba por *tabla de verdad*.

La corrección de este método no está asegurada por el momento, ya que depende de que el predicado ***EQUAL*** sea definido correctamente como una *relación de congruencia*; es decir, debe ser *relación de equivalencia* (*reflexiva*, *simétrica* y *transitiva*) y *preservar funciones*[23].

La terminación de una aplicación de este método se deriva de que el sistema de reescritura es terminante, pues ***EQ_T****(A, B)* se reduce a su forma normal aplicando un número finito de reglas. No obstante, podría darse una aplicación infinita del método si se siguen estos pasos:

$A =_T B$ [24]

¢ ⟨ *Aplicación de la prueba semántica* ⟩

$\mathbf{EQ_T}(A, B) =_{BOOL} TRUE$

¢ ⟨ *Aplicación de reescritura* ⟩

$(\mathbf{EQ_T}(A, B))^* =_{BOOL} TRUE$

¢ ⟨ *Aplicación de la prueba semántica* ⟩

$\mathbf{EQ_BOOL}((\mathbf{EQ_T}(A, B))^*, TRUE) =_{BOOL} TRUE$

¢ ...

En la versión anterior del *probador de teoremas*, este problema se resolvía usando únicamente reglas de reescritura para decidir sobre la veracidad de la conjetura propuesta por la prueba semántica, sin permitir el encadenamiento con otros métodos (más adelante en este capítulo y en el próximo se trata la *aplicación de métodos*).

Dicha restricción fue modificada para mantener homogeneidad entre los métodos; pero se limitó la aplicación de la prueba semántica, de modo que no es usada cuando uno de los parámetros del predicado ***EQUAL*** es un valor básico de tipo BOOL (***TRUE***, ***FALSE***). Con esta limitación no se pierde capacidad de prueba, ya que los axiomas de ***EQ_BOOL*** sólo definen un significado para los valores básicos[25]; así que cuando la conjetura no puede ser probada por otros métodos la prueba semántica no da más herramientas, pues ningún axioma *corresponde* a la expresión.

[23] Sobre estas propiedades se hablará más adelante; por el momento no se dan definiciones.
[24] El subíndice después del símbolo = se refiere al tipo de las expresiones izquierda y derecha de la igualdad. Aunque esta notación no es utilizada en ManTa, sino que se maneja el tipo de la igualdad implícitamente, se adoptó la utilización de estos subíndices en la igualdad para tener una mayor claridad dentro de las explicaciones de este documento.
[25] EQ_BOOL(TRUE, TRUE) = TRUE, EQ_BOOL(TRUE, FALSE) = FALSE, EQ_BOOL(FALSE, TRUE) = FALSE y EQ_BOOL(FALSE, FALSE) = TRUE.

Predicado EQUAL:

Este predicado adquiere una semántica, al igual que las demás funciones selectoras, del conjunto de axiomas que hace parte de la especificación de todo *TAD*[26]. No se puede decir lo mismo de los tipos no específicos (tipos parámetro), para los cuales no se supone semántica alguna, pues mantienen un lugar que será ocupado por un tipo concreto.

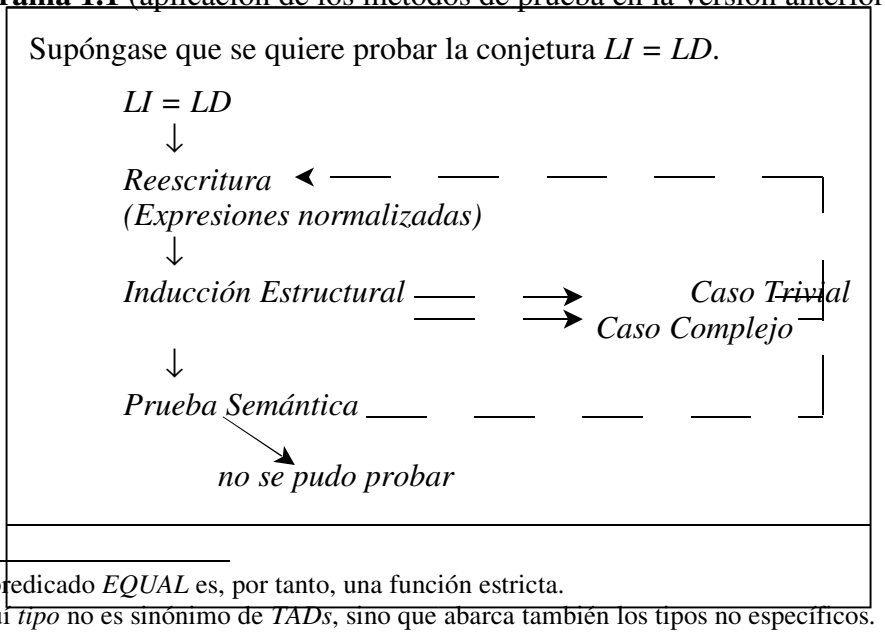
Sin embargo, debido a la sintaxis adoptada en ManTa (en la que el símbolo '=' no es válido dentro de una expresión), el predicado *EQUAL* es utilizado, también, para expresar igualdad sintáctica dentro de las expresiones (i.e. $x \in y4L \equiv x = y \vee x \in L$ se escribe *memberList(cons(y, L), x) = OR(EQUAL(x, y), memberList(L, x)*). Por este motivo, se incluyó una regla de reescritura, para la nueva versión, que es aplicable a tipos concretos y no concretos,

- *EQ_T(E, E) ¢ TRUE*, $\forall T$ *: tipo*[27]*, E : Term.*

1.2.3. Aplicación de los métodos de prueba

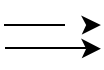
Hasta el momento los métodos de prueba han sido expuestos de forma independiente; a continuación se presenta gráficamente la forma en que se encadenaban los métodos de prueba, para lograr la prueba de una conjetura:

Diagrama 1.1 (aplicación de los métodos de prueba en la versión anterior)



[26] El predicado *EQUAL* es, por tanto, una función estricta.

[27] Aquí *tipo* no es sinónimo de *TADs*, sino que abarca también los tipos no específicos.


Utiliza para llevar a cabo la prueba
Si fracasa le pasa la prueba

2 APORTES AL PROBADOR DE TEOREMAS DE MANTA

En este capítulo se exponen las principales modificaciones y mejoras que se implementan, en este trabajo de tesis, para el *probador de teoremas* de ManTa[1]. Todas estas modificaciones se hacen usando las rutinas ya existentes y la teoría desarrollada por anteriores colaboradores de ManTa.

2.1. REESCRITURA

Para la implementación de las nuevas formas de reescritura, fue necesario modificar las estructuras de datos utilizadas para representar *expresiones* e *igualdades*. En el *Anexo B* se encuentra una descripción de dichas estructuras.

2.1.1. Manejo del Elemento *ERROR*

En cuanto al manejo del elemento *ERROR* se hacen dos modificaciones fundamentales: una consistió en tomar la igualdad débil para los teoremas y, la segunda, consistió en implementar, en el motor de reescritura, reglas para funciones estrictas y la función *if−then−else*.

Como primera medida, el *ERROR* no se toma como elemento del tipo de interés, sino que se asume como un elemento que no tiene la propiedad de ser evaluado por una igualdad. Por esta razón, en la igualdad sintáctica no puede afirmarse que *ERROR* sea igual a sí mismo.

Por otro lado, en ManTa todas las funciones son estrictas, a excepción de *if−then−else*, lo cual garantiza un sistema de reescritura completo ([Man1974]). Esto exige que en el motor

[1] También se implementan otras opciones, que no son mencionadas en este capítulo, por no influir sobre la capacidad de prueba (i.e. limpieza de pruebas). Una explicación sobre todas las nuevas opciones del módulo de teoremas se encuentra en el *Manual de Uso* (*Anexo A*).

de reescritura se cuente con reglas generales para reducir expresiones cuando un parámetro está indefinido.

Para la sintaxis de ManTa se formulan, entonces, las siguientes reglas de reescritura:

1) *f(..., ERROR, ...)* ¢ *ERROR,* ∀ *f: función* | *f* ≠ *if−then−else*.

2) *if ERROR then A else B fi* ¢ *ERROR*.

2.1.2. Reescritura con Lemas

Una práctica común, en la construcción de conocimiento, es la utilización de las afirmaciones que ya se han admitido como válidas, para dar pasos hacia ideas más complejas. Este es, precisamente, el papel que cumplen los lemas y teoremas en las demostraciones matemáticas, pues se trata de verdades probadas dentro de la teoría en que se está trabajando.

En ManTa, ya se ha visto, es posible demostrar igualdades en las cuales se puede confiar, para dar pasos correctos dentro de otras pruebas, manteniendo el principio de cambiar iguales por iguales. Con esta motivación, se construye la opción de conservar las conjeturas que se demuestran, para ser posteriormente usadas a manera de reglas de reescritura.

Puesto que los lemas se guardan con la finalidad de no repetir cálculos innecesarios en las demostraciones, hay ciertos casos en los que no se conservan porque, aunque se trata de reglas correctas, no tienen utilidad para otras pruebas. Parte de la decisión sobre la utilidad de un lema se le deja al usuario, pero ManTa descarta, automáticamente, los teoremas que cumplen alguna de las siguientes condiciones:

- El teorema es instancia de un axioma o regla especial de reescritura[2].
- El teorema es instancia de un lema ya existente.
- El teorema, intercambiando el lado izquierdo con el derecho, cumple alguna de las condiciones anteriores.

[2] Las reglas especiales de reescritura se usan igual que los axiomas y tratan las funciones *EQUAL* e *if−then−else*, así como los casos en que un parámetro está indefinido.

Los lemas, entonces, son utilizados en el motor de reescritura como si se tratara de axiomas, tomando en cuenta algunas limitaciones y diferencias, que se describen a continuación:

Terminación

Cuando se definen *TADs* en ManTa, se garantiza que los axiomas constituyen un sistema de reescritura terminante, lo cual no es cierto para los lemas (v. gr. en los números naturales, la conmutatividad de la suma es siempre aplicable). Por esta razón, es necesario disponer de una estrategia que asegure que la aplicación de lemas no conduce a un ciclo interminable de reescritura de expresiones.

Una primera posibilidad está en dejar a decisión del usuario la aplicación de cada lema, que ya se sabe unificable[3]. En el *probador de teoremas* de ManTa existen dos opciones sobre la aplicación de lemas: un modo manual, en el que se le pregunta al usuario antes de aplicar un lema; y otro automático, que exige otra estrategia para evitar la no terminación al rescribir.

Una manera más automática es usar cada lema sólo una vez en una expresión, lo cual requiere tener una estructura de datos, para la representación de expresiones, que lleve el registro de los lemas que han sido utilizados. En ManTa se implementa una extensión natural de esta estrategia, que permite modificar el número de veces que pueden ser aplicados los lemas, con lo cual se mantiene la idea de tener un número finito de pasos.

Los lemas son usados para cambiar una subexpresión por otra igual, de modo que el registro debe llevarse sobre todas las subexpresiones, para que la aplicación de un lema en una expresión no lo inhabilite para las subexpresiones que la componen (véase el *ejemplo 6* en el *Anexo C*).

[3] Un lema es *unificable* si uno de los lados de la igualdad corresponde a la expresión que se está rescribiendo.

Por último, existe un algoritmo descrito en [B&M1979], con el que se automatiza el proceso. El algoritmo consiste en que dada una regla *R1* ($li_{R1} = ld_{R1}$), que ya fue aplicada y puede ser aplicada por segunda vez, se aplica[4] *R1* a la expresión ld_{R1} y se compara el resultado (*ld*) con li_{R1}. Si son diferentes, la regla puede ser aplicada por segunda vez, sin que esto conduzca a retroceder en la demostración. Este procedimiento se puede seguir para decidir sobre la aplicabilidad de la regla por n−ésima vez: se aplica *R1* sobre *ld*, para obtener *ld*, y se sigue aplicando sobre el resultado hasta tener *ld*, que se compara con li_{R1} y, de ser diferente, el lema puede ser aplicado.

Esta última forma de control no se usa en el *probador de teoremas* de ManTa. Para poder usarla, sin embargo, habría que ampliar el algoritmo aplicando sobre ld_{R1} todas las reglas posibles, para garantizar que no hubiera ciclos debidos a la aplicación alternada de un conjunto de reglas[5].

Dirección de la Igualdad

Los axiomas dan semántica a las selectoras de un *TAD* y son las reglas de reescritura con las que se puede reducir una expresión a su forma normal, por lo cual son igualdades con dirección[6], para rescribir una expresión con otra más simple. De los lemas, por el contrario, no se afirma que una de las expresiones sea más compleja que la otra, razón por la cual, al querer utilizar lemas como reglas de reescritura, se toma la igualdad en ambas direcciones (i.e. teniendo el lema $li_L = ld_L$ se tienen las reglas $li_L \¢ ld_L$ y $ld_L \¢ li_L$).

Existen, sin embargo, dos excepciones a la utilización bidireccional de los lemas, por tratarse de casos en los que una de las expresiones es, evidentemente, más simple que la otra. Cuando uno de los lados de la igualdad es una variable o una constante, se puede tomar dicha expresión como más simple que la del otro lado; por lo tanto, los lemas que

[4] Si *R1* no es aplicable a ld_{R1} no habría conflicto.
[5] En este trabajo no se ahonda más en este aspecto de la aplicación de lemas (en [B&M1979] pueden encontrarse más detalles al respecto).
[6] Se dice que los axiomas tienen dirección porque, si bien los dos lados son efectivamente iguales, la reescritura de iguales por iguales sólo se hace cuando la expresión unifica con el lado izquierdo de la igualdad, en cuyo caso dicha expresión es reemplazada por el lado derecho. Es decir, el axioma $li_{Ax} = ld_{Ax}$ debe entenderse como $li_{Ax} \¢ ld_{Ax}$.

cumplen una de estas condiciones son usados como reglas sólo para reemplazar la expresión compleja por la simple.

Nuevas Variables en la Expresión que Rescribe

Existen lemas en los que una variable aparece sólo en uno de los lados de la igualdad (v. gr. en el *TAD Stack*, $s \neq \emptyset \mid\!\!- isEmpty(x4s) = isEmpty(s)$). Cuando esto ocurre, la variable no es sustituida por otra al unificarse el lema, por lo cual aparece en la expresión con el nombre que tenía en el lema.

Cuando un lema con esta característica es utilizado para rescribir una subexpresión, se corre el riesgo de que la variable que sólo aparece en uno de los lados del lema tenga el mismo nombre de una variable existente en otro lugar de la expresión. De darse este caso se perdería generalidad en la conjetura y el resto de la prueba sería sobre una igualdad cuyo cumplimiento no permite deducir que la conjetura inicial sea verdadera.

Para mantener, entonces, la corrección al aplicar lemas, en ManTa se contemplan estos lemas y se crean variables nuevas para sustituir todas las apariciones de las variables que no aparecen en el lado del lema que corresponde a la subexpresión a rescribir, de modo que no se suponen relaciones entre dichas variables y las ya existentes en la conjetura.

Tipo del Lema y TADs involucrados

Una conjetura en ManTa puede involucrar funciones de diferentes tipos, debido a que la definición de un *TAD* se apoya en los que ya están definidos y a que los teoremas se usan para verificar propiedades de las relaciones existentes en el modelo del usuario. No obstante, a los lemas se les asigna un tipo, para que sean cargados al ambiente cuando el *TAD* se carga, con el objeto de evitar ocupar memoria sin necesidad.

Por lo anterior, es necesario determinar el tipo de los teoremas probados, cuando se quieren conservar como lemas. Esta decisión se le deja al usuario, a quien se le presentan las siguientes opciones (tomemos la conjetura $f(\boldsymbol{a}) = g(\boldsymbol{b})$):

1) Codominio de *f* (que tiene que ser el mismo de *g*).

2) *TAD* al que pertenece *f*.

3) *TAD* al que pertenece *g*.

Al aplicar el lema surge otra dificultad, debida al hecho de tener diferentes tipos involucrados, ya que puede rescribirse una expresión con otra que hace referencia a funciones de otros tipos; por tal motivo, es necesario corroborar que la expresión que rescribirá es totalmente entendida en el ambiente.

2.1.3. Reescritura con Hipótesis

Las hipótesis son tomadas como atributo de las expresiones, pues son usadas como reglas de reescritura, para cambiar una expresión por otra igual. Sin embargo, existe una diferencia esencial entre las reglas de reescritura mencionadas anteriormente y las hipótesis; ésta es que las variables de una hipótesis deben ser tomadas como constantes, pues sería incorrecto hacer sustitución sobre ellas[7] ([Gri1994]).

Debido a esta diferencia, para determinar si una hipótesis puede ser usada no se revisa si el lado izquierdo *corresponde* a la expresión que se quiere rescribir (ver *1.2.1.1. Reescritura con axiomas*), sino que es necesario tener una hipótesis cuyo lado izquierdo sea sintácticamente idéntico a dicha expresión (v. gr. $x \vee y$ no es una hipótesis aplicable a la expresión $p \vee q$).

Un *contexto* es una conjunción de predicados ecuacionales que se suponen verdaderos para una expresión, de modo que dicha información puede ser usada para calcular el valor de la expresión. Dada una expresión *E*, el *probador de teoremas* define un contexto *h*, que restringe las variables de *E* y, por tanto, su significado.

---

[7] Por ejemplo, podría llegarse a un resultado incorrecto si quisiéramos demostrar a = *b* |— *f(b)* = *f(c)* y hacemos la siguiente prueba:

f(b) = *f(c)*
= ⟨ Hipótesis a = b, con *a* := *c* ⟩
f(b) = *f(b)* — Identidad de = —

En una prueba en la que exista una expresión con contexto (usaremos $(| h : E)$ para denotar que la expresión E está condicionada por el contexto h), es lícito usar h para rescribir E. Así, por ejemplo, la expresión

$$(| x = y : x4L =_{List} y4L)$$

puede rescribirse en

$$(| x = y : y4L =_{List} y4L)$$

y, por tanto,

$$(| x = y : true)$$

lo cual es equivalente a *true*.

En ManTa esta manipulación se encuentra inmersa en el *probador de teoremas*. El manejo de los contextos que se hace puede sintetizarse con las siguientes reglas:

i. $(| : E) \equiv (| true : E) \equiv E$. Este es el caso en que no se hacen suposiciones que den un contexto para la reescritura de la expresión.

ii. $(| h : I) \equiv I$, donde I es un valor *inicial*. Esta regla se debe a que un valor *inicial* no puede ser rescrito.

iii. $(| false : E) \equiv true$. Cuando las hipótesis son falsas, sin importar el valor E, la expresión con contexto es verdadera.

iv. $(| h \wedge e_1 = e_2 : E(e_1)) \equiv (| h \wedge e_1 = e_2 : E(e_2))$[8]. Con esta regla se ilustra la forma en que son utilizadas las hipótesis que conforman el contexto para rescribir la expresión[9].

Nótese que los contextos no son modificados, pues éstos son supuestos externos a la expresión (i.e. los contextos se usan para rescribir la expresión, pero son supuestos hechos en base a la posición de la expresión dentro de una conjetura o de la demostración). Existen diferentes motivos utilizados en el *probador de teoremas*, por los cuales se supone como verdadera una igualdad. Estos motivos se enuncian a continuación:

1) Aplicación del *Teorema de la Deducción* (véase *2.2. Conjeturas con Hipótesis*). En este caso la hipótesis es considerada verdadera para las dos expresiones igualadas en la conjetura.

[8] $E(e_1)$ se usa para denotar que e_1 es subexpresión de la expresión E.
[9] Es importante apuntar que la igualdad de las hipótesis tiene una dirección; por esto, no se podría rescribir e_2 con e_1.

2) Aplicación de los métodos de *inducción* o *prueba por casos*, en los que la suposición de una hipótesis hace parte del método. Aquí, la hipótesis es también para las dos expresiones.
3) Si *E* es parámetro de otra expresión *E'*, todas las hipótesis de *E'* son consideradas verdaderas para *E*, ya que lo supuesto para *E'* puede suponerse para toda la expresión.
4) Si *E* es parte de otra expresión de la forma *IMP(H, E)*, entonces $H =$ ***TRUE*** es una hipótesis verdadera para *E*. La validez de rescribir con estas hipótesis no se deriva de la regla de Leibniz (*cambiar iguales por iguales*), sino del hecho de que para probar $f(..., H \Rightarrow C, ...)$ basta probar $f(..., H \Rightarrow C, ...)$, como se demuestra en el Metateorema 1 (*Anexo D*).

En todos los casos anteriores las hipótesis son expresiones *booleanas*, que se suponen equivalentes a *true*. Para que sean usadas como reglas ecuacionales se convierten en igualdades, dependiendo de la forma de la expresión *H*:

1) $H \equiv EQ_X(A, B)$, hipótesis ecuacional: $A =_X B$.
2) $H \equiv NOT(H')$, hipótesis ecuacional: $H' =_{BOOL} false$.
3) $H \equiv AND(H_1, H_2)$, las hipótesis son H_1 y H_2.
4) En otro caso la hipótesis ecuacional es $H =_{BOOL} true$.

2.2. CONJETURAS CON HIPÓTESIS

Ya se mencionó que el segundo parámetro de la función *IMP* toma como hipótesis el primer parámetro. No obstante, esta exposición se hizo sobre expresiones en general, que podían ser subexpresiones de otras; pero existe un caso particular, de especial interés para las conjeturas ecuacionales, que es el de una conjetura de la forma $\boldsymbol{h} \Rightarrow \boldsymbol{a} = \boldsymbol{b}$[10], porque puede reformularse la conjetura.

El *Teorema de la Deducción* permite que una conjetura de la forma $p \Rightarrow q$ sea reformulada y la prueba se limita a demostrar *q*, usando *p* como hipótesis. Tratándose de conjeturas

[10] Escrita en la sintaxis de ManTa: $IMP(\boldsymbol{h}, EQ_X(\boldsymbol{a}, \boldsymbol{b})) =_{BOOL} TRUE$.

ecuacionales, este teorema sólo es aplicable si la consecuencia es una igualdad, en cuyo caso se cambia el tipo de la conjetura.

Por esta razón, en el probador de teoremas de ManTa se entienden las conjeturas de la forma $\boldsymbol{h} \Rightarrow \boldsymbol{a} =_X \boldsymbol{b}$ como $\boldsymbol{h} \vdash \boldsymbol{a} =_X \boldsymbol{b}$; es decir, se intenta probar la igualdad entre ***a*** y ***b***, usando ***h***[11] como regla de reescritura, en lugar de intentar reducir la conjetura inicial a *true* (ver el *ejemplo 7*, *Anexo C*).

La utilidad del *Teorema de la Deducción* como herramienta de demostración es bien conocida. Para el *probador de teoremas* de ManTa se obtiene un beneficio mayor, porque hace posible plasmar en la conjetura el contexto dentro del cual se quiere que el teorema sea verdadero, permitiendo el uso de información adicional para la prueba. Un buen ejemplo de la aplicación de esta herramienta es la verificación del predicado *EQUAL* (sección *2.5* de este documento).

2.3. MÉTODOS DE PRUEBA

2.3.1. Prueba por Casos

La función *if−then−else* tiene un comportamiento particular en el motor de reescritura (ver *1.2.1.1. Reescritura con Axiomas*), debido al cual las expresiones del segundo y tercer argumento no son reducidas. Con esta función es posible expresar un comportamiento condicional, al especificar la semántica de las funciones selectoras.

Pero se da el caso en que, dado el contexto en que aparece una función selectora, su resultado es el mismo, independiente de que la condición sea verdadera o falsa. Por este motivo se implementa en el *probador de teoremas*, un método en el que se intenta probar la conjetura asumiendo la condición de la función *if−then−else*, primero como verdadera y luego como falsa.

[11] En *2.1.3. Reescritura con Hipótesis* se vio cómo se traduce ***h*** a una hipótesis ecuacional.

Si tenemos, entonces, una conjetura de la forma $f(\boldsymbol{x}, \text{ if } C \text{ then } A \text{ else } B \text{ fi}, \boldsymbol{y}) = g(\boldsymbol{w})$ (donde $\boldsymbol{x}$, $\boldsymbol{y}$ y $\boldsymbol{w}$ son vectores arbitrarios de expresiones) podríamos reformularla y probar la conjetura $C \vee \neg C \mid\!\!- f(\boldsymbol{x}, \text{ if } C \text{ then } A \text{ else } B \text{ fi}, \boldsymbol{y}) = g(\boldsymbol{w})$, cuya validez estriba en que $C \vee \neg C \equiv true$. Otra forma de ver la corrección de este método es considerando el teorema $E \wedge E \equiv E$ (demostración en el *Anexo D*)[12], donde $E := f(\boldsymbol{x}, \text{ if } C \text{ then } A \text{ else } B \text{ fi}, \boldsymbol{y}) = g(\boldsymbol{w})$, $z := C$ y $p := C$.

En ManTa este método se aplica, entonces, planteando dos conjeturas independientes (una en la que se supone $C = true$ y otra en la que se supone $\neg C = true$[13]), que de poderse demostrar permitirían asegurar que la conjetura inicial es verdadera:

> *Probar* $f(\boldsymbol{x}, \text{ if } C \text{ then } A \text{ else } B \text{ fi}, \boldsymbol{y}) = g(\boldsymbol{w})$ *es equivalente a probar de forma independiente:*
>
> **(1)** $f(\boldsymbol{x}, A, \boldsymbol{y}) = [g(\boldsymbol{w})]$ y
>
> **(2)** $f(\boldsymbol{x}, B, \boldsymbol{y}) = [g(\boldsymbol{w})]$.

2.3.2. Prueba por Shannon

En este trabajo de tesis se desarrolló un método que luego fue desechado, por ser equivalente a la *prueba por casos*. Este método consiste en la aplicación del *Teorema de Shannon*, que enuncia que $E \equiv (p \wedge E) \vee (\neg p \wedge E)$ ([Gri1994]), lo cual permite reformular la conjetura.

Este método se implementó, al igual que la *prueba por casos*, para expresiones con la función *if−then−else* en el cuerpo, de modo que para probar $f(\boldsymbol{x}, \text{ if } C \text{ then } A \text{ else } B \text{ fi}, \boldsymbol{y}) = g(\boldsymbol{w})$ se intenta probar alguna de las siguientes conjeturas:

(1) $([f(\boldsymbol{x}, \text{ if } C \text{ then } A \text{ else } B \text{ fi}, \boldsymbol{y}) = g(\boldsymbol{w})] \wedge C)$

(2) $([f(\boldsymbol{x}, \text{ if } C \text{ then } A \text{ else } B \text{ fi}, \boldsymbol{y}) = g(\boldsymbol{w})] \wedge \neg C)$

[12] Se trata de una idea equivalente a hacer una *tabla de verdad* para la variable z.
[13] En *2.1.3. Reescritura con Hipótesis* se vio cómo se convierte una hipótesis *booleana* en *ecuacional*.

Este método resulta ser menos efectivo que la *prueba por casos*, pues su utilidad se pierde al tener que reducir *C* a *true* o *false* para poder afirmar que alguna de las nuevas conjeturas es verdadera; y, como se mencionó en la *prueba por casos*, la necesidad de este método es para los casos en los que el cumplimiento de la condición de *if−then−else* no afecta la veracidad de la conjetura.

La utilidad que tiene, entonces, el *Teorema de Shannon* es que permite demostrar la corrección del método de *prueba por casos* (ver *Anexo D*, *metateorema 2*).

2.3.3. Generalización

En ocasiones resulta más fácil demostrar una conjetura más débil[14] que la planteada inicialmente (por supuesto, cuando la veracidad se mantiene), porque se pueden aplicar métodos como *inducción estructural*, sobre una variable nueva. Además, al debilitar la conjetura se obtiene una hipótesis de inducción también más general.

La escogencia de una conjetura más general puede ser entendida como el proceso inverso de hallar la sustitución *match* entre un *patrón* y un *objetivo*. Dicho *match* se definió como una sustitución σ : *Var* ¢ *Term*, con la que se obtiene una correspondencia entre las dos expresiones. El problema ahora consiste en tomar un término de una expresión *E* y sustituirlo por una variable libre[15], para obtener una nueva expresión *E'*, de manera que exista *match* entre *E'* (*patrón*) y *E* (*objetivo*).

Resulta evidente que este problema de decisión es más complejo y se necesita alguna heurística para escoger una generalización apropiada. En [B&M1979] se dan algunas pautas sobre la aplicación de la generalización en forma automática, pero no hay una forma infalible de hacerlo sin un gran costo computacional[16]. Como una primera aproximación al problema se optó por dejar al usuario la decisión sobre la generalización, mostrándole las posibilidades existentes.

[14] Por *más débil* se entiende que la conjetura generalizada abarca un dominio más amplio que la inicialmente planteada.
[15] Se llama *variable libre* a una variable que no esté cuantificada; en ManTa esto es equivalente a poner a la variable un nombre que no aparezca en la conjetura.
[16] Siempre sería posible intentar la prueba de todas las conjeturas producto de una generalización, pero esto lleva a un número muy grande de posibilidades.

De modo, pues, que el método de *generalización* que se implementa, extrae una lista de términos presentes en la conjetura y muestra al usuario cómo quedaría la conjetura, para cada término listado, al reemplazar todas sus apariciones por una variable libre. Con base en dicha lista, el usuario escoge una de las opciones y la conjetura se generaliza automáticamente (ver *ejemplo 8*, *Anexo C*).

La corrección de este método resulta obvia si vemos la conjetura generalizada como un lema a probar, que puede ser usado para probar la conjetura inicial.

> *Sea M = N la conjetura sin generalizar. Se escoge generalizar un término t, obteniendo así una conjetura nueva M' = N', donde M' y N' son el resultado de reemplazar todas las ocurrencias de t por una nueva variable v en M y N, respectivamente.*
>
> *Supóngase, ahora, que se prueba M' = N' (pues de lo contrario no se puede afirmar nada sobre la veracidad de M = N).*
>
> *Podemos, entonces, tomar M' = N' como un lema, en la cual se puede sustituir la variable v por una expresión, sin que esto modifique su valor (i.e. es una expresión booleana verdadera).*
>
> *Reemplazamos, entonces, la variable v por el término t:*
>
> *M' = N'*
> = ⟨ Leibniz ⟩
> *(M' = N')[v := t]*
> = ⟨ *sustitución distribuye sobre* = ⟩
> *M'[v := t] = N'[v := t]*
>
> *y esta última expresión ya sabemos, por la forma como se construyó M' = N', que es M = N. De aquí concluimos que para probar la conjetura inicial basta con probar una generalización.*

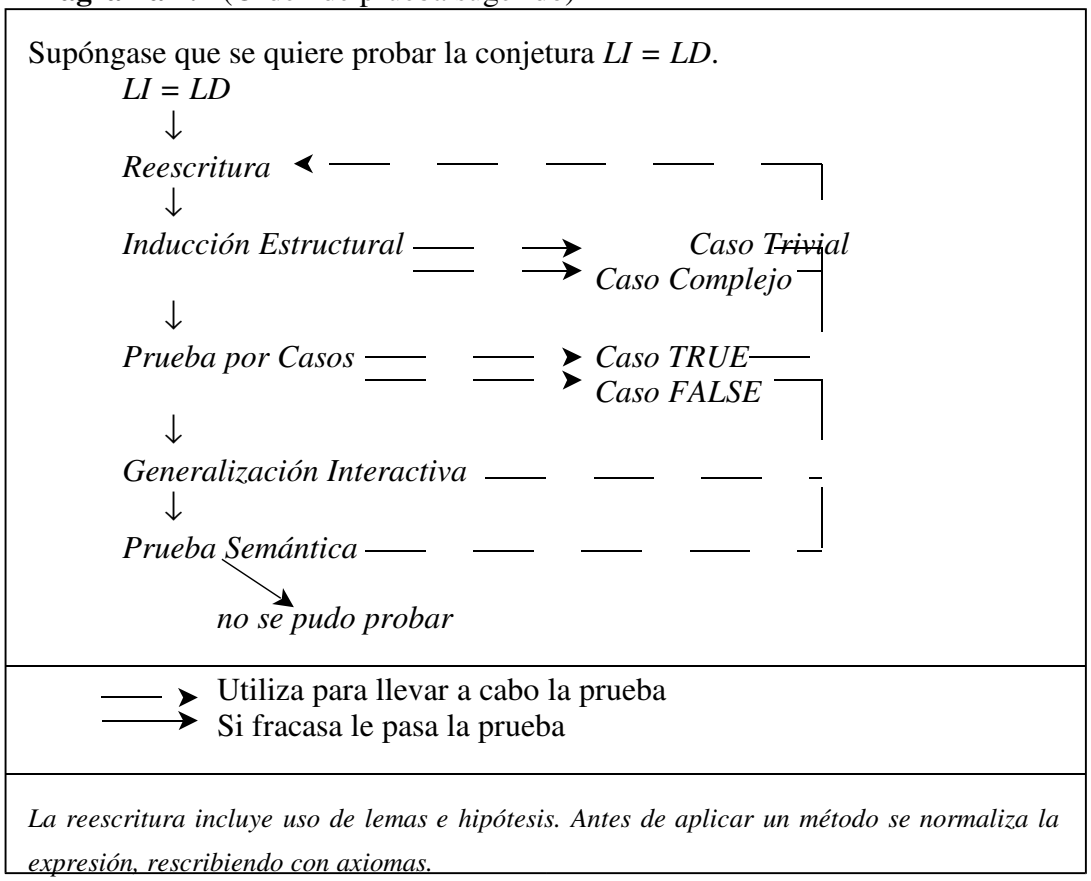
2.4. APLICACIÓN DE LOS MÉTODOS DE PRUEBA

Contando ya con los métodos mencionados, se predefinió un encadenamiento de estos para llevar a cabo la demostración de un teorema (*Diagrama 2.1*), que se basa en la efectividad de los métodos para la mayoría de las pruebas. Este orden sugerido es una ampliación del planteado en *1.2.3. Aplicación de los Métodos de Prueba*.

Sin embargo, para diferentes conjeturas, resultan más efectivos órdenes distintos. Por esta razón, se implementó el orden de los métodos, de modo que puede ser fácilmente cambiado por el usuario.

Con esta opción se pueden aplicar en cualquier orden y repitiendo el número de veces que se quiera, los métodos de *reescritura con lemas*, *reescritura con hipótesis*, *inducción estructural*, *prueba por casos*, *generalización interactiva* y *prueba semántica*. La *reescritura con axiomas* es aplicada al principio de cada método, para normalizar las expresiones de la conjetura

Diagrama 2.1 (Orden de prueba sugerido)

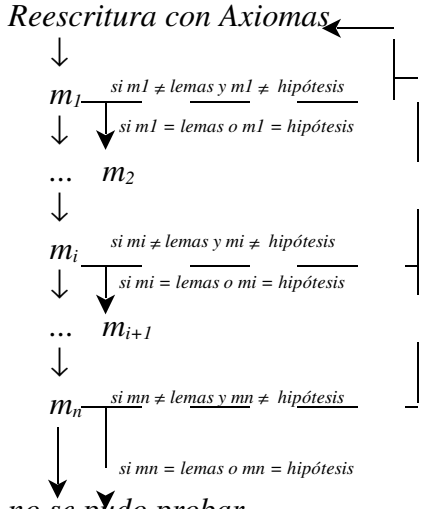


Si se tiene, por ejemplo, un orden $o = \{m_1, ..., m_n\}$ la prueba se empieza por normalizar las expresiones, luego se aplica el método m_1 que modifica la conjetura e intenta probarla con apoyo de todos los métodos aplicados en el orden *o*. Si no se puede demostrar con m_1, se intenta con m_2, que a su vez se apoya en los métodos m_1, ..., m_n; y así, sucesivamente, se intenta hasta llegar a m_n o demostrar la conjetura (Gráficamente este proceso aparece en el *Diagrama 2.2*).

Si bien la reescritura con lemas e hipótesis es tratada como cualquier otro método, existe una diferencia en su aplicación, pues al aplicar la reescritura, la conjetura es modificada y en el siguiente método se intentará probar la conjetura como quedó después de rescribir con los lemas o las hipótesis. Cuando se aplica uno de los otros métodos, por el contrario, si no fue posible hacer la prueba, se le pasa al siguiente método la conjetura tal como fue recibida.

Diagrama 2.2 (aplicación de métodos en el orden $o = \{m_1, ..., m_n\}$)

Sea *LI = LD* la conjetura a probar.



⟶ Una vez aplicado el método se intenta probar la nueva conjetura

⟶ Si la prueba falla se pasa la conjetura[17]

En el *Anexo C* se encuentran varios ejemplos, con diferentes órdenes de aplicación de métodos, en los que se ilustra la utilidad de flexibilizar la prueba de esta forma.

2.5. APOYO A LA VERIFICACIÓN DEL PREDICADO EQUAL

Como ya se dijo, la corrección de la prueba semántica depende de la definición del predicado ***EQUAL*** del *TAD* en cuestión. Este predicado debe ser una *relación de congruencia*; es decir, cumplir con las propiedades de *reflexividad*, *simetría*, *transitividad* y *preservación funciones*.

[17] Cuando el método no es *reescritura con axiomas, lemas* ni *hipótesis* la conjetura que se pasa es la que se recibió.

El requisito de *preservación funciones* consiste en el cumplimiento de la regla de Leibniz, que postula que [Gri1994]:

> *Dos expresiones son iguales en todo estado si y sólo si reemplazar una por la otra, en cualquier expresión E, no cambia el valor de E.*

$$\frac{X = Y}{E[z := X] = E[z := Y]}$$

Con base en lo anterior, se implementa un método de verificación consistente en probar que (dado que el predicado afirma que *x* e *y* son iguales) *x* se puede reemplazar por *y* en todos los parámetros de las selectoras del *TAD*, manteniendo la expresión igual.

El procedimiento para llevar a cabo esta verificación consiste en crear una serie de conjeturas, que intentan ser demostradas en el *probador de teoremas* (ver *ejemplo 10*, *Anexo C*).

> Tomemos la especificación de un ***TAD Y***, en la que se definieron *iniciales* (I_Y), *constructoras* (C_Y) y *selectoras* (S_Y), junto con sus axiomas (Ax_Y).
>
> Sea Ax_{EQ_Y} el conjunto de axiomas del predicado ***EQ_Y***.
>
> Para toda $s \in S_Y$, tal que $s \neq$ ***EQ_Y***, donde *s* : Z_1 * ... * Z_n ¢ Z_{n+1} es la signatura de *s*, se crea una conjetura por cada Z_i que sea igual a ***Y*** y por cada elemento del conjunto Ax_{EQ_Y}, de la siguiente forma (Siendo ***EQ_Y***$(a, b) = f(a, b) \in Ax_{EQ_Y}$, $f(a, b) \neq false$ y Z_k = **Y**):
>
> $$f(a, b) \vdash s(z_1, \ldots, z_{k-1}, a, z_{k+1}, \ldots, z_n) = s(z_1, \ldots, z_{k-1}, b, z_{k+1}, \ldots, z_n).$$

Esta verificación es sólo un apoyo a la decisión del usuario sobre la validez del predicado ***EQUAL***, pues de la falla en la prueba de alguna de las conjeturas propuestas no se puede deducir que el predicado sea inválido. Además, la verificación no incluye pruebas sobre la *reflexividad*[18], *simetría* y *transitividad* de la función.

[18] Todo predicado ***EQUAL*** se puede asegurar que es reflexivo, por la existencia de la regla especial de reescritura ***EQ_T***(E, E) ¢ *TRUE*.

3 CONCLUSIONES

Se ha explicado, en los capítulos anteriores, el funcionamiento general de ManTa y, en mayor detalle, la teoría sobre la que se desarrolla el *probador de teoremas*. Durante este trabajo de tesis quedaron algunos elementos sin considerar, que son dejados para desarrollos posteriores. En este capítulo se enuncian esos problemas que no se resolvieron y se sugieren algunas ideas, sobre las que podría trabajarse para el *probador de teoremas*.

3.1. PROBLEMAS CONOCIDOS DEL PROBADOR DE TEOREMAS

Parámetros Indefinidos en Leftmost − Outermost[1]

Como se expuso en la sección *1.2.1.1.*, *Reescritura con Axiomas*, el régimen utilizado en el motor de reescritura de ManTa es *Leftmost − Outtermost*, que fue escogido por tener mejores cualidades sobre las otras opciones ([Tam1998]). Sin embargo, al aplicar esta regla no se hace una revisión exhaustiva de la definición de las subexpresión[2].

El problema radica en que la primera expresión a la que se le busca un *patrón correspondiente* es a la principal (raíz del árbol sintáctico) y, de encontrarse, la reescritura se lleva a cabo, aun si el *patrón* encontrado tiene una variable en la posición correspondiente a una subexpresión cuya forma normal es *ERROR*. De modo, pues, que puede darse una situación como esta:

> Sea $E = f(E_1, ..., E_i, ..., E_n)$, para n ≥ 1, la expresión a rescribir y Ax ($Ax_{izq} = Ax_{der}$) un axioma o regla especial de reescritura tal que existe un σ que es *match* entre Ax_{izq} y E; es decir, $Ax_{izq} = f(X_1, ..., X_n)$, donde $\sigma(X_j) = E_j$ ($1 \le j \le n$). Según lo que se ha dicho, E es reemplazado por $\sigma(Ax_{der})$.
>
> Supóngase, ahora, que $E_i^* = ERROR$, luego $\sigma(Ax_{der})^* = E^* = ERROR$, porque f es una función estricta.

[1] Al editar este documento este problema ya había sido solucionado. La solución consistió en cambiar el régimen de reescritura a *Leftmost − Innermost*.
[2] En el *Anexo C* se presenta un ejemplo de este problema, para el *TAD Stack* (*ejemplo 13*).

Pero, si $Ax_{der} = g(X_1, ..., X_{i-1}, X_{i+1}, ..., X_m)$ (es decir, no depende de X_i), obtendremos que $\sigma(Ax_{der})^* \neq ERROR$, lo cual contradice el hecho de que la forma normal es única.

La forma que se ha planteado para solucionar esta eventual incorrección en la reescritura, consiste en hacer una verificación de definición con un régimen *innermost*[3]. De este modo, la definición de una expresión se calcula de acuerdo a la definición de sus parámetros y, así, se cumple la regla $def(f(E_1, ..., E_n)) \Rightarrow def(E_1) \wedge ... \wedge def(E_n)$[4].

Seguimiento de la Prueba

Dentro de la nueva funcionalidad de ManTa se introdujeron métodos en los que se da parte del control al usuario (*generalización* y *modo manual de aplicación de lemas*); ello hace necesario que éste disponga de cierta información sobre la prueba. Con este fin, dentro de estas opciones se muestran al usuario la conjetura y las acciones posibles.

Sin embargo, cuando se le hace una pregunta, el usuario desconoce cómo se ha desarrollado la prueba hasta el punto en cuestión, lo cual dificulta mucho su decisión.

Sería conveniente poder mostrar la diferente información en varias ventanas (i.e. especificación de los TADs, evolución de la prueba, lemas cargados, etc.). Pero, sin necesidad de un cambio sustancial de la interfaz, podría mostrarse en un archivo el desarrollo que se lleva de la prueba.

3.2. PROPUESTAS PARA IMPLEMENTAR EN EL PROBADOR DE TEOREMAS DE MANTA

Los temas tratados en este subcapítulo presentan algunas ideas que surgieron durante el desarrollo de la nueva versión del *probador de teoremas*, las cuales no se desarrollaron en su totalidad. Las ideas expuestas son sólo una primera aproximación, sobre las que debe hacerse un mayor ahondamiento para conocer su verdadera utilidad dentro de ManTa.

[3] Se verifica primero la definición de las subexpresiones más internas.
[4] Una mayor documentación sobre este problema se encuentra en [ManDoc], *ManTa 3 − Descripción Formal*.

Aprovechamiento de Información sobre el TAD

Para la especificación de *TADs* en ManTa existen algunas opciones con las que la tarea es llevada a cabo de forma semi−automática, gracias a lo cual es posible mantener un registro con información sobre algunas relaciones entre los tipos del ambiente. Esta información se deriva del hecho de que un tipo puede ser definido con base en los existentes, por *copia*, *restricción*, *extensión* o *producto cartesiano*.

Basándose, entonces, en dicha información, se podrían aplicar lemas de un tipo en otro, haciendo traducción de los nombres de las funciones, mediante la biyección que existe entre ellas, dada la relación establecida al crear el TAD (ver *Tabla 3.1*).

Tabla 3.1 (utilización de lemas dada la creación del *TAD*)

TADs base	***TAD construido***	***Método de Construcción***	***Utilización de lemas***
T_1	T_2	Copia	Los lemas de T_1 se pueden usar para T_2 y viceversa.
T_1, T_2	T_3	Producto Cartesiano	Los lemas de T_1 y T_2 pueden usarse para T_3.
T_1	T_2	Restricción	Los lemas de T_1 pueden usarse para T_2, por abarcar un tdi que contiene el de T_2.
T_1	T_2	Extensión	Los lemas de T_2 pueden usarse para T_1, porque T_2 contiene a T_1.

Ejemplo:
Sea NAT1 un *TAD* creado por restricción de NAT, donde $\forall x \in$ NAT1 : $x \neq 0$.
El lema

$x +_{NAT} y = y +_{NAT} x$ (lema de NAT)

puede ser usado, también, como

$x +_{NAT1} y = y +_{NAT1} x$ (lema de NAT1).

Para poder hacer uso de la información dada por la creación del tipo, es necesario asegurar que los tipos involucrados no han sido modificados, pues de no hacerse esta revisión se podría llegar a incorrecciones, debido a que no existe ya un isomorfismo entre los TADs. En el ejemplo de arriba, los axiomas de la suma podrían cambiarse por una definición sin el cero, por lo cual sería incorrecto intercambiar $+_{NAT}$ por $+_{NAT1}$.

Prueba por Casos para Disyunción de Hipótesis

Ya se implementó en ManTa la reescritura con hipótesis (*sección 2.1.3.*), en la que se hace una traducción de la hipótesis *booleana* a una igualdad, contemplando tres casos especiales. El caso en que la hipótesis *booleana* es una disyunción de expresiones no es contemplado actualmente y éste permitiría hacer una reformulación de la prueba, mediante los casos en que una disyunción es verdadera.

Teniendo una hipótesis de la forma $H_1 \vee H_2$ se puede, cuando la hipótesis es válida para toda la conjetura, partir la demostración en dos partes, haciendo uso del siguiente metateorema (ver *metateorema 3*, *Anexo D*):

$H_1 \vee H_2 \vdash E_{izq} = E_{der}$ si y sólo si $H_1 \vdash E_{izq} = E_{der}$ y $H_2 \vdash E_{izq} = E_{der}$

Utilización de Hipótesis al Generalizar

En [B&M1979], cuando se trata el tema de la generalización, se señala que el método puede mejorarse si se logra conservar parte de la información con que se cuenta antes de generalizar. Parte de esta mejora ya está implementada, pues una de las características de un término es el tipo al que pertenece; esto es conservado al generalizar, pues la variable toma el tipo de la expresión que reemplaza.

No obstante, hay características de las expresiones que son más complejas, como la afirmación de que la nueva variable no puede tomar ciertos valores. Estas son las propiedades que podrían mantenerse por medio de hipótesis, como se ilustra en el siguiente ejemplo:

> Sea $\forall\ x, y, z \in NAT : x \neq z \vdash x * (y+1) \neq (y+1) * z$ la conjetura.
>
> Podemos generalizar la conjetura reemplazando $y+1$ por una variable nueva v. Nótese que la conjetura generalizada sería falsa si no agregamos como hipótesis que $v \neq 0$, lo cual se deduce de que $y+1 \neq 0$, $\forall\ y \in NAT$.
>
> La conjetura resultante sería, entonces, $\forall\ x, z, v \in NAT : x \neq z \wedge v \neq 0 \vdash x*v \neq v*z$.

Contraejemplos

En el *probador de teoremas* de ManTa las pruebas consisten en demostrar que la igualdad se cumple para todos los valores que pueden tomar las variables involucradas (como se dijo en *1.1.3.*, *Proposición de Conjeturas*, al proponer una conjetura se hace, implícitamente, cuantificación universal de todas las variables), pero cuando la conjetura no puede ser probada, no se sabe si se debe a que es falsa o, simplemente, que no pudo ser probada.

El problema de decidir si la conjetura es falsa resulta tanto o más difícil que el de decidir si es verdadera, en la mayoría de los casos. Sin embargo, hay ocasiones en las que, en medio de la demostración, se llega a la comparación sintáctica de dos términos básicos que en su forma normal no son iguales (i.e. *false = true* o *2 = 1*), de lo cual puede concluirse que la conjetura es falsa.

Esta conclusión es correcta si la comparación de los términos básicos ocurre fuera de los métodos de *prueba por casos* y *generalización*, ya que en estos métodos se hacen asunciones que pueden cambiar el significado de la conjetura.

Cuando se llega a la comparación de expresiones diferentes, sin variables, dentro del método de *inducción estructural*, es posible, de acuerdo a las sustituciones que se han hecho, determinar un contraejemplo; esto, debido a que se conocen los valores que se asignaron las variables de la conjetura inicial, con los cuales se obtuvieron las expresiones diferentes.

Tomemos, por ejemplo, la conjetura *PRED(SUC(X)) = SUC(PRED(X))*.

Demostración:

```
+---------------------- Conjetura no probada ---------------------+
PRED(SUC(X)) = SUC(PRED(X))

Regla al lado izquierdo PRED(SUC(N1)) = N1
X = SUC(PRED(X))

  Comienzo de la inducción
  Caso trivial
  Sustitución: X -> ZERO
      ZERO = SUC(PRED(ZERO))

      Regla al lado derecho: PRED(ZERO) = ZERO
      ZERO = SUC(ZERO)
  Falla en la prueba del caso trivial
  Fin de la inducción
Falla en la prueba
```

Para este caso se ha podido dar el contraejemplo *X = 0*, porque se halló que en el caso trivial de inducción, la reescritura condujo a comparar cero con uno, que se saben diferentes.

Pienso que es muy interesante de ManTa el que se haya logrado generalizar y sistematizar un proceso que ordinariamente depende de la habilidad del diseñador de software. No me resta más que agradecer a todos los que han contribuido con este proyecto, sin cuyos aportes no hubiera sido posible el presente trabajo.

4 BIBLIOGRAFIA

[Bun1983] BUNDY, Alan. *The Computer Modeling of Mathematical Reasoning*, Academic Press Inc., 1983.

[B&M1979] BOYER, Robert S. y MOORE, J. Strother. *A Computational Logic*, Academic Press, 1979.

[Car1993] CARDOSO, Rodrigo. *Verificación y Desarrollo de Programas*, Ediciones Uniandes. Bogotá, 1993.

[Car1995] CARDOSO, Rodrigo. Certified Implementation of Abstract Data Types, CLEI95, 1995.

[Car1996] CARDOSO, Rodrigo. Notas del curso 'Deducción Automática', Universidad de los Andes, 1996.

[Gri1994] GRIES, David y Schneider, Fred B. *A Logical Approach to Discrete Math*, Springer–Verlag, 1994.

[ItpInt] ITPCPP&B. *Induction Theorem Proving Challenge Problems and Benchmarks*. http://www.ai.mit.edu/people/dam/induction/benchmarks.html

[Man1974] MANNA, Zohar. *Mathematical Theory of Computation*, McGraw–Hill Inc., 1974.

[ManCod] MANTA. Código fuente de ManTa. http://agamenon.uniandes.edu.co/manta/

[ManDoc] TÁMARA, Vladimir. Documentación publicada de Manta. http://agamenon.uniandes.edu.co/manta/

[Muñ1992] MUÑOZ, César. *ManTa: Un Sistema para la Implantación Certificada y Automática de Tipos Abstractos de Datos*, Tesis de Magíster en Ingeniería de Sistemas, Universidad de los Andes, Santafé de Bogotá, 1992.

[Poi1907] POINCARÉ, Henri. *La Ciencia y La Hipótesis*, Traducción al español de Pedro M. González, Librería Gutenberg de José Ruiz, Madrid 1907.

[Tam1997] TÁMARA, Vladimir. *ManTa: Manejador de Tipos Abstractos de Datos*, Tesis de Pregrado en Ingeniería de Sistemas, Universidad de los Andes, Santafé de Bogotá, 1997.

[Tam1998] TÁMARA, Vladimir, *ManTa: Manejador de Tipos Abstractos de Datos*, Tesis de Magíster en Ingeniería de Sistemas, Universidad de los Andes, Santafé de Bogotá, 1998.

ANEXO A
MANUAL DE USO DEL PROBADOR DE TEOREMAS

En este anexo se explica cómo usar el módulo de *prueba de teoremas* de ManTa con la interfaz de DOS. Una explicación sobre el funcionamiento general del programa y de los otros módulos que lo componen, se encuentra en la página en Internet de ManTa (http://agamenon.uniandes.edu.co/manta/).

Presentación Inicial

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                    Bienvenido a ManTa 3.1 Beta 5

      M A N I P U L A D O R   D E   T I P O S   A B S T R A C T O S

                        UNIVERSIDAD DE LOS ANDES
                    GRUPO DE CONSTRUCCION DE SOFTWARE

                                1985 - 2001

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


 BOOL
_____________________________________________________________________________
MANTA
Tipos Teoremas Prototipos Salir ?
-----------------------------------------------------------------------------

Opcion:
```

Aquí se presentan los tres módulos que componen ManTa: *Especificación de Tipos*, *Prueba de Teoremas* y *Generación Automática de Prototipos*. Para escoger una opción se deben escribir las letras iniciales necesarias para diferenciarla de las demás.

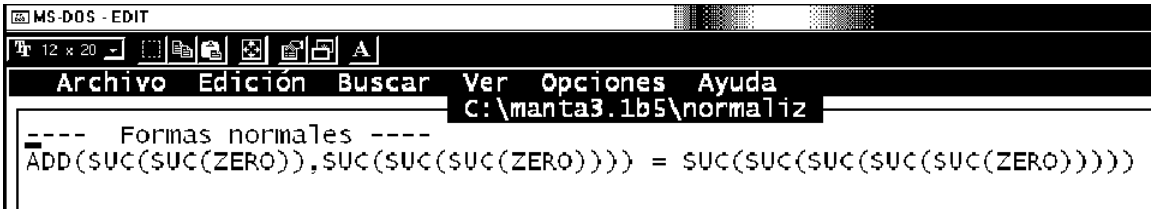
Prueba de Teoremas

```
Opcion:  te
 BOOL
_____________________________________________________________________________
TEOREMAS
Directorio Normalizar Pruebas Verificar_Representacion
Verificar_EQUAL Salvar_Probadas Salvar_No_Probadas Salir ?
-----------------------------------------------------------------------------

Opcion:  _
```

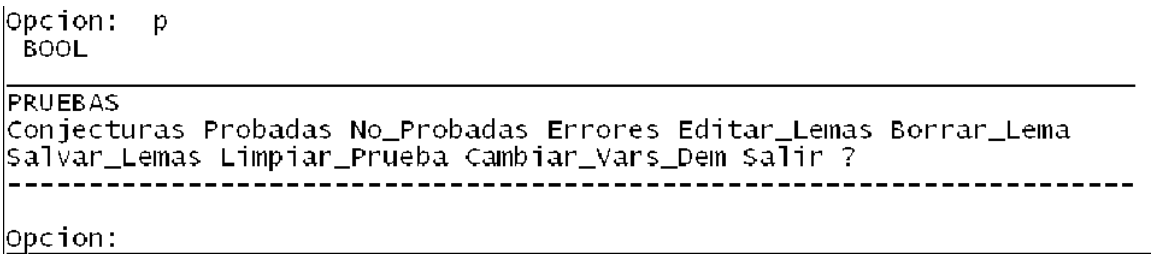
Al escribir 'te' se carga el menú de *teoremas*, en el que se presentan las opciones de la gráfica anterior:

Directorio: Escribe en pantalla el directorio en el que se encuentran las demostraciones.

Normalizar: Permite proponer una expresión para ser reducida a su forma normal.



Pruebas: Carga el menú que con las opciones para proponer conjeturas y controlar la demostración de las mismas.



Conjeturas: Permite plantear conjeturas para que sean demostradas por ManTa, en forma semi–automática.

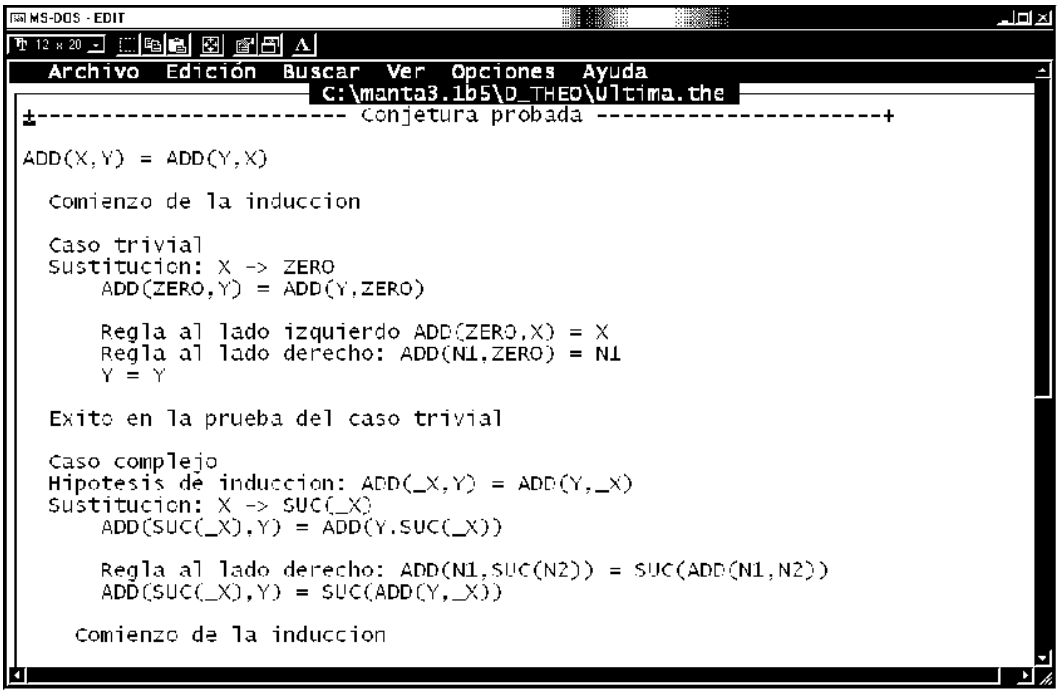
En medio de las pruebas se hacen preguntas al usuario, sobre cómo debe continuarse la prueba, cuando el método de *generalización* está entre el orden o cuando el modo de aplicación de lemas se pone en *manual*.

```
En la expresion
   ADD(ADD(W,Z),ADD(X,Y))
puede usarse el lema
   ADD(X,Y) = ADD(Y,X)
en la subexpresion
   ADD(W,Z)
quiere usarlo? (s/n): s
En la expresion
   ADD(ADD(Z,W),ADD(X,Y))
puede usarse el lema
   ADD(X,Y) = ADD(Y,X)
en la subexpresion
   ADD(X,Y)
quiere usarlo? (s/n):
```

Cuando una conjetura es probada con éxito, se le pregunta al usuario si desea guardarla y de qué tipo es la conjetura.

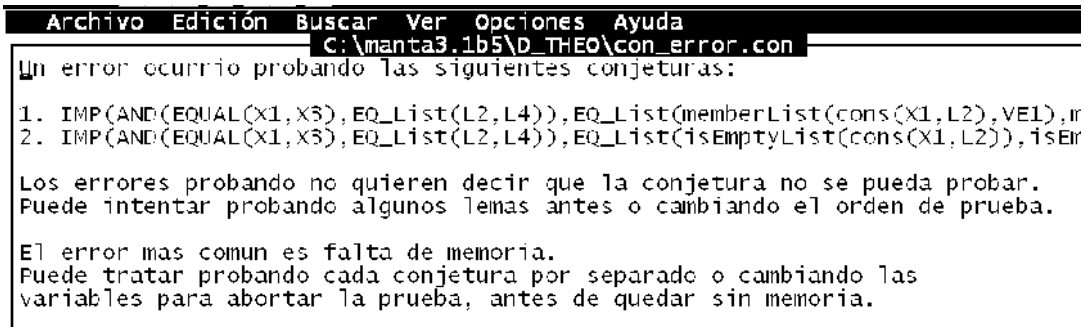
```
Opcion:  c

Quiere conservar el teorema
  IMP(EQUAL(X,Y),EQUAL(cons(X,L),cons(Y,L))) = TRUE
como  ema? (s/n):  s
Escoja uno de los siguientes tipos para el lema
IMP(EQUAL(X,Y),EQUAL(cons(X,L),cons(Y,L))) = TRUE:
  1. BOOL
  2. List
Tipo  :  2

Resultado
 Conjeturas probadas    : 1
 Conjeturas no probadas : 0
 Errores Probando : 0
```

Probadas: Edita la demostración de las conjeturas que pudieron ser probadas.



No_Probadas: Edita el intento de demostración de las conjeturas que no pudieron ser probadas, con el formato de la demostración anterior.

Errores: Edita en un archivo las conjeturas en cuyas demostraciones ocurrió un error de tiempo real.



Editar_Lemas: Edita un archivo con la lista de lemas que han sido cargados en el ambiente. Estos lemas son utilizados como reglas de reescritura, cuando el *lemas* se encuentra dentro de la lista de métodos.

```
Actualmente se han cargado los siguientes lemas:
  1. Lema del TAD BOOL: IMP(NOT(isEmptyStack(s)),EQ_BOOL(isEmptyStack(push(x,
  2. Lema del TAD BOOL: EQ_BOOL(EQ_BOOL(Q,P),R) = EQ_BOOL(Q,EQ_BOOL(P,R))
  3. Lema del TAD BOOL: IMP(OR(A,B),C) = AND(IMP(A,C),IMP(B,C))
  4. Lema del TAD BOOL: OR(EQ_NAT(SUC(X),SUC(ZERO)),GT(SUC(X),SUC(ZERO))) = T
  5. Lema del TAD NAT: ADD(X,Y) = ADD(Y,X)
  6. Lema del TAD NAT: ISZERO(X) = if(EQ_NAT(X,ZERO),TRUE,FALSE)
  7. Lema del TAD NAT: ADD(ADD(X,Y),ADD(W,Z)) = ADD(ADD(Z,W),ADD(Y,X))
  8. Lema del TAD NAT: ADD(SUC(X),ADD(Y,Z)) = ADD(X,ADD(SUC(Y),Z))
  9. Lema del TAD NAT: ADD(SUC(X),Y) = ADD(X,SUC(Y))
  10. Lema del TAD NAT: ADD(SUC(X),Y) = SUC(ADD(X,Y))
  11. Lema del TAD NAT: MULT(X,SUC(ZERO)) = MULT(SUC(ZERO),X)
  12. Lema del TAD NAT: IMP(EQ_NAT(X,ZERO),EQ_NAT(ADD(X,Y),Y)) = TRUE
  13. Lema del TAD NAT: LT(X,SUC(ZERO)) = ISZERO(X)
  14. Lema del TAD NAT: GE(SUC(X),SUC(ZERO)) = TRUE
  15. Lema del TAD NAT: SUBS(SUC(X),SUC(ZERO)) = X
  16. Lema del TAD NAT: ADD(X,ADD(SUC(ZERO),Y)) = ADD(ADD(SUC(Y),ZERO),X)
```

Borrar_Lema: Edita la lista de lemas, dando un número a cada uno y pregunta por el número del lema que se quiere borrar.

```
Actualmente se han cargado los siguientes lemas:
(Salgase del editor de archivos y escoja el numero del lema que quiere borrar)
  0. No borrar ningun lema
  1. Lema del TAD BOOL: IMP(NOT(isEmptyStack(s)),EQ_BOOL(isEmptyStack(push(x,s
  2. Lema del TAD BOOL: EQ_BOOL(EQ_BOOL(Q,P),R) = EQ_BOOL(Q,EQ_BOOL(P,R))
  3. Lema del TAD BOOL: IMP(OR(A,B),C) = AND(IMP(A,C),IMP(B,C))
  4. Lema del TAD BOOL: OR(EQ_NAT(SUC(X),SUC(ZERO)),GT(SUC(X),SUC(ZERO))) = TR
  5. Lema del TAD NAT: ADD(X,Y) = ADD(Y,X)
  6. Lema del TAD NAT: ISZERO(X) = if(EQ_NAT(X,ZERO),TRUE,FALSE)
  7. Lema del TAD NAT: ADD(ADD(X,Y),ADD(W,Z)) = ADD(ADD(Z,W),ADD(Y,X))
  8. Lema del TAD NAT: ADD(SUC(X),ADD(Y,Z)) = ADD(X,ADD(SUC(Y),Z))
  9. Lema del TAD NAT: ADD(SUC(X),Y) = ADD(X,SUC(Y))
  10. Lema del TAD NAT: ADD(SUC(X),Y) = SUC(ADD(X,Y))
  11. Lema del TAD NAT: MULT(X,SUC(ZERO)) = MULT(SUC(ZERO),X)
  12. Lema del TAD NAT: IMP(EQ_NAT(X,ZERO),EQ_NAT(ADD(X,Y),Y)) = TRUE
  13. Lema del TAD NAT: LT(X,SUC(ZERO)) = ISZERO(X)
  14. Lema del TAD NAT: GE(SUC(X),SUC(ZERO)) = TRUE
  15. Lema del TAD NAT: SUBS(SUC(X),SUC(ZERO)) = X
  16. Lema del TAD NAT: ADD(X,ADD(SUC(ZERO),Y)) = ADD(ADD(SUC(Y),ZERO),X)
```

Salvar_Lemas: Hace depuración de la lista de lemas del ambiente.

Limpiar_Prueba: Quita de la última demostración exitosa, los intentos de aplicación de métodos que no participaron de la prueba (intentos fallidos).

Cambiar_Vars_Dem: Muestra el menú de propiedades configurables para la prueba de conjeturas.

```
Opcion:  ca
 BOOL
_____________________________________________________________________
CAMBIAR VARIABLES DE DEMOSTRACION
Valores_Actuales Aplicacion_Lemas Aplicar_Teorema_Deduccion
Editar_Orden_Dem Orden_Predeterminado Veces_Lemas Variables_De_Parar
Salir ?
---------------------------------------------------------------------

Opcion:
```

Valores_Actuales: Muestra el modo actual de *aplicación de lemas*, si el *Teorema de la Deducción* se está aplicando y el número de veces que se está aplicando cada lema en las expresiones.

```
CAMBIAR VARIABLES DE DEMOSTRACION
Valores_Actuales Aplicacion_Lemas Aplicar_Teorema_Deduccion
Editar_Orden_Dem Orden_Predeterminado Veces_Lemas Variables_De_Parar
Salir ?
------------------------------------------------------------------------

Opcion:  va
Variables de Demostracion:
*Modo de aplicar lemas: Automatico
*Teorema de la Deduccion: prendido
*Numero de veces que se puede aplicar un lema: 1
```

Aplicación_Lemas: Permite cambiar entre modo manual y automático para la aplicación de lemas. En el modo manual, el probador pregunta si se quiere aplicar el lema o no.

Aplicar_Teorema_Deducción: Permite prender y apagar el uso del *Teorema de la Deducción*, que se aplica cuando la función principal de la expresión izquierda de la conjetura es *IMP* y el segundo parámetro de ésta es una igualdad. Cuando se aplica, una conjetura de la forma $IMP(\boldsymbol{h}, EQ_T(\boldsymbol{x}, \boldsymbol{y})) =_{BOOL} TRUE$ se transforma a la forma $\boldsymbol{h} \mid\!\!- \boldsymbol{x} =_T \boldsymbol{y}$.

Editar_Orden_Dem: Edita en un archivo el orden en el cual se intenta aplicar los métodos de prueba para las demostraciones. Cada método tiene asignado un código (que aparece entre paréntesis al frente de cada método), el cual es usado para insertar los métodos al querer cambiar el orden actual.

El orden de los métodos se puede cambiar escribiendo en el archivo los códigos de los métodos en el orden deseado, dentro del que se puede incluir más de una vez cada método.

```
  Archivo  Edición  Buscar  Ver  Opciones  Ayuda
                C:\manta3.1b5\D_THEO\ordeneng.dem
Actualmente ManTa esta aplicando los metodos de prueba en el siguiente orden:

METODOS
LEMAS(l)
HIPOTESIS(H)
INDUCCION(I)
CASOS(c)
SEMANTICA(s)
FIN_METODOS

Si quiere cambiarlo, escriba los codigos de los metodos en
el orden deseado (los codigos estan entre parentesis,
enfrente a los metodos que se estan usando).

NOTA: La correctitud de la prueba semantica depende de la
definicion del predicado EQUAL
```

NOTA: Cuando el método de hipótesis no se incluye en el orden, los métodos de *inducción* y *casos* pierden su efectividad, pues la hipótesis que se supone en éstos no es aplicada.

Orden_Predeterminado: Restablece el orden predeterminado de demostración:

Lemas(L)
Hipótesis(H)
Inducción(I)
Casos(C)
Generalización(G)
Semántica(S)

Veces_Lemas: Permite cambiar el número de veces que un mismo lema puede aplicarse sobre una expresión.

Variables_De_Parar: Despliega el menú con las variables que determina que una prueba se pueda seguir. Estas variables son utilizadas para detener la prueba cuando ya no parece llevar hacia una respuesta.

```
Opcion:  var
 BOOL
_____________________________________________________________________
VARIABLES DE PARA
Valores_Actuales Altura_Arbol_Sintactico Nodos_Arbol_Sintactico
Tamano_Expresion Salir ?
---------------------------------------------------------------------

Opcion: _
```

Valores_Actuales: Muestra los valores actuales de las variables de detención de demostraciones.

Altura_Arbol_Sintactico: Altura máxima para los árboles sintácticos de las expresiones de la conjetura.

Nodos_Arbol_Sintactico: Número máximo de nodos de los árboles sintácticos de las expresiones de la conjetura.

Tamano_Expresion: Longitud máxima de las expresiones de la conjetura, escritas en caracteres.

Verificar_Representación: Verifica que la representación de un *TAD* en otro cumple con los axiomas con que se especificó.

Verificar_EQUAL: Verifica que el predicado *EQUAL* de un *TAD* preserva las funciones selectoras; es decir, se verifica que cumpla con la regla de Leibnitz: cambiar una expresión por otra igual da el mismo resultado en todas las funciones selectoras del *TAD*.

```
Opcion:  verificar_e
 List
-------------------------------------------------------------------------
TADS_AMBIENTE
BOOL List NAT Salir
-------------------------------------------------------------------------

Opcion:  1
No hay memoria suficiente
No hay memoria suficiente
Error rewriting: No hay memoria suficienteNo hay memoria suficiente
No hay memoria suficiente
Fallo la verificacion del predicado EQUAL del TAD List


R e s u l t a d o
 Conjeturas probadas    :11
 Conjeturas no probadas : 2
 Errores Probando : 1
 List
-------------------------------------------------------------------------
TEOREMAS
Directorio Normalizar Pruebas Verificar_Representacion
Verificar_EQUAL Salvar_Probadas Salvar_No_Probadas Salir ?
-------------------------------------------------------------------------

Opcion:
```

Salvar_Probadas: Salva en un archivo texto la demostración de las últimas conjeturas que pudieron probarse.

Salvar_No_Probadas: Salva en un archivo texto la demostración de las últimas conjeturas que no pudieron probarse.

El proceso que se sigue para probar una conjetura incluye, entonces, determinar cuál es el orden apropiado para aplicar los diferentes métodos. También puede ser necesario demostrar algunos lemas intermedios, cuya aplicación puede ser controlada. Estas decisiones de personalización del *probador de teoremas* pueden determinarse por medio de pruebas, para lo cual se encuentran las opciones de edición de demostraciones e intentos de demostración.

Para dar una guía sobre las opciones de los diferentes menús, en todos ellos se encuentra una opción '?', que al pedirla despliega una descripción de las opciones que se ofrecen en el menú.

ANEXO B
MANUAL DE MANTENIMIENTO

ESTRUCTURAS DE DATOS

Para la implementación de las modificaciones al *probador de teoremas*, se hicieron algunos cambios sobre las estructuras de datos. A continuación se encuentra la definición de las estructuras de datos que se utilizan en el *probador de teoremas*.

***nodo_expresion*:**

```
typedef          struct      nodoexpresion {
                             cad12             nombre;
                             cad20             tipo;
                             cad12                   clase;
                             struct nodoargum  *argumentos;
                             struct nodoigualdad     *hipotesis;
                             struct nodolema   *lemas_usados;
                 } nodo_expresion;
```

`nombre`: Cadena de caracteres que identifica la función principal de la expresión.

`tipo`: Tipo al que pertenece la expresión. Corresponde al codominio de la función principal.

`clase`: La clase de la expresión se dice si es una *variable*, *inicial*, *constructora*, *selectora*, *if−then−else* o *equal*.

`argumentos`: Argumentos de la función principal.

`hipotesis`: Hipótesis que se suponen como reglas de reescritura para la expresión. Este es el contexto de la expresión.

`lemas_usados`: Lista de lemas que han sido utilizados como reglas de reescritura, durante la prueba. Este campo se usa para llevar control de ciclos en la aplicación de lemas y, así, asegurar terminación en la reescritura (ver *sección 2.1.2. Reescritura con Lemas*).

***nodo_igualdad*:**

```
typedef          struct      nodoigualdad {
                             cad20              tipo;
                             texpr              li_str,
                                                ld_str;
                             nodo_expresion     *lado_izq,
                                                *lado_der;
                             struct nodoigualdad     *sig;
                 } nodo_igualdad;
```

`tipo`: Nombre del TAD al que pertenece la igualdad. Este campo se usa para asociar los lemas con un TAD.

`li_str`: Cadena de caracteres que representa la expresión izquierda de la igualdad.

`ld_str`: Cadena de caracteres que representa la expresión derecha de la igualdad.

`lado_izq`: Árbol sintáctico de la expresión izquierda.

`lado_der`: Árbol sintáctico de la expresión derecha.

`sig`: Siguiente igualdad, cuando es parte de una lista.

ESTRUCTURA DE ARCHIVOS DE *kernel\d_theo*

Dentro de la estructura del código fuente se separó el kernel (núcleo de la aplicación), de la interfaz. Aquí se expone la estructura de archivos que compone el módulo de prueba de teoremas, que hace parte del kernel de la aplicación.

El módulo de prueba de teoremas se compone de los siguientes archivos de código fuente:

`enc_teo.h`: Encabezados de las funciones del módulo de teoremas.

`ana_teo.c`: Funciones para el análisis sintáctico de las conjeturas y para la manipulación de conjeturas, necesario para los métodos de prueba.

`man_teo.c`: Funciones para manejar el módulo de teoremas. Controla la interacción del usuario.

`rewrite.c`: Funciones para rescribir expresiones. Controla reescritura con axiomas, lemas e hipótesis.

`sistre.c`: Funciones de interfaz para la reescritura de expresiones.

`sustituc.c`: Funciones para la aplicación de sustituciones sobre expresiones.

`tabla.c`: Implementación del TAD tabla (lista de pares <selectora, ítem>).

Una descripción completa de la estructura de archivos de ManTa se encuentra en [Muñ1992].

RELACIÓN DE FUNCIONES DEL MÓDULO DE TEOREMAS

En el siguiente cuadro se presenta el mapa de dependencia de las funciones que llevan a cabo las principales tareas del *probador de teoremas*. En el mapa se incluyen las funciones para (1) Permitir al usuario proponer una conjetura para ser probada, (2) Limpiar la demostración de una conjetura probada y (3) Verificar el predicado *EQUAL* de un *TAD*.

Mapa de Funciones

(1)
```
void
edite_conjeturas (nodo_igualdad **conj, int manual)
```
Recibe una lista de igualdades e intenta probarlas.

```
edite_conjeturas
  analice_conjeturas
    analice_conj
      determine_tipo_teo
      es_identificador
      analice_if
      analice_equal
      es_inicial_tipo_dom
      analice_argum_generadora
      es_constructora_tipo_dom
      es_selectora_tipo_dom
      analice_expr_der
    es_error_esperaba
    tipos_var_consistentes
      listar_var_con_tipo
      dar_tipo_var
  probar_conjeturas_man
    leer_orden_prueba
    aplicaTdlD
    teoremaDeLaDeduccion
      agregar_hipotesis
    probar_igualdad_man
      fnlo
        rewriteOutermost
          rewriteOutermost_nth
            rewriteSubexpression
              changeSubexpressionByError
                changeExpressionBySubexpression
              rewriteSubexpressionIF
                changeExpressionBySubexpression
              rewriteSubexpressionEQUAL
              rewriteSubexpressionSelector
                rewriteSubexpressionWithRules
                  apl
                  changeSubexpressionByExpression
              busque_arch_tad
              rewriteSubexpressionWithAxioms
                rewriteExpressionWithOneAxiom
                  changeSubexpressionByExpression
            rewriteOutermost_nth
```

```
equal
seguir_demostración
  hayERROR
  seguir_expresion
numero_metodo
probar_lemas_man
  usarLemas
    rewriteOutermostWithLemmas
      rewriteOutermostWithLemmas_nth
        rewriteSubexpressionWithLemmas
          useable_lemma
            understandableExpression
          umg1
            match
              es_variable_exp
              equal
              loctab
              match
          refreshFreeVariable
            veces_esta_nombre
            nombre_en_hipot
            adjtab
            refreshFreeVariable
          allowed_lemma
          apl
            es_variable_exp
            loctab
          changeSubExpresionByExpression
          regUsedLemma
        rewriteOutermostWithLemmas_nth
probar_hipótesis_man
  iniciar_hipótesis
  usarHipotesis
    rewriteOutermostWithHypothesis
      rewriteOutermostWithHypothesis_nth
        rewriteSubexpressionWithHypothesis
          equal
          changeSubExpresionByExpression
        rewriteOutermostWithHypothesis_nth
probar_inducción_man
  buscar_variable_ind
  probar_inicial_man
    probar_igualdad_man
  probar_constructora_man
    probar_igualdad_man
probar_casos_man
  probar_igualdad_man
probar_generalización_man
  manejar_generalización
    edite_arch_generalizacion
      escriba_arch_generalización
        buscar_terminos_generalizables
        generalizar_expresion
          generalizar_expresion
  probar_igualdad_man
probar_semántica_man
  probar_igualdad_man
```

(2)

```
void
limpie_demost(int tipo)
```

Elimina los pasos que fallaron en la demostración de la última conjetura probada.

```
limpie_demost
  arreglar_demostración
```

```
   arreglar_historia
     arreglar_historia
```

(3)
`void manejar_verificacion_equal(void)`

Pregunta al usuario de cuál *TAD* quiere verificar el predicado *EQUAL*, genera conjeturas que garanticen la preservación de funciones e intenta probarlas (ver *2.5 Apoyo a la Verificación del Predicado EQUAL*).

```
manejar_verificación_equal
  pida_tad_ambiente
  verificar_equal
    arme_conjet_equal
    probar_conjeturas_man
```

ANEXO C
EJEMPLOS DE DEMOSTRACIONES HECHAS POR EL PROBADOR DE TEOREMAS DE MANTA

En este anexo se presentan algunos ejemplos de demostraciones hechas por el *probador de teoremas* de ManTa[1], para ilustrar su funcionamiento. Los ejemplos fueron hechos en la nueva versión, salvo algunos casos en los que se dice lo contrario.

1. En este ejemplo se muestra primero el tipo *ListNAT* (que fue creado automáticamente en ManTa, instanciando el *parámetro* del *TAD List* con *NAT* y luego se añadió la selectora *sumaLista*) y luego la demostración de una conjetura[2], en la que se usaron los métodos *Hipótesis*, *Lemas* e *Inducción* (en la demostración se quitaron los intentos fallidos con la opción 'Limpiar_Prueba' del menú de Pruebas).

```
ADT ListNAT [  ]

INITIALS
nil_ListNAT :                                   -> ListNAT

CONSTRUCTORS
cons_ListNAT: NAT * ListNAT                     -> ListNAT

SELECTORS
head_ListNAT   : ListNAT(i)                     -> NAT
tail_ListNAT   : ListNAT(i)                     -> ListNAT
length_ListNAT   : ListNAT(i)                   -> NAT
concat_ListNAT   : ListNAT(i) * ListNAT         -> ListNAT
memberList_ListNAT   : ListNAT(i) * NAT         -> BOOL
isEmptyList_ListNAT   : ListNAT(i)              -> BOOL
sumaLista   : ListNAT(i)                        -> NAT
EQ_ListNAT(c): ListNAT(i) * ListNAT(i)          -> BOOL

AXIOMS
head_ListNAT(nil_ListNAT) = ERROR
head_ListNAT(cons_ListNAT(X1,L2)) = X1

tail_ListNAT(nil_ListNAT) = ERROR
tail_ListNAT(cons_ListNAT(X1,L2)) = L2

length_ListNAT(nil_ListNAT) = ZERO
length_ListNAT(cons_ListNAT(X1,L2)) = SUC(length_ListNAT(L2))

concat_ListNAT(nil_ListNAT,L1) = L1
concat_ListNAT(cons_ListNAT(X1,L2),L3) =
                              cons_ListNAT(X1,concat_ListNAT(L2,L3))
```

[1] El formato del texto fue modificado para el documento, manteniendo la forma en que se presentan las demostraciones

[2] Esta prueba no se podía hacer en la versión anterior del probador de teoremas.

```
memberList_ListNAT(nil_ListNAT,X1) = FALSE
memberList_ListNAT(cons_ListNAT(X1,L2),X3) = if EQ_NAT(X1,X3)
                                               then TRUE
                                               else
                                               memberList_ListNAT(L2,X3)
                                             fi

isEmptyList_ListNAT(nil_ListNAT) = TRUE
isEmptyList_ListNAT(cons_ListNAT(X1,L2)) = FALSE

sumaLista(nil_ListNAT) = ZERO
sumaLista(cons_ListNAT(N1,L2)) = ADD(N1,sumaLista(L2))

EQ_ListNAT(nil_ListNAT,nil_ListNAT) = TRUE
EQ_ListNAT(nil_ListNAT,cons_ListNAT(X1,L2)) = FALSE
EQ_ListNAT(cons_ListNAT(X1,L2),nil_ListNAT) = FALSE
EQ_ListNAT(cons_ListNAT(X1,L2),cons_ListNAT(X3,L4)) =
                                    AND(EQ_NAT(X1,X3),EQ_ListNAT(L2,L4))

+------------------------ Conjetura probada ----------------------+
sumaLista(cons_ListNAT(x,cons_ListNAT(y,l))) =
sumaLista(cons_ListNAT(y,cons_ListNAT(x,l)))

Regla al lado izquierdo: sumaLista(cons_ListNAT(N1,L2)) =
                          ADD(N1,sumaLista(L2))
Regla al lado derecho: sumaLista(cons_ListNAT(N1,L2)) =
                         ADD(N1,sumaLista(L2))
ADD(x,sumaLista(cons_ListNAT(y,l))) = ADD(y,sumaLista(cons_ListNAT(x,l)))

Regla al lado izquierdo: sumaLista(cons_ListNAT(N1,L2)) =
                          ADD(N1,sumaLista(L2))
Regla al lado derecho: sumaLista(cons_ListNAT(N1,L2)) =
                         ADD(N1,sumaLista(L2))
ADD(x,ADD(y,sumaLista(l))) = ADD(y,ADD(x,sumaLista(l)))

Lema al lado izquierdo: ADD(X,Y) = ADD(Y,X)
ADD(ADD(y,sumaLista(l)),x) = ADD(y,ADD(x,sumaLista(l)))

Lema al lado izquierdo: ADD(X,Y) = ADD(Y,X)
ADD(ADD(sumaLista(l),y),x) = ADD(y,ADD(x,sumaLista(l)))

Lema al lado derecho: ADD(X,Y) = ADD(Y,X)
ADD(ADD(sumaLista(l),y),x) = ADD(ADD(x,sumaLista(l)),y)

Lema al lado derecho: ADD(X,Y) = ADD(Y,X)
ADD(ADD(sumaLista(l),y),x) = ADD(ADD(sumaLista(l),x),y)

  Comienzo de la induccion
  Caso trivial
  Sustitucion: x -> ZERO
      ADD(ADD(sumaLista(l),y),ZERO) = ADD(ADD(sumaLista(l),ZERO),y)

      Regla al lado izquierdo ADD(N1,ZERO) = N1
      Regla al lado derecho: ADD(N1,ZERO) = N1
      ADD(sumaLista(l),y) = ADD(sumaLista(l),y)
  Exito en la prueba del caso trivial

  Caso complejo
  Hipotesis de induccion: ADD(ADD(sumaLista(l),y),_x) =
                          ADD(ADD(sumaLista(l),_x),y)
  Sustitucion: x -> SUC(_x)
      ADD(ADD(sumaLista(l),y),SUC(_x)) = ADD(ADD(sumaLista(l),SUC(_x)),y)

      Regla al lado izquierdo ADD(N1,SUC(N2)) = SUC(ADD(N1,N2))
      Regla al lado derecho: ADD(N1,SUC(N2)) = SUC(ADD(N1,N2))
```

```
      SUC(ADD(ADD(sumaLista(l),y),_x)) = ADD(SUC(ADD(sumaLista(l),_x)),y)

      Hipotesis en el lado izquierdo: ADD(ADD(sumaLista(l),y),_x) =
                                      ADD(ADD(sumaLista(l),_x),y)
      SUC(ADD(ADD(sumaLista(l),_x),y)) = ADD(SUC(ADD(sumaLista(l),_x)),y)

      Lema al lado izquierdo: SUC(ADD(X,Y)) = ADD(SUC(X),Y)
      ADD(SUC(ADD(sumaLista(l),_x)),y) = ADD(SUC(ADD(sumaLista(l),_x)),y)
  Exito en la prueba del caso complejo
  Fin de la induccion
Exito en la prueba
```

2. Este ejemplo ilustra la reescritura con el régimen *Leftmost − Outtermost*.

```
+------------------------ Conjetura probada ----------------------+
SUBS(SUC(X),SUC(ZERO)) = PRED(SUC(X))

Regla al lado izquierdo: SUBS(N1,SUC(N2)) = PRED(SUBS(N1,N2))
Regla al lado derecho: PRED(SUC(N1)) = N1
PRED(SUBS(SUC(X),ZERO)) = X

Regla al lado izquierdo: SUBS(N1,ZERO) = N1
PRED(SUC(X)) = X

Regla al lado izquierdo: PRED(SUC(N1)) = N1
X = X
Éxito en la prueba
```

3. Este es un ejemplo de una demostración hecha en la versión anterior del *probador de teoremas*, en la que las dos expresiones reducen a *ERROR*, aunque son de diferentes tipos.

```
+------------------------ Proved conjecture ----------------------+
head(nil) = top(emptyStack)

Rule to left side: head(nil) = error
Rule to right side: top(emptyStack) = error
ERROR = ERROR
Successful proof
```

4. En este ejemplo, también de la versión anterior, se ilustra el problema que existía al no revisar si las variables introducidas en *inducción* ya existían en la conjetura.

```
+------------------------ Proved conjecture ----------------------+
concat(x,y) = concat(y,x)

  Beginning of induction
  Base case
  Substitution: x -> nil
      concat(nil,y) = concat(y,nil)

      Rule to left side: concat(nil,L1) = L1
      y = concat(y,nil)

    Beginning of induction
    Base case
    Substitution: y -> nil
```

```
        nil = concat(nil,nil)

        Rule to right side: concat(nil,L1) = L1
        nil = nil
    Base case proof successful

    Induction step
    Induction hypothesis : concat(_y,nil) = _y
    Substitution: y -> cons(X_1,_y)
        cons(X_1,_y) = concat(cons(X_1,_y),nil)

        Rule to right side: concat(cons(X1,L2),L3) = cons(X1,concat(L2,L3))
        cons(X_1,_y) = cons(X_1,concat(_y,nil))

        Rule to right side: concat(_y,nil) = _y
        cons(X_1,_y) = cons(X_1,_y)

    Induction step proof successful
    End of induction
  Base case proof successful

  Induction step
  Induction hypothesis : concat(_x,y) = concat(y,_x)
  Substitution: x -> cons(X_1,_x)
      concat(cons(X_1,_x),y) = concat(y,cons(X_1,_x))

      Rule to left side: concat(cons(X1,L2),L3) = cons(X1,concat(L2,L3))
      cons(X_1,concat(_x,y)) = concat(y,cons(X_1,_x))

      Rule to left side: concat(_x,y) = concat(y,_x)
      cons(X_1,concat(y,_x)) = concat(y,cons(X_1,_x))

    Beginning of induction
    Base case
    Substitution: y -> nil
        cons(X_1,concat(nil,_x)) = concat(nil,cons(X_1,_x))

        Rule to left side: concat(nil,L1) = L1
        Rule to right side: concat(nil,L1) = L1
        cons(X_1,_x) = cons(X_1,_x)

    Base case proof successful

    Induction step
    Induction hypothesis : concat(_y,cons(X_1,_x)) = cons(X_1,concat(_y,_x))
    Substitution: y -> cons(X_1,_y)
       cons(X_1,concat(cons(X_1,_y),_x)) = concat(cons(X_1,_y),cons(X_1,_x))

       Rule to left side: concat(cons(X1,L2),L3) = cons(X1,concat(L2,L3))
       Rule to right side:    concat(cons(X1,L2),L3) =
                              cons(X1,concat(L2,L3))
       cons(X_1,cons(X_1,concat(_y,_x))) =
       cons(X_1,concat(_y,cons(X_1,_x)))

       Rule to right side:    concat(_y,cons(X_1,_x)) =
                              cons(X_1,concat(_y,_x))
       cons(X_1,cons(X_1,concat(_y,_x))) =
       cons(X_1,cons(X_1,concat(_y,_x)))
    Induction step proof successful
    End of induction
  Induction step proof successful
  End of induction
Successful proof
```

Nótese que al construir la lista por la que sustituiría **y**, en el paso inductivo, se asumió que se la agrega **X_1** a la lista **_y**; es decir, el mismo elemento que se le agregó a la lista **_x**, al construir la sustitución de **x**.

5. En esta prueba se usaron los métodos *Semántica* e *Hipótesis* en ese orden

```
+------------------------ Conjetura probada ----------------------+
Teorema de la Deducción
Hipótesis: X = Y
Conjetura: cons(X,L) = cons(Y,L)

cons(X,L) = cons(Y,L)
  Prueba semántica
      EQUAL(cons(X,L),cons(Y,L)) = TRUE

      Regla al lado izquierdo: EQUAL(X,Y) = EQ_List(X,Y)
      EQ_List(cons(X,L),cons(Y,L)) = TRUE

      Regla al lado izquierdo: EQ_List(cons(X1,L2),cons(X3,L4)) =
                               AND(equal(X1,X3),EQ_List(L2,L4))
      AND(EQUAL(X,Y),EQ_List(L,L)) = TRUE

      Regla al lado izquierdo: EQUAL(X,Y) = EQ_X(X,Y)
      AND(EQ_X(X,Y),EQ_List(L,L)) = TRUE

      Regla al lado izquierdo: EQ_List(X,X) = TRUE
      AND(EQ_X(X,Y),TRUE) = TRUE

      Regla al lado izquierdo: AND(B1,TRUE) = B1
      EQ_X(X,Y) = TRUE

      Hipótesis en el lado izquierdo: X = Y
      EQ_X(Y,Y) = TRUE

      Regla al lado izquierdo: EQ_X(X,X) = TRUE
      TRUE = TRUE
  Éxito en la prueba semántica
Éxito en la prueba
```

6. En el siguiente ejemplo se uso únicamente *reescritura con lemas*.

```
+------------------------ Conjetura probada ----------------------+
ADD(ADD(X,Y),ADD(W,Z)) = ADD(ADD(Z,W),ADD(Y,X))

Lema al lado izquierdo: ADD(X,Y) = ADD(Y,X)
ADD(ADD(W,Z),ADD(X,Y)) = ADD(ADD(Z,W),ADD(Y,X))

Lema al lado izquierdo: ADD(X,Y) = ADD(Y,X)
ADD(ADD(Z,W),ADD(X,Y)) = ADD(ADD(Z,W),ADD(Y,X))

Lema al lado izquierdo: ADD(X,Y) = ADD(Y,X)
ADD(ADD(Z,W),ADD(Y,X)) = ADD(ADD(Z,W),ADD(Y,X))
Éxito en la prueba
```

7. La siguiente es la demostración de $X = 0 \Rightarrow X + Y = Y$, que escrito con la sintaxis de ManTa queda *IMP(EQ_NAT(X, ZERO), EQ_NAT(ADD(X, Y), Y)) = TRUE*.

```
+------------------------ Conjetura probada ----------------------+
Teorema de la Deducción
Hipótesis: X = ZERO
Conjetura: ADD(X, Y) = Y
  ADD(X, Y) = Y
  Hipótesis en el lado izquierdo: X = ZERO
  ADD(ZERO, Y) = Y
  Regla al lado izquierdo ADD(ZERO, X) = X
  Y = Y
Éxito en la prueba
```

Este ejemplo ilustra la utilidad de aplicar el *Teorema de la Deducción* y de usar hipótesis en la demostración, pues de no haberse aplicado hubiera sido necesario hacer inducción sobre *X* e *Y*.

8. Para la siguiente demostración se usaron los métodos *Hipótesis*, *Inducción* y *Generalización* en ese orden:

```
+---------------------- Conjetura probada -------------------+
LT(X, SUC(ZERO)) = ISZERO(X)
  Comienzo de la inducción
  Caso trivial
  Sustitución: X -> ZERO
      LT(ZERO, SUC(ZERO)) = ISZERO(ZERO)
      Regla al lado izquierdo LT(ZERO, SUC(N1)) = TRUE
      Regla al lado derecho: ISZERO(ZERO) = TRUE
      TRUE = TRUE
  Éxito en la prueba del caso trivial
  Caso complejo
  Hipótesis de inducción: LT(_X, SUC(ZERO)) = ISZERO(_X)
  Sustitución: X -> SUC(_X)
      LT(SUC(_X),SUC(ZERO)) = ISZERO(SUC(_X))
      Regla al lado izquierdo LT(SUC(N1),SUC(N2)) = LT(N1,N2)
      Regla al lado derecho: ISZERO(SUC(N1)) = FALSE
      LT(_X, ZERO) = FALSE
    Conjetura generalizada:
    Sustitución: LT(_X, ZERO) -> LT(NV1,ZERO)
        LT(NV1,ZERO) = FALSE
      Comienzo de la inducción
      Caso trivial
      Sustitución: NV1 -> ZERO
          LT(ZERO, ZERO) = FALSE
          Regla al lado izquierdo LT(ZERO, ZERO) = FALSE
          FALSE = FALSE
      Éxito en la prueba del caso trivial
      Caso complejo
      Hipótesis de inducción: LT(_NV1,ZERO) = FALSE
      Sustitución: NV1 -> SUC(_NV1)
          LT(SUC(_NV1),ZERO) = FALSE
          Regla al lado izquierdo LT(SUC(N1),ZERO) = FALSE
          FALSE = FALSE
      Éxito en la prueba del caso complejo
      Fin de la inducción
    Prueba exitosa de conjetura generalizada
  Éxito en la prueba del caso complejo
  Fin de la inducción
Éxito en la prueba
```

9. En esta prueba se usaron los métodos *Hipótesis, Lemas e Inducción*, en ese orden. Para que la prueba fuera posible se probaron, previamente, con ManTa, los lemas que aparecen dentro de la demostración.

```
+------------------------ Conjetura probada ----------------------+
MULT(X, Y) = MULT(Y, X)

  Comienzo de la inducción
  Caso trivial
  Sustitución: X -> ZERO
      MULT(ZERO, Y) = MULT(Y, ZERO)

      Regla al lado izquierdo MULT(ZERO, N1) = ZERO
      Regla al lado derecho: MULT(X, ZERO) = ZERO
      ZERO = ZERO
  Éxito en la prueba del caso trivial

  Caso complejo
  Hipótesis de inducción: MULT(_X, Y) = MULT(Y, _X)
  Sustitución: X -> SUC(_X)
      MULT(SUC(_X), Y) = MULT(Y, SUC(_X))

      Regla al lado izquierdo MULT(SUC(N1), N2) = ADD(MULT(N1,N2), N2)
      ADD(MULT(_X, Y), Y) = MULT(Y, SUC(_X))

      Hipótesis en el lado izquierdo: MULT(_X, Y) = MULT(Y, _X)
      ADD(MULT(Y, _X), Y) = MULT(Y, SUC(_X))

      Lema al lado izquierdo: ADD(X, Y) = ADD(Y, X)
      ADD(Y, MULT(Y, _X)) = MULT(Y, SUC(_X))

    Comienzo de la inducción
    Caso trivial
    Sustitución: Y -> ZERO
        ADD(ZERO, MULT(ZERO, _X)) = MULT(ZERO, SUC(_X))

        Regla al lado izquierdo ADD(ZERO, X) = X
        Regla al lado derecho: MULT(ZERO, N1) = ZERO
        MULT(ZERO, _X) = ZERO

        Regla al lado izquierdo MULT(ZERO, N1) = ZERO
        ZERO = ZERO
    Éxito en la prueba del caso trivial

    Caso complejo
    Hipótesis de inducción: MULT(_Y, SUC(_X)) = ADD(_Y, MULT(_Y, _X))
    Sustitución: Y -> SUC(_Y)
        ADD(SUC(_Y), MULT(SUC(_Y), _X)) = MULT(SUC(_Y), SUC(_X))

        Regla al lado izquierdo MULT(SUC(N1), N2) = ADD(MULT(N1,N2), N2)
        Regla al lado derecho: MULT(SUC(N1), N2) = ADD(MULT(N1, N2), N2)
        ADD(SUC(_Y), ADD(MULT(_Y, _X), _X)) = ADD(MULT(_Y, SUC(_X)), SUC(_X))

        Lema al lado izquierdo: ADD(X, ADD(Y, Z)) = ADD(ADD(X, Y), Z)
        ADD(ADD(SUC(_Y), MULT(_Y, _X)), _X) = ADD(MULT(_Y, SUC(_X)), SUC(_X))

        Lema al lado izquierdo: ADD(SUC(X), Y) = ADD(X, SUC(Y))
        ADD(ADD(_Y, SUC(MULT(_Y, _X))), _X) = ADD(MULT(_Y, SUC(_X)), SUC(_X))

        Lema al lado izquierdo: ADD(X, Y) = ADD(Y, X)
        ADD(ADD(SUC(MULT(_Y, _X)), _Y), _X) = ADD(MULT(_Y, SUC(_X)), SUC(_X))

        Lema al lado izquierdo: ADD(SUC(X), Y) = SUC(ADD(X, Y))
        ADD(SUC(ADD(MULT(_Y, _X), _Y)), _X) = ADD(MULT(_Y, SUC(_X)), SUC(_X))

        Lema al lado izquierdo: ADD(SUC(X), Y) = ADD(X, SUC(Y))
        ADD(ADD(MULT(_Y, _X), _Y), SUC(_X)) = ADD(MULT(_Y, SUC(_X)), SUC(_X))

        Lema al lado izquierdo: ADD(X, Y) = ADD(Y, X)
        ADD(ADD(_Y, MULT(_Y, _X)), SUC(_X)) = ADD(MULT(_Y, SUC(_X)), SUC(_X))
```

```
        Regla al lado derecho: ADD(N1, SUC(N2)) = SUC(ADD(N1, N2))
        ADD(ADD(_Y, MULT(_Y, _X)), SUC(_X)) = SUC(ADD(MULT(_Y, SUC(_X)), _X))

        Hipótesis en el lado derecho: MULT(_Y, SUC(_X)) = ADD(_Y, MULT(_Y,_X))
        ADD(ADD(_Y, MULT(_Y,_X)),SUC(_X)) = SUC(ADD(ADD(_Y, MULT(_Y,_X)),_X))

        Lema al lado derecho: SUC(ADD(X, Y)) = ADD(SUC(X), Y)
        ADD(ADD(_Y, MULT(_Y,_X)), SUC(_X)) = ADD(SUC(ADD(_Y, MULT(_Y,_X))),_X)

        Lema al lado derecho: ADD(SUC(X), Y) = ADD(X, SUC(Y))
        ADD(ADD(_Y, MULT(_Y,_X)),SUC(_X)) = ADD(ADD(_Y, MULT(_Y,_X)),SUC(_X))
    Éxito en la prueba del caso complejo
    Fin de la inducción

  Éxito en la prueba del caso complejo
  Fin de la inducción

Éxito en la prueba
```

En este ejemplo se ilustra la utilidad de los lemas, sin los cuales no se podría probar. También se puede ver el control que se hace para no aplicar más veces de las debidas un mismo lema, pues casos como el de *ADD(X, Y) = ADD(Y, X)* harían que no se pudiera avanzar.

10. Este ejemplo contiene las demostraciones a las conjeturas planteadas para la verificación del predicado *EQUAL* (ver *2.5 Apoyo a la Verificación del Predicado EQUAL*) del ***TAD List***, cuya especificación se encuentra en el *Cuadro 1.1*.

```
+------------------------ Conjetura probada ----------------------+
Teorema de la Deducción
Hipótesis: X1 = X3
           L2 = L4
Conjetura: head(cons(X1,L2)) = head(cons(X3,L4))

head(cons(X1,L2)) = head(cons(X3,L4))

Regla al lado izquierdo head(cons(X1,L2)) = X1
Regla al lado derecho: head(cons(X1,L2)) = X1
X1 = X3

Hipótesis en el lado izquierdo: X1 = X3
X3 = X3
Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
Teorema de la Deducción
Hipótesis: X1 = X3
           L2 = L4
Conjetura: tail(cons(X1,L2)) = tail(cons(X3,L4))

      tail(cons(X1,L2)) = tail(cons(X3,L4))

      Regla al lado izquierdo tail(cons(X1,L2)) = L2
      Regla al lado derecho: tail(cons(X1,L2)) = L2
      L2 = L4

      Hipotesis en el lado izquierdo: L2 = L4
      L4 = L4
Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
length(nil) = length(nil)

      Regla al lado izquierdo length(nil) = ZERO
      Regla al lado derecho: length(nil) = ZERO
```

```
      ZERO = ZERO

Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
Teorema de la Deducción
Hipótesis: X1 = X3
           L2 = L4
Conjetura: length(cons(X1,L2)) = length(cons(X3,L4))

      length(cons(X1,L2)) = length(cons(X3,L4))

      Regla al lado izquierdo: length(cons(X1,L2)) = SUC(length(L2))
      Regla al lado derecho: length(cons(X1,L2)) = SUC(length(L2))
      SUC(length(L2)) = SUC(length(L4))

      Hipótesis en el lado izquierdo: L2 = L4
      SUC(length(L4)) = SUC(length(L4))

Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
concat(nil,VE1) = concat(nil,VE1)

      Regla al lado izquierdo concat(nil,L1) = L1
      Regla al lado derecho: concat(nil,L1) = L1
      VE1 = VE1

Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
concat(VE1,nil) = concat(VE1,nil)

Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
Teorema de la Deducción
Hipótesis: X1 = X3
           L2 = L4
Conjetura: concat(cons(X1,L2),VE1) = concat(cons(X3,L4),VE1)

      concat(cons(X1,L2),VE1) = concat(cons(X3,L4),VE1)

      Regla al lado izquierdo: concat(cons(X1,L2),L3) = cons(X1,concat(L2,L3))
      Regla al lado derecho: concat(cons(X1,L2),L3) = cons(X1,concat(L2,L3))
      cons(X1,concat(L2,VE1)) = cons(X3,concat(L4,VE1))

      Hipótesis en el lado izquierdo: X1 = X3
      cons(X3,concat(L2,VE1)) = cons(X3,concat(L4,VE1))

      Hipótesis en el lado izquierdo: L2 = L4
      cons(X3,concat(L4,VE1)) = cons(X3,concat(L4,VE1))

Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
Teorema de la Deducción
Hipótesis: X1 = X3
           L2 = L4
Conjetura: concat(VE1,cons(X1,L2)) = concat(VE1,cons(X3,L4))

      concat(VE1,cons(X1,L2)) = concat(VE1,cons(X3,L4))

      Hipótesis en el lado izquierdo: X1 = X3
      concat(VE1,cons(X3,L2)) = concat(VE1,cons(X3,L4))

      Hipótesis en el lado izquierdo: L2 = L4
      concat(VE1,cons(X3,L4)) = concat(VE1,cons(X3,L4))

Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
```

```
memberList(nil,VE1) = memberList(nil,VE1)

      Regla al lado izquierdo: memberList(nil,X1) = FALSE
      Regla al lado derecho: memberList(nil,X1) = FALSE
      FALSE = FALSE
Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
isEmptyList(nil) = isEmptyList(nil)

      Regla al lado izquierdo: isEmptyList(nil) = TRUE
      Regla al lado derecho: isEmptyList(nil) = TRUE
      TRUE = TRUE
Éxito en la prueba
+------------------------ Conjetura probada ----------------------+
Teorema de la Deducción
Hipótesis: X1 = X3
           L2 = L4
Conjetura: isEmptyList(cons(X1,L2)) = isEmptyList(cons(X3,L4))

      isEmptyList(cons(X1,L2)) = isEmptyList(cons(X3,L4))

      Regla al lado izquierdo: isEmptyList(cons(X1,L2)) = FALSE
      Regla al lado derecho: isEmptyList(cons(X1,L2)) = FALSE
      FALSE = FALSE

Éxito en la prueba
_____________________________________________________________________
+----------------------- Conjetura no probada ----------------------+
head(nil) = head(nil)

Regla al lado izquierdo: head(nil) = error
Regla al lado derecho: head(nil) = error
ERROR = ERROR
+--- Esta forma de probar fue detenida porque porque uno de los lados era ERROR ---+
Falla en la prueba
+----------------------- Conjetura no probada ----------------------+
tail(nil) = tail(nil)

      Regla al lado izquierdo: tail(nil) = error
      Regla al lado derecho: tail(nil) = error
      ERROR = ERROR
+--- Esta forma de probar fue detenida porque porque uno de los lados era ERROR ---+
Falla en la prueba
_____________________________________________________________________
Un error ocurrió probando las siguientes conjeturas:

1.    IMP(AND(EQUAL(X1,X3),EQ_List(L2,L4)),EQ_List(memberList(cons(X1,L2),
VE1), memberList(cons(X3,L4),VE1))) = TRUE

2.  IMP(AND(EQUAL(X1,X3),EQ_List(L2,L4)),EQ_List(isEmptyList(cons(X1,L2)),
isEmptyList(cons(X3,L4)))) = TRUE

Los errores probando no quieren decir que la conjetura no se pueda
probar.
Puede intentar probando algunos lemas antes o cambiando el orden de
prueba.

El error mas común es falta de memoria.
Puede tratar probando cada conjetura por separado o cambiando las
variables para abortar la prueba, antes de quedar sin memoria.
```

11. La siguiente es la demostración de una conjetura falsa, que es detenida por llegarse a *ERROR* en la reducción de las expresiones.

```
+---------------------- Conjetura no probada ----------------------+
DIV(X,ZERO) = DIV(X,ZERO)
Regla al lado izquierdo: DIV(N1,N2) =
      if(ISZERO(N2),error,if(GE(N1,N2),SUC(DIV(SUBS(N1,N2),N2)),ZERO))
Regla al lado derecho: DIV(N1,N2) =
      if(ISZERO(N2),error,if(GE(N1,N2),SUC(DIV(SUBS(N1,N2),N2)),ZERO))
if(ISZERO(ZERO),ERROR,if(GE(X,ZERO),SUC(DIV(SUBS(X,ZERO),ZERO)),ZERO))
=
if(ISZERO(ZERO),ERROR,if(GE(X,ZERO),SUC(DIV(SUBS(X,ZERO),ZERO)),ZERO))

Regla al lado izquierdo: ISZERO(ZERO) = TRUE
Regla al lado derecho: ISZERO(ZERO) = TRUE
if(TRUE,ERROR,if(GE(X,ZERO),SUC(DIV(SUBS(X,ZERO),ZERO)),ZERO))
=
if(TRUE,ERROR,if(GE(X,ZERO),SUC(DIV(SUBS(X,ZERO),ZERO)),ZERO))

Regla al lado izquierdo: if TRUE then X else Y fi = X
Regla al lado derecho: if TRUE then X else Y fi = X
ERROR = ERROR
+---  Esta forma de probar fue detenida
      porque uno de los lados era ERROR ---+
Falla en la prueba
```

12. En este ejemplo también se trata de una conjetura falsa. Aquí se ilustra la regla de reescritura para funciones estrictas, cuando hay un parámetro indefinido.

```
+---------------------- Conjetura no probada ----------------------+
cons(head(nil),L) = cons(head(nil),L)
      Regla al lado izquierdo: head(nil) = error
      Regla al lado derecho: head(nil) = error
      cons(ERROR, L) = cons(ERROR, L)

      Regla al lado izquierdo: f(..., ERROR, ...) = ERROR
      Regla al lado derecho: f(..., ERROR, ...) = ERROR
      ERROR = ERROR
+---  Esta forma de probar fue detenida
      porque uno de los lados era ERROR ---+
Falla en la prueba
```

13. En este ejemplo se ilustra un problema que existe con la regla *leftmost – Outtermost*, sobre el manejo del elemento *ERROR*.

```
+----------------------- Conjetura probada ----------------------+
isEmptyStack(push(top(emptyStack), s1)) = isEmptyStack(push(x, s2))

Regla al lado izquierdo: isEmptyStack(push(X1, S2)) = FALSE
Regla al lado derecho: isEmptyStack(push(X1, S2)) = FALSE
FALSE = FALSE

Éxito en la prueba
```

Nótese que top(emptyStack) reduce a *ERROR*, luego toda la expresión izquierda ha debido reducir a *ERROR*.

ANEXO D
DEMOSTRACIONES DE TEOREMAS USADOS EN ESTE DOCUMENTO

En este anexo se presentan las demostraciones a los metateoremas que se han usado a lo largo del documento. Para las demostraciones se usaron teoremas de [Gri1994], cuyas demostraciones se encuentran en dicho libro y fueron elaboradas con base en un conjunto correcto y completo de axiomas.

Metateorema 1. [*Sección 2.1.3.*]

f(x, H ⇒ C, y) p f(x, H ⇒ C, y)

Demostración:

Dado que *H* es el primer parámetro de la función *implicación*, podemos deducir que se trata de una expresión de tipo BOOL. Por tal razón, podemos mirar los casos H = *true* y H = *false*.

Miremos, entonces, que ocurre con la expresión $H \Rightarrow C$ en cada uno de estos casos:

Caso H = *true*:

$H \Rightarrow C$

$=$ $\langle$ *en el caso H = true* $\rangle$

$true \Rightarrow C$

$=$ $\langle$ $true \Rightarrow p \equiv p$ $\rangle$

C

caso H = *false*:

$H \Rightarrow C$

$=$ $\langle$ *en el caso H = false* $\rangle$

$true \Rightarrow C$

$=$ $\langle$ $false \Rightarrow p \equiv true$ $\rangle$

$true$

Tenemos, por tanto, que si *C* reduce a *true*, el resultado $H \Rightarrow C$ es el mismo para ambos casos.

Por esta razón, decimos que si $f(\boldsymbol{x}, H \Rightarrow C, \boldsymbol{y})$ puede probarse, entonces también puede probarse $f(\boldsymbol{x}, H \Rightarrow C, \boldsymbol{y})$. No se afirmar el reciproco, pero con una de las direcciones basta para la aplicación de hipótesis en el *probador de teoremas*.

Metateorema 2. [*Sección 2.3.1*]

$\boldsymbol{E, E \vdash\!\!\!- E}$

Demostración:

E

= ⟨ *Teorema de Shannon* $(E^z_p \equiv (p \wedge E^z_{true}) \vee (\neg p \wedge E^z_{false}))$ ⟩

$(p \wedge E) \vee (\neg p \wedge E)$

= ⟨ *h1* (E), *h2* (E) ⟩

$(p \wedge true) \vee (\neg p \wedge true)$

= ⟨ identidad de $\wedge$ $(p \wedge true \equiv p)$, con p := p y p := ¬p⟩

$p \vee \neg p$ — Medio excluido $(p \vee \neg p \equiv true)$ —

Metateorema 3. [*Sección 3.2*]

$\boldsymbol{H_1 \vee H_2 \vdash\!\!\!- E_{izq} = E_{der}}$ ***si y sólo si*** $\boldsymbol{H_1 \vdash\!\!\!- E_{izq} = E_{der}}$ ***y*** $\boldsymbol{H_2 \vdash\!\!\!- E_{izq} = E_{der}}$

Demostración:

Podemos, por el *Teorema de la Deducción*, reformular el teorema así:

$A \vee B \Rightarrow C \equiv (A \Rightarrow C) \wedge (B \Rightarrow C)$

$(A \Rightarrow C) \wedge (B \Rightarrow C) \equiv A \vee B \Rightarrow C$

= ⟨ *Definición de* $\Rightarrow$ $(p \Rightarrow q \equiv p \vee q \equiv q)$, *tres veces* ⟩

$(A \vee C \equiv C) \wedge (B \vee C \equiv C) \equiv A \vee B \vee C \equiv C$

= ⟨ $p \wedge (q \equiv r) \equiv p \wedge q \equiv p \wedge r \equiv p$, $p := A \vee C \equiv C$, $q := B \vee C$, $r := C$ ⟩

$(A \vee C \equiv C) \wedge (B \vee C) \equiv (A \vee C \equiv C) \wedge C \equiv A \vee C \equiv C \equiv A \vee B \vee C \equiv C$

= ⟨ $p \wedge (q \equiv r) \equiv p \wedge q \equiv p \wedge r \equiv p$, $p := B \vee C$, $q := A \vee C$, $r := C$ *y*

$p \wedge (q \equiv r) \equiv p \wedge q \equiv p \wedge r \equiv p$, $p := C$, $q := A \vee C$, $r := C$ *y*

Identidad de ≡ *(p ≡ p ≡ true),* dos veces con *p := C* ⟩

(A ∨ C) ∧ (B ∨ C) ≡ C ∧ (B ∨ C) ≡ B ∨ C ≡ (A ∨ C) ∧ C ≡ C ∧ C ≡ C ≡ A ∨ C ≡ A ∨ B ∨ C

= ⟨ Absorción (*p ∧ (p ∨ q) ≡ p*), p := C, q := B *y* con p := C, q := A ⟩

(A ∨ C) ∧ (B ∨ C) ≡ C ≡ B ∨ C ≡ C ≡ A ∨ C ≡ A ∨ B ∨ C

= ⟨ Identidad de ≡, dos veces con p := C ⟩

(A ∨ C) ∧ (B ∨ C) ≡ A ∨ C ≡ B ∨ C ≡ A ∨ B ∨ C

= ⟨ Regla de Oro (*p ∧ q ≡ p ≡ q ≡ p ∨ q*) con p := A ∨ C, q := B ∨ C ⟩

(A ∨ C) ∨ (B ∨ C) ≡ A ∨ B ∨ C

= ⟨ Asociatividad e idempotencia de ∨ ⟩

A ∨ B ∨ C ≡ A ∨ B ∨ C — Identidad de ≡ —

ÍNDICE ANALÍTICO

A continuación encontrará una lista de las palabras consideradas importantes en este documento con los números de las páginas en que se hace referencia a ellas. Algunas de las palabras están relacionadas y clasificadas, para dar una referencia más amplia, que permita comprender los conceptos y definiciones.

La lista está ordenada alfabéticamente; sin embargo, algunos términos se encuentran incluidos dentro de categorías más generales, la cual cambia su posición dentro del índice.

Ambiente 12
Árbol sintáctico 12
Axioma 6, 7, 10, 12, 14
Casos 15, 26, 28, 33
Condicional 28
Congruencia 34
Conjetura 9, 10, 11
Constante 6, 9, 13
Contexto 26
Contraejemplos 40
Corrección 15, 17
Correspondencia de patrones 13, 14
Diseño 5
Especificación 6
ERROR 8, 9, 14, 15, 20, 21
Expresión 8, 9, 12, B−1
Forma normal 12, 14, 17
Función 6, 13
    EQUAL 9, 15, 17, 34
    if−then−else 8, 9, 20, 21, 28, 29
    inicial 6

constructora 7
selectora 7
Generalización 30, 31, 33, 37
Hipótesis al generalizar 39
Hipótesis 25, 27, 33, 34
Hipótesis al generalizar 39
Disyunción de hipótesis 29
Suposición de hipótesis 26
Inducción 16, 19, 30, 34
Igualdad 6, 12, 15, 17, 19, B−2
Innermost 37
Leftmost − Outermost 12, 14, 37
Leibniz 27, 35
Lema 11, 21 – 24, 34
ManTa 2, 5
Match 13
Método (ver también *prueba*) 15, 19, 26, 28, 32
Inducción 16, 19, 30, 34
Casos 15, 26, 28, 33
Generalización 30, 31, 33, 37
Semántica 6, 14, 17, 19
Modelo 2, 5, 10
Notación 3
de ManTa 9
Objetivo 13
Operación (ver también *función*) 16
Orden 19, 34
Parámetro 6, 7, 8
Patrón 13
Preservación de funciones 34
Probador de teoremas 2, 5, 9, 11
Prueba (ver también *métodos*) 2, 15
Regla 26

de reescritura 7, 14, 16, 20, 21
especial 14
Reescritura 9, 12, 13, 14, 19, 20, 21, 25, B−2
Semántica 6, 14, 17, 19
Shannon 29 D−2
Signatura 6, 7
Sintaxis 6, 8, 9
Sistema 12, 22
Total 12
confluente 12
completo 20
Suposición de hipótesis 26
Sustitución 13
TAD − Tipo Abstracto de Datos 2, 5, 6, 7, 8
TADs ecuacionales 5
Teorema (ver también *lemas* y *conjeturas*) 2, 10, 11
de la Deducción 26, 27, 28
Terminación 17, 20
Término (ver también *expresión*) 20
Tipo de interés (*tdi*) 6, 7
Valor 7, 9, 12, 14, 18
Variable 6, 9, 12
Verificación de EQUAL 28, 34