Homomorfismos entre módulos de cuerda - Un algoritmo implementado en Java

Tesis que para obtener el grado de

MAESTRA EN CIENCIAS (MATEMÁTICAS)

presenta

Sandra Dieckmann Fulger

Directores de Tesis: Dr. Christof Geiss

Dr. José Antonio de la Peña Mena

Facultad de Ciencias

Universidad Nacional Autónoma de México

México, D.F.

2000

Agradecimientos

No he tratado de buscar en ninguna parte las palabras (ni en español ni en otro idioma) con las cuales quiero agradecer y dedicar este trabajo a la gente que ha estado conmigo.

Agradezco en especial a Evelyn por sus observaciones del programa en Java y los diagramas de UML y por haberme enseñado como organisar el projecto de mis tesis en general. Le agradezco a Paco por haberme apoyado siempre.

A todos mis profesores que han contribuido a que pueda estar haciendo esto y a todos los amigos que durante la carrera yo he conseguido.

Finalmente quiero agradecer a las valiosas observaciones de Rita, que hizieron la tesis legible y a Benjamin por tener la pacienda sacar este texto de la computadora.

Contenido

Introducción

En teoría de representaciones de álgebras se quiere describir módulos y los homomorfismos entre ellos. Si nos restringimos a los módulos de dimensión finita , por el teorema de Krull-Schmidt nos podemos concentrar en los módulos inescindibles. La teoría de Auslander-Reiten sirve como una primera aproximación a una descripción de los homomorfismos.

En la búsqueda de modelos para aclarar las relaciones entre módulos y homomorfismos destaca el de las álgebras de cuerda, por ser fácil y claro. Trabajo recientes sobre este tema han sido desarolladas por Geiss [Ge2], Ringel [Ri2] y Schröer [Schroe], [Schroe2]. En un álgebra de cuerda se puede determinar con un algoritmo los homomorfismos para dos módulos inescindibles dados.

En la presente tesis se implementa un algoritmo en Java para determinar dada un álgebra de cuerdas $\Lambda = kQ/I$ y dos módulos de cuerda $M(v)$, $M(w)$ una k-base del espacio vectorial de homomofismo $\mathrm{Hom}_\Lambda(M(v), M(w))$.

En la parte práctica se desarrolla una interfaz gráfica de usuario que permite editar de una forma cómoda el álgebra $\Lambda = kQ/I$ y las cuerdas v y w. El programa determina una base del espacio de homomorfismos entre los dos módulos insertados.

El apéndice contiene explicaciones acerca de la programación orientada a objetos y en especial de Java. En un glosario se explican algunos de los términos usados. Se anexa un disket con el programa en Java.

Primera Parte

La Parte de la Teoría

Capítulo 1
Preliminares

En este capítulo se dan las bases para el trabajo subsecuente, se ve aquí la teoría básica necesaria para el desarrollo de los temas importantes contenidos en esta tesis. En este capítulo nos hemos apoyado en el trabajo de Ciblis, Larrión y Salmerón [CiLaSa], que recomendamos a quien desee estudiar más a fondo los temas tratados.

Se define un carcaj con sus relaciones y se le asocia una k-álgebra Λ de dimensión finita donde k es un campo algebraicamente cerrado.

El interés en estudiar carcajes es motivado por la equivalencia entre las representaciones de los carcajes con relaciones y los módulos sobre el álgebra cociente del álgebra de caminos asociada.

1.1 Carcajes y sus representaciones

En este trabajo se denota con Λ una k-álgebra de dimensión finita donde k es un campo algebraicamente cerrado.

La teoría de representaciones de álgebras de dimensión finita pretende estudiar a Λ a través de la categoría de sus módulos de dimensión finita, denotada por $mod\Lambda$. Estos módulos se pueden descomponer en una suma directa de módulos inescindibles y con el teorema de Krull-Schmidt se asegura que esta descomposición sea única hasta isomorfismo.

Recordemos que Λ se llama de tipo de representación finita si existe únicamente un número finito de clases de isomorfia de módulos inescindibles en modΛ.

1.1.1 Carcajes con relaciones

Definición 1 Un *carcaj* $\mathbf{C} = (\mathbf{C}_0,\mathbf{C}_1)$ es una gráfica orientada, conexa y finita con $\mathbf{C}_0$ el conjunto de los vértices y $\mathbf{C}_1$el conjunto de las flechas. Para toda $\alpha \in \mathbf{C}_1$, denotaremos como $\mathbf{s}(\alpha)$ su *vértice inicial* y como $\mathbf{t}(\alpha)$ su *vértice final.*

Definición 2 Un *camino dirigido* p (de longitud $n\geq 1$) del vértice i al vértice j en un carcaj C es una sucesión ordenada de flechas $p=\alpha_n \cdots \alpha_1$ con $t(\alpha_t)=s(\alpha_{t+1})$ para $1\leq t<n$. Además en el caso de que n=0 tomando i=j se asocia a cada vértice $i\in C_0$ su *camino trivial* denotado τ_i. Para un camino no trivial $p=\alpha_n \cdots \alpha_1$ definimos $s(p)=s(\alpha_1)$ y $t(p)=t(\alpha_n)$. Un camino no trivial se llama *cíclo orientado* si s(p)=t(p).

Definición 3 A cualquier carcaj C se le puede asociar el *álgebra de carcaj* **kC** de la siguiente manera. Primero se denota por kC el k-espacio vectorial que tiene como k-base la colección de todos los caminos dirigidos (de longitud finita) de C, incluidos los caminos triviales. Si ρ y σ son dos caminos, el producto $\rho\sigma$ es el camino compuesto, si el inicio de ρ coincide con el final de σ, y cero en otro caso. Este producto se extiende de manera única a todo kC de tal forma que se obtiene una k-álgebra denotada por kC.

Observación 1 Los caminos triviales se pueden identificar con los vértices de C y $1=\sum_{i\in C_0} \tau_i$ forma un sistema completo de idempotentes ortogonales y primitivos de kC. Los caminos de longitud uno en C se identifican con las flechas de C, además los vértices y las flechas de C constituyen un sistema de generadores para el álgebra kC.

Proposición 1 *Un álgebra de carcaj kC es una k-álgebra de dimensión finita si y sólo si C no tiene cíclos dirigidos.*

Definición 4 Sea Λ una k-álgebra básica. El *carcaj ordinario* $\mathbf{C}_\Lambda$ de Λ se define como sigue: se toma un vértice por cada elemento del sistema completo de idempotentes ortogonales y primitivos $\{e_1, \ldots, e_n\}$ que se haya elegido. En lo que concierne a las flechas de C_Λ, el número de ellas que principan en el vértice i y terminan en el vértice j es $\dim_k[e_j(rad\Lambda/rad^2\Lambda)e_i]$.

Lema 1 Para $C = C_\Lambda$ el carcaj ordinario de Λ existe un morfismo suprayectivo de k-álgebras ϕ :k$C_\Lambda \to \Lambda$.

Definición 5 Un *ideal admisible* $\mathcal{R}$ de kC está contenido en el cuadrado $\mathcal{F}^2$ del ideal $\mathcal{F}$ de kC generado por las flechas de C y contiene a $\mathcal{F}^m$ para m suficientemente grande.

Lema 2 El núcleo del morfismo ϕ es un ideal admisible.

En consecuencia, toda k-álgebra básica Λ de dimensión finita es cociente $\Lambda = kC/\mathcal{R}$ de un álgebra de carcaj kC módulo un ideal admisible $\mathcal{R}$. Además, todo ideal admisible de un álgebra de carcaj es finitamente generado y admite un sistema finito de generadores legibles.

Definición 6 El par $(C,\mathcal{R})$, formado por un carcaj C y un ideal admisible $\mathcal{R}$ del álgebra de carcaj se llama un *carcaj con relaciones.*

A menudo se sustituye $\mathcal{R}$ por un sistema finito de relaciones legibles que lo generen.

1.1.2 Módulos y representaciones

Vimos en la sección pasada como se asocia un carcaj con relaciones $(C,\mathcal{R})$ a un álgebra $\Lambda = kC/\mathcal{R}$. Veremos que se le puede asociar a cada Λ-módulo de dimensión finita una representación de $(C,\mathcal{R})$.

Definición 7 Una *k-representación (o representación)* del carcaj C es una pareja $V = ((V_i),(f_\alpha))$ de una familia de k-espacios vectoriales V_i, uno por cada vértice i de C, y una familia de transformaciones k-lineales $f_\alpha : V_i \to V_j$, por cada flecha $\alpha : i \to j$ de C.

Definición 8 Un *morfismo de representaciones* $\Phi : V \to V'$ es una familia $\Phi = (\Phi_i) = (\Phi_i : V_i \to V_i')$ de transformaciones lineales, una por cada vértice i de C, tales que para cada flecha $\alpha : i \to j$ en C, el siguiente diagrama es conmutativo,

$$\begin{array}{ccc} \mathrm{V}_i & \xrightarrow{f_\alpha} & \mathrm{V}_j \\ \Phi_i \downarrow & & \Phi_j \downarrow \\ \mathrm{V'}_i & \xrightarrow{f'_\alpha} & \mathrm{V'}_j \end{array}$$

o sea que $\Phi_j f_\alpha = f'_\alpha \Phi_i$ para toda $\alpha : i \to j$ en C.

Definición 9 Si V es una representación de C y $\gamma = \alpha_n, \alpha_{n-1}, \ldots, \alpha_1$ con $s(\gamma) = i$ y $t(\gamma) = j$ es un camino dirigido no trivial de C, podemos evaluar V en γ como sigue:

$$V(\gamma) := f_{\alpha_n} \circ f_{\alpha_{n-1}} \circ \ldots \circ f_{\alpha_1}$$

Por lo tanto $V(\gamma) : V_i \to V_j$ es una transformación lineal.

Si $\rho = \sum \lambda_\gamma \gamma$ es una relación legible, de i a j, tiene sentido evaluar V en ρ poniendo:

$$V(\rho) = \sum \lambda_\gamma V(\gamma)$$

Nuevamente, $V(\rho) : V_i \to V_j$ es una transformación líneal.

Definición 10 Sea V una representación de C y ρ una relación legible de $\mathcal{R}$. Se dice que V *satisface* ρ si la anula, es decir, si $V(\rho) = 0$. Se dice que V *satisface* $\mathcal{R}$ si V satisface cada relación legible de $\mathcal{R}$.

Definición 11 El conjunto de representaciones de (C, R) junto con los morfismos definidos anteriormente forman una categoría denotada por $Mod(C, R)$.

Con $mod(C, R)$ se denota a la subcategoría plena de $Mod(C, R)$ cuyos objetos son las representaciones V de $Mod(C, R)$ tales que V_i es de dimensión finita para todo vértice i$\in C_0$.

En virtud de lo anterior se obtiene.

Teorema 1 *Las k-categorías mod*Λ *y mod(C,* $\mathcal{R}$*) son equivalentes.*

Hemos visto que las representaciones de dimensión finita de un carcaj con relaciones (C, R) están en biyección con los módulos de dimensión finita del álgebra $\Lambda = kC/\mathcal{R}$, se pueden definir representaciones simples, proyectivos inescindibles, radicales etc.

Capítulo 2
Sucesiones de Auslander-Reiten

Sea Λ una k-álgebra de dimensión finita con k un campo algebraicamente cerrado. En este capítulo se considera la categoría de los $\Lambda-$módulos de dimensión finita denotada por $mod\Lambda$. Por el teorema de Krull-Schmidt nos basta enfocarnos en los módulos inescindibles.

En este parráfo se consideran las sucesiones de Auslander-Reiten, sucesiones que casi se dividen los cuales son un tipo especial de sucesiones exactas cortas. Con lo anterior veremos también los morfismos irreducibles entre dos módulos inescindibles. Para mayores detalles pueden consultarse los libros de Auslander, Reiten y Smalø [AuReSm] y de Ringel [Ri4].

2.1 Sucesiones de Auslander-Reiten

Recordemos:

Definición 12 Sea A un Λ-módulo y $P_1 \xrightarrow{f} P_0 \to A \to 0$ una presentación minimal proyectiva, entonces la *traspuesta* TrA de A es el cokernel del mapeo $\mathrm{Hom}_\Lambda(f, \Lambda)$ en $\mathrm{mod}\Lambda^{op}$, con Λ^{op} el álgebra opuesto de Λ.

Definición 13 Sea Λ un k-álgebra y $X \xrightarrow{f} Y$ un morfismo con X, y Y Λ-módulos inescindibles.

a) Se dice que f es un *epimorfismo que se divide* si f es epimorfismo y si existe $\mu : Y \to X$ tal que $f\mu = 1_Y$.

b) Se dice que f es un *monomorfismo que se divide* si f es monomorfismo y si existe $\sigma : Y \to X$ tal que $\sigma f = 1_X$.

Definición 14 Sea $f{:}X \to Y$ un morfismo entre dos inescindibles, se dice que f
a) *casi se divide por la derecha* si

f no es epimorfismo que se divide

y para todo morfismo $g : Z \to Y$ que no es epimorfismo que se divide existe $\sigma : Z \to X$ tal que $f\sigma = g$.

Análogamente:

b) *casi se divide por la izquierda* si

f no es monomorfismo que se divide

y para todo morfismo $h : X \to Z$ que no es monomorfismo que se divide existe $\mu : Z \to X$ tal que $\mu f = h$.

Definición 15 Una sucesión exacta de Λ-módulos de dimensión finita $0 \to A \xrightarrow{g} B \xrightarrow{f} C \to 0$, se llama *sucesión de Auslander-Reiten* (ó sucesión que casi se divide) si f casi se divide por la derecha y g casi se divide por la izquierda.

Proposición 2 *Sea* $\delta : 0 \to A \xrightarrow{g} B \xrightarrow{f} C \to 0$ *una sucesión exacta. Entonces son equivalentes:*

a) δ *es sucesión de Auslander-Reiten.*

b) El módulo A es inescindible y f casi se divide por la derecha.

c) El módulo C es inescindible y g casi se divide por la izquierda.

d) El módulo C es isomorfo al TrDA y g casi se divide por la izquierda.

e) El módulo A es isomorfo al DTrC y f casi se divide por la derecha.

donde TrC es el módulo traspuesto de C y DA el módulo dual de A.

Por la proposición anterior se obtiene una biyección entre las clases de isomorfía de los módulos inescindibles no proyectivos de un álgebra y las clases de isomorfía de los módulos inescindibles no inyectivos dado por la traspuesta de Auslander-Reiten. Para A un Λ-módulo se denota con τA el módulo DTrA y con τ^-A el módulo TrDA.

La existencia de sucesiones de Auslander-Reiten esta dada por lo siguiente.

Proposición 3

a) Sea C un módulo inescindible no proyectivo en modΛ*. Entonces existe una sucesión que casi se divide* $0 \to \tau^- C \xrightarrow{g} B \xrightarrow{f} C \to 0$.

b) Sea A un módulo inescindible no inyectivo en modΛ*. Entonces existe una sucesión que casi se divide* $0 \to A \xrightarrow{g} B \xrightarrow{f} \tau A \to 0$.

Definición 16 Sean X,Y$\in mod\Lambda$, un *morfismo* $f : X \to Y$ se llama *irreducible* si

a) no es epimorfismo divisible, ni es monomorfismo que se divide y

b) si $f = vu$ entonces u es monomorfismo divisible ó v es epimorfismo que se divide.

Lema 3 Sean B y C Λ-módulos entonces:

1) Si $g : B \to C$ es un morfismo irreducible en $mod\Lambda$, g es monomorfismo o g es epimorfismo.

2) Si $g : B \to C$ es un monomorfismo irreducible, entonces B es sumando de todos los submódulos propios de C que contienen a B.

3) Si $g : B \to C$ es un epimorfismo irreducible, entonces C es sumando de B/I, para todo I submódulo de B tal que $0 \neq I \subset Kerg$.

Lema 4 Si $g : B \to C$ es un morfismo irreducible, entonces g es minimal por la derecha y por la izquierda.

El siguiente teorema da la conexión entre morfismos irreducibles y morfismos que casi se dividen por la derecha y la izquierda.

Teorema 2

1) *Sean B y C módulos inescindibles. Entonces* $g : B \to C$ *es un morfismo irreducible si y sólo si existe un morfismo* $g' : B' \to C$ *tal que el morfismo inducido* $(g, g') : B \bigoplus B' \to C$ *es morfismo minimal por la derecha que casi se divide.*

2) *Sean A y B módulos inescindibles. Entonces* $g : A \to B$ *es un morfismo irreducible si y solo si existe un morfismo* $g' : B' \to C$ *tal que el morfismo inducido* $\binom{g}{g'} : A \to B \bigoplus B'$ *es morfismo minimal por la izquierda que casi se divide.*

La siguiente proposición considera la conexión entre morfismos irreducibles con morfismos que se dividen por la derecha o la izquierda.

Proposición 4 *Sea* $\delta : 0 \to A \to B \to C \to 0$ *una sucesión exacta. Entonces es* δ *una sucesión que casi se divide si y solo si f y g son irreducibles.*

Proposición 5 *Sea* $\delta : 0 \to A \xrightarrow{f} B \xrightarrow{g} C \to 0$ *una sucesión exacta. Entonces*

1) *si f es un morfismo irreducible,* $End_\Lambda(C)$ *es un anillo local y por lo tanto C es un módulo inescindible.*

2) *si g es un morfismo irreducible,* $End_\Lambda(A)$ *es un anillo local y por lo tanto A es un módulo inescindible.*

3) δ *es una sucesión de Auslander-Reiten si y sólo si f y g son morfismos irreducibles.*

2.2 El término medio de una sucesión que casi se divide

Para una sucesión que casi se divide $0 \to A \to B \to C \to 0$ decimos que los módulos A y C son los elementos laterales y que el elemento B es el término medio de la sucesión.

Aunque los elementos laterales A y C son inescindibles, el término medio se descompone en general.

Definición 17 Definimos por $\alpha(C)$ al número de sumandos inescindibles de un término medio de una sucesión de Auslander Reiten, el cual es un invariante importante de C.
Se puede interpretar el invariante $\alpha(C)$ como una medida la complejidad de los morfismos que terminan en C.

Para cada entero positivo t existe un módulo inescindible sobre algún álgebra de dimensión finita sobre un campo k con $\alpha(C) = t$. Para un álgebra Λ de esta forma se denota con $\alpha(\Lambda)$ el supremo de los $\alpha(C)$ para todos los inescindibles C.

2.3 El funtor de extensión

En esta sección recordemos algunas definiciones del funtor $\mathrm{Ext}^1_\Lambda(C, A)$, para más detalles consulte el libro de Rotman [Ro].

Definición 18

1) Dos sucesiones exactas de $\Lambda-$módulos $E : 0 \to A \to B \to C \to 0$ y $E' : 0 \to A \to B' \to C \to 0$ se llaman *equivalentes* si existe $\varphi \in \mathrm{Hom}_\Lambda(B, B')$ tal que conmuta el siguiente diagrama:

$$\begin{array}{ccccccccc} E: & 0 & \to & A & \to & B & \to & C & \to & 0 \\ & & & \| & & \varphi\downarrow & & \| & & \\ E' & 0 & \to & A & \to & B' & \to & C & \to & 0 \end{array}$$

2) Se denota la clase de equivalencia de una sucesión exacta corta con $[E]$ y el conjunto de todas estas clases de equivalencia con $\mathrm{Ext}^1_\Lambda(C, A)$.

Proposición 6 *Dos sucesiones exactas cortas que empiezan y terminan en el mismo módulo son equivalentes.*

Proposición 7

1) Sea $E : 0 \to A \to B \to C \to 0$ *una sucesión exacta. Consideremos el siguiente diagrama:*

$$\begin{array}{lccccccccc} E: & 0 & \to & A & \xrightarrow{f} & B & \xrightarrow{g} & C & \to & 0 \\ & & & \alpha\downarrow & & & & \| & & \\ & & & A' & & & & C & & \end{array}$$

Entonces existe una sucesión exacta $E' : 0 \to A' \xrightarrow{f'} B' \xrightarrow{g'} C \to 0$ *y un homomorfismo* $\beta : B \to B'$*, tal que el siguiente diagrama conmuta:*

$$\begin{array}{lccccccccc} E: & 0 & \to & A & \xrightarrow{f} & B & \xrightarrow{g} & C & \to & 0 \\ & & & \alpha\downarrow & & \beta\downarrow & & \| & & \\ E': & 0 & \to & A' & \xrightarrow{f'} & B' & \xrightarrow{g'} & C & \to & 0 \end{array}$$

Además en cada diagrama de este estilo, el primer cuadrado es un pushout. Finalmente si E es una sucesión que se divide, tambíen E' se divide.

2) Analogamente para el siguiente diagrama:

$$\begin{array}{lccccccccc} & & & A & & & & C'' & & \\ & & & \| & & & & \gamma\downarrow & & \\ E: & 0 & \to & A & \xrightarrow{f} & B & \xrightarrow{g} & C & \to & 0 \end{array}$$

existe una sucesión exacta $E'' : 0 \to A \xrightarrow{f''} B'' \xrightarrow{g''} C'' \to 0$ *y un homomorfismo* $\beta' : B'' \to B$*, que completan el diagrama para que conmute; en este caso el segundo cuadro es un pullback y E'' se divide si E se divide.*

Definición 19 Las clases de equivalencia de las sucesiones E' y E'' de la proposición anterior se denotan $[\alpha E]$ y $[E\gamma]$, con frecuencia los representantes de estas clases de equivalencia se escriben simplemente como αE, repectivamente $E\gamma$.

Teorema 3 *Con la suma de Baer* $\mathrm{Ext}^1_\Lambda(C, A)$ *resulta ser un grupo abeliano aditivo.*

Definición 20 *Sean* $\alpha : A \to A'$ *y* $\gamma : C' \to C$ *morfismos de* Λ*-módulos. Se definen los funtores*:

$$\mathrm{Ext}^1_\Lambda(C, \alpha) : \mathrm{Ext}^1_\Lambda(C, A) \to \mathrm{Ext}^1_\Lambda(C, A') \text{ tal que } [E] \longmapsto [\alpha E] \text{ y}$$
$$\mathrm{Ext}^1_\Lambda(\gamma, A) : \mathrm{Ext}^1_\Lambda(C, A) \to \mathrm{Ext}^1_\Lambda(C', A) \text{ tal que } [E] \longmapsto [E\gamma]$$

Teorema 4 *Sean* A *y* C $\Lambda-$ *módulos y sea* Ab *la categoría de grupos abelianos. Entonces: El funtor* $\mathrm{Ext}^1_\Lambda(C, -) : \Lambda - Mod \longrightarrow Ab$ *esta dado por* $A \longmapsto \mathrm{Ext}^1_\Lambda(C, A)$.

Si $\alpha \in \mathrm{Hom}_\Lambda(A', A'')$ *entonces* $\alpha \longmapsto \mathrm{Ext}^1_\Lambda(C, \alpha)$ *resulta ser un funtor covariante aditivo.*

El funtor $\mathrm{Ext}^1_\Lambda(-, A) : \Lambda - Mod \longrightarrow Ab$ *esta dado por* $C' \longmapsto \mathrm{Ext}^1_\Lambda(C', A)$.

Si $\gamma \in \mathrm{Hom}_\Lambda(C', C'')$ *entonces* $\gamma \longmapsto \mathrm{Ext}^1_\Lambda(\gamma, A)$ *resulta ser un funtor contravariante y aditivo.*

2.4 Equivalencia entre extensiones y homomorfismos

Definición 21 Un $\Lambda-$módulo M se llama *finitamente presentado* si existe una sucesión exacta $P_1 \to P_0 \to M \to 0$ con P_0 y P_1 $\Lambda-$módulos finitamente generados y projectivos. En lo que queda de la sección sea C un $\Lambda-$módulo izquierdo, A un $\Lambda-$módulo izquierdo

finitamente presentado y Γ un anillo, tal que TrA es un $\Gamma - \Lambda-$bimódulo. El anillo Γ existe, porque TrA es un $\mathrm{End}_\Lambda(TrA) - \Lambda-$bimódulo.

Definimos $Q_1 \xrightarrow{g_1} Q_0 \xrightarrow{g_0} A \to 0$ una presentación proyectiva de A tal que los Q_i son finitamente generados. La TrA es dado por la proyección canonica $\pi : Q_1^* \to TrA = Q_1^*/\operatorname{Im}(g_1^*)$.

Además sea P_0 un $\Lambda-$módulo izquierdo proyectivo. Tomamos $f_0 : P_0 \to C$ un epimorfismo y definimos $K :=\mathrm{Ker}(f_0)$ entonces la sucesión $0 \to K \xrightarrow{f_1} P_0 \xrightarrow{f_0} C \to 0$ es exacta.

Lema 5 Sea I un $\Gamma-$módulo izquierdo. $\mathrm{Hom}_\Gamma(TrA, I)$ es un $\Lambda-$módulo izquierdo, con $(\lambda \cdot f)(x) := f(x\lambda)$, para $\lambda \in \Lambda$, f$\in\mathrm{Hom}_\Gamma(TrA, I)$ y x$\in TrA$.

Definición 22 Dado dos $\Lambda-$módulos M y N sea $\underline{\mathrm{Hom}}_\Lambda(M, N)$ el cociente del Hom(M,N) modulo el subgrupo de todos los morfismos que factorisan sobre un $\Lambda-$módulo proyectivo. Análogamente se define $\underline{\mathrm{End}}_\Lambda(M)$.

Lema 6 $\underline{\mathrm{Hom}}_\Lambda$(A,C) es un $\underline{\mathrm{End}}_\Lambda$(A)-módulo derecha.

Lema 7 $\underline{\mathrm{Hom}}_\Lambda(A, C)$ es $\Gamma-$módulo izquierda.

Vimos que $\mathrm{Hom}_\Gamma(TrA, I)$ es un $\Lambda-$módulo y $\underline{\mathrm{Hom}}_\Lambda(A, C)$ un $\Gamma-$módulo entonces podemos construir un isomorfismo de grupos : $\mathbf{Ext}^1_\Lambda(C,\mathbf{Hom}_\Gamma(TrA, I)) \xrightarrow{\simeq}\mathbf{Hom}_\Gamma(\underline{\mathbf{Hom}}_\Lambda(A, C), I)$. Este isomorfismo se llama *formula de Auslander-Reiten*. Más detalles sobre la demostración de la formula de Auslander-Reiten se encuentra en [AuReSm].

Capítulo 3
Álgebras Biseriales

3.1 Álgebras biseriales

Definición 23 La k-álgebra de dimensión finita Λ es *biserial* si todo Λ-módulo proyectivo inescindible izquierdo o derecho P contiene dos submódulos uniseriales (i.e., con una única serie de composición) K y L que son Λ-módulos de tal manera que la suma K+L es P o es el máximo submódulo propio de P (i.e., el radical de Jacobson de P) y la intersección K$\cap$L es cero o es el mínimo submódulo no nulo de P.

Definición 24 Un álgebra Λ se llama *biserial especial* si es Morita equivalente a un álgebra de carcaj $kC/\mathcal{R}$ donde la pareja (C,R) satisface las siguientes condiciones:

1) a) En ningun vértice empiezan más que dos flechas.

b) En ningun vértice terminan más que dos flechas.

2) a) Para toda flecha $\alpha \in C_1$ existe a lo más una flecha β tal que $\beta\alpha$ no esta en $\mathcal{R}$.

b) Para toda flecha $\alpha \in C_1$ existe a lo más una flecha γ tal que $\alpha\gamma$ no esta en $\mathcal{R}$.

3) Existe un límite para la longitud de los caminos en C que no están en $\mathcal{R}$.

Un álgebra biserial especial Λ satisface $\alpha(\Lambda) \leq 2$. Es decir, las sucesiones de Auslander-Reiten tienen a lo más dos sumandos no-proyectivos en su término medio.

Lema 8 Toda álgebra biserial epecial es biserial.

3.2 Álgebras de cuerda

Definición 25 Sea C un carcaj, R un conjunto dado por caminos de longitud ≥ 2 y k un campo algebraicamente cerrado. Consideramos kC el álgebra de caminos y $\mathcal{R}$ el ideal generado por R. El álgebra de caminos $A = kC/\mathcal{R}$ es un *álgebra de cuerda* si satisface lo siguiente:

1) Para cada vértice i en C hay a lo más 2 flechas que llegan a i y a lo más 2 flechas que salen de i.

2) Para cada flecha β en C hay a lo más una flecha γ tal que $\beta\gamma \notin R$ y a lo más una flecha α tal que $\alpha\beta \notin R$.

3) Para cada flecha β existe un límite $n(\beta)$ tal que cada camino $\beta_{n(\beta)}...\beta_1$, con $\beta_1 = \beta$ contiene un subcamino en R y un límite $n'(\beta)$ tal que cada camino $\beta_{n'(\beta)}...\beta_1$, con $\beta_{n'(\beta)} = \beta$ contiene un subcamino en R.

Lema 9 Toda álgebra de cuerda es un álgebra biserial especial. Por otra parte, un álgebra biserial especial Λ cuyos módulos inescindibles que son proyectivos e inyectivos simultaneamente, son seriales, es un álgebra de cuerda.

Para definir los módulos inescindibles de un álgebra de cuerda definimos las siguientes nociones: sea $\beta \in C_1$, β^{-1} denota su inversa formal con $s(\beta^{-1}) := t(\beta)$ y $t(\beta^{-1}) :=$

$s(\beta)$, además $(\beta^{-1})^{-1} = \beta$. Se construyen "*palabras*" $c_n \dots c_1$ de longitud $n \geq 1$ donde c_i es de la forma β o β^{-1} y $s(c_{i+1}) = t(c_i)$, $1 \leq i < n$. Además se define $(c_n \dots c_1)^{-1} = c_1^{-1} \dots c_n$, $s(c_n \dots c_1) = s(c_1)$, $t(c_n \dots c_1) = t(c_n)$.

Definición 26 Una palabra $c_n \dots c_1$ de longitud $n \geq 1$ se llama *cuerda* si para toda $1 \leq i < n$ se tiene $c_i \neq c_{i+1}^{-1}$ y ninguna subpalabra $c_{i+t} \dots c_{i+1} c_i$ ni su inverso están en R.

Para cada vértice $u \in C_0$ existen dos cuerdas de longitud 0, denotadas $1_{(u,1)}$ y $1_{(u,-1)}$ con $s(1_{(u,t)}) = u$, $t(1_{(u,1)}) = u$ y $t = 1, -1$. Además definimos $1_{(u,i)}^{-1} = 1_{(u,-i)}$.

Con objeto de definir la composición de cuerdas introducimos las funciones $\sigma, \varepsilon : C_1 \to \{1, -1\}$ con las siguientes propiedades:

1) si $\beta_1 \neq \beta_2$ son flechas con $s(\beta_1) = s(\beta_2)$ entonces $\sigma(\beta_1) = -\sigma(\beta_2)$.

2) si $\gamma_1 \neq \gamma_2$ son flechas con $t(\gamma_1) = t(\gamma_2)$ entonces $\varepsilon(\gamma_1) = -\varepsilon(\gamma_2)$.

3) si β,γ son flechas con $s(\beta) = t(\gamma)$ y $\beta\gamma \in R$ entonces $\sigma(\beta) = -\varepsilon(\gamma)$.

Extendemos las funciones σ y ε al conjunto de todas las cuerdas como sigue:

Si $\beta \in C_1$, $\sigma(\beta^{-1}) = \varepsilon(\beta)$, $\varepsilon(\beta^{-1}) = \sigma(\beta)$; para $V = c_n \dots c_1$ una cuerda de longitud $n \geq 1$ sea $\sigma(V) = \sigma(c_n)$, $\varepsilon(V) = \varepsilon(c_n)$ y para $\sigma(1_{(u,t)}) = -t$, $\varepsilon(1_{(u,t)}) = t$.

Sea $W = d_m \dots d_1$ una cuerda de longitud $m \geq 1$, la *composición* de V y W *está definida* si $VW = c_n \dots c_1 d_m \dots d_1$ es una cuerda. La composición de $1_{(u,t)}W = W$ está definida si $t(W) = u$, $\varepsilon(W) = -t$ y la composición de $V1_{(u,t)} = V$ está definida si $s(V) = u$, $\sigma(V) = -t$.

Observemos que si VW está definido, entonces necesariamente $\sigma(V) = -\varepsilon(W)$.

3.3 Los módulos de cuerda y de banda

Se considera para la definición de los módulos de cuerda y los módulos de banda unos conjuntos de cuerdas:

Sea $\mathcal{V}(u,t)$ el conjunto de cuerdas V con $t(V) = u$, $\varepsilon(V) = t$, i.e. $\mathcal{V}(u,t)$ contiene $1_{(u,t)}$ y todas las cuerdas de la forma $V = c_n \dots c_1$ donde $c_n = \beta$, con $t(\beta) = u$, $\varepsilon(\beta) = t$ o $c_n = \gamma^{-1}$, con $s(\gamma) = u$, $\sigma(\gamma) = t$.

Sea $\mathcal{W}(u,t)$ el subconjunto de $\mathcal{V}(u,t)$, tal que para cada $V \in \mathcal{W}(u,t)$ están definidos todos los V^n con $n \in \mathbb{N}_1$ y $V \neq W^m$ para algún $m \in \mathbb{N}$, $W \in \mathcal{W}(u,t)$.

Sea $\mathcal{V}$ el conjunto de todas las cuerdas y $\mathcal{W}$ el subconjunto de $\mathcal{V}$ que contiene a todas las cuerdas que pertenecen a algún $\mathcal{W}(u,t)$.

Sobre $\mathcal{V}$ se define una relación de equivalencia ρ que identifica a cada cuerda V con su inversa V^{-1}. Sobre $\mathcal{W}$ se define la relación de equivalencia ρ' que identifica a cada cuerda $V = c_n \dots c_1$ en $\mathcal{W}$ con las cuerdas que permutan ciclicamente $V_{(i)} = c_i c_{i-1} \cdots c_1 c_n \dots c_{i+1}$ y sus inversas $V_{(i)}^{-1}$, $1 \leq i \leq n$.

$\mathcal{V}'$ es el conjunto completo de representantes de $\mathcal{V}$ relativo a ρ y $\mathcal{W}'$ es el conjunto completo de representantes de $\mathcal{W}$ relativo a ρ'.

Definición 27 Para cada cuerda $V = c_n \dots c_1$ ó $V = 1_{(u,t)}$ se define una representación $M(V)$ del carcaj C como sigue: Sea $u(i) = t(c_{i+1})$, $0 \leq i < n$ y $u(n) = s(V)$. Dado un vértice v en C_0 definimos $I_v = \{i | u(i) = v\} \subseteq \{0, 1, \dots, n\}$. $M(V)_v$ es un espacio vectorial, su dimensión es la cardinalidad de I_v y con z_i para $i \in I_v$ denotamos los vectores base de $M(V)_v$. Para toda $1 \leq i \leq n$ tenemos con $c_i = \beta$ una flecha $\beta(z_{i-1}) = z_i$, para

$c_i = \beta^{-1}$ la flecha invers definimos $\beta(z_i) = z_{i-1},$. Sea $\gamma : w \to w'$ una flecha y para algún $j \in I_w$ es z_j un vector base de $M(V)_w$. Si $\gamma(z_j)$ no se ha definido en el proceso anterior entonces definimos $\gamma(z_j) = 0$. Obtenemos una representación $M(V)$ de C que satisface las relaciones en R, este $M(V)$, la llamamos *módulo de cuerda.*

Nota: $M(V)$ y $M(V^{-1})$ son representaciones isomorfas y $M(1_{(u,t)})$ es la representación simple del vértice u.

Definición 28 Sea $V = c_n \dots c_1$ una cuerda en $\mathcal{W}$, Z un espacio vectorial y φ un automorfismo en Z. Se define la representación $M(V, \varphi)$ de C como sigue. Dado un vértices v en C definimos $I_v = \{i | s(c_{i+1}) = v\} \subseteq \{0, 1, \dots, n-1\}$ con $|I_v| = n$. Para toda $v \in C_0$ se escribe con $M(V, \varphi)_v$ la suma directa $\bigoplus_{i \in I_V} Z_i$ con Z_i copias de Z. Para $c_n = \beta$ una flecha se define la acción de β en Z_{n-1} mandando $z \in Z_{n-1}$ a $\varphi(z) \in Z_n$; para $c_n = \beta^{-1}$ mandamos $z \in Z_n$ a $\varphi^{-1}(z) \in Z_{n-1}$, i.e. la flecha c_n es la multiplicación por el bloque de Jordan $J_n(\nu)$ con ν el valor propio. Las demás flechas se interpretan como morfismo identidad. Si para una flecha $\gamma : w \to w'$ la acción de γ sobre algún $Z_j \subseteq M(V, \varphi)_w$ no está definida entonces definimos $\gamma | Z_j$ como el mapeo cero. Se obtiene la representación $M(V, \varphi)$ de C que satisface las relaciones en R, la llamamos *módulo de banda.*

Teorema 5 [BuRi] *Los módulos de cuerda $M(V)$ y los módulos de banda $M(V, \varphi)$ forman un sistema completo e irreducible de representaciones de las clases de isomorfía de los módulos inescindibles del álgebra de cuerda $kC/ < R >$.*

3.4 Homomorfismos entre módulos de cuerda o de banda

Un carcaj se llama *árbol* o *cíclico* si es un carcaj finito cuyo gráfica es simplemente conexo o $\tilde{A}_n$, respectivamente.

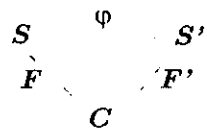
Definición 29 Sea S un árbol o cíclico. Una *bobina* es un homomorfismo de carcajes $F : S \to C$ con las siguientes condiciones:

1) F es inyectivo en las fuentes y los pozos, i.e. en S no existe ningún subcarcaj con $F(\alpha) = F(\beta)$ de las siguientes formas $\cdot \xleftarrow{\alpha} \cdot \xrightarrow{\beta} \cdot$ o $\cdot \xrightarrow{\alpha} \cdot \xleftarrow{\beta} \cdot$ en S.

2) Si S es cíclico F no es periódico, i.e. no existe un automorfismo $\sigma \neq id_S$ con $F\sigma = F$.

Un bobina se llama *cíclica* si S es cíclico.

Un *morfísmo entre bobinas* es un triángulo conmutativo como se muestra en la Figura 1 con φ un homomorfismo de carcaj.

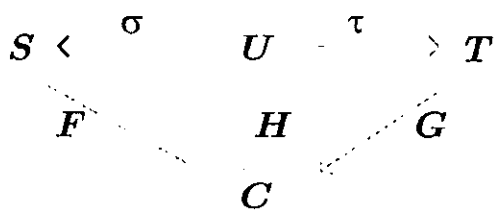


1. Morfismo entre Bobinas

Sean $X, Y \in mod(C, I)$ módulos de cuerda o de banda constituidos por las cuerdas S y T (en el caso de los módulos de banda se denotan con ν y μ los valores propios

en los bloques de Jordan $J_n(\nu)$ y $J_m(\mu)$, respectivamente). Se tiene una descripción de $\mathrm{Hom}(X, Y)$ basado en las bobinas $F : S \to C$ y $G : T \to C$.

Vamos a construir una conexión entre las bobinas F y G a través de una terna (U, σ, τ) junto con $H : U \to C$ una bobina donde $\sigma : U \to S$ y $\tau : U \to T$ son morfismos de bobinas, de tal manera que el siguiente diagrama sea conmutativo:



2. Conexión entre Bobinas

Sea (U, σ, τ) la conexión construida y (U', σ', τ') otra conexión entre F y G con $H' : U' \to C$, $\sigma' : U' \to S$ y $\tau' : U' \to T$. Un morfismo entre (U, σ, τ) y (U', σ', τ') es un morfismo de bobinas $\iota : U' \to U$ con $\sigma' = \sigma\iota$ y $\tau' = \tau\iota$. Esto es un isomorfismo si ι es un isomorfismo, si ι es una inclusión se define $(U', \sigma', \tau') \leq (U, \sigma, \tau)$.

Para esas ternas denotamos con $\mathfrak{U} = \mathfrak{U}(F, G)$ el conjunto completo de representantes de clases de isomorfía, como la relación $\leq$ induce un órden parcial sobre $\mathfrak{U}$, también lo induce sobre cualquier subconjunto de $\mathfrak{U}$.

Un elemento (U, σ, τ) se llama *cíclico* si U es cíclico

Definición 30 Una terna (U, σ, τ) se llama *admisible* si satisface los siguientes condiciones:

1) Si x es un vértice en U y β una flecha en S que termina en $\sigma(x)$ entonces existe una flecha $\alpha \in U$ que termina en x con $\sigma(\alpha) = \beta$.

2) Si x es un vértice en U y β es una flecha en T que empieza en $\tau(x)$ entonces existe una flecha $\alpha \in U$ que empieza en z con $\tau(x) = \beta$.

Definición 31 Sea (i, j) una pareja de vértices en S y T, respectivamente. Definimos $\mathfrak{U}_{i,j} = \{(U, \sigma, \tau) | \textit{ existe un vértice } x \textit{ en } U \textit{ con } \sigma(x) = i, \tau(x) = j\}$.

Lema 10 Existe a lo más una terna admisible en $\mathfrak{U}_{i,j}$ dada una pareja (i,j) de vértices en S y T, respectivamente.

Sea a $= (U, \sigma, \tau)$ admisible, se le asocia un espacio H_{a} de mapeos k-lineales como sigue:

$$H_{\mathsf{a}} = \begin{cases} \mathrm{Hom}_k(k^n,k^m), & \text{si a no es cíclico} \\ \{f \in \mathrm{Hom}_k(k^n,k^m) | f = (*)\}, & \text{si a es cíclico y } \nu{=}\mu \\ 0, & \text{si a es cíclico y } \nu \neq \mu \end{cases}$$

$$(*) = \begin{pmatrix} 0 & \cdots & 0 & \kappa_r & \cdots & \kappa_2 & \kappa_1 \\ \vdots & & & & \ddots & & \kappa_2 \\ \vdots & & & & & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & \kappa_r \end{pmatrix} \quad \text{ó} \quad (*) = \begin{pmatrix} \kappa_r & \cdots & \kappa_2 & \kappa_1 \\ 0 & \ddots & & \kappa_2 \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \kappa_r \\ \vdots & & & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & \cdots & 0 \end{pmatrix}$$

con $\kappa_i \in k$, $1 \leq i \leq r$ y $r = min(n, m)$.

Teorema 6 [Kr] *Sea $A = kC/ < R >$ un álgebra de cuerda y X, Y (con cuerdas F y*

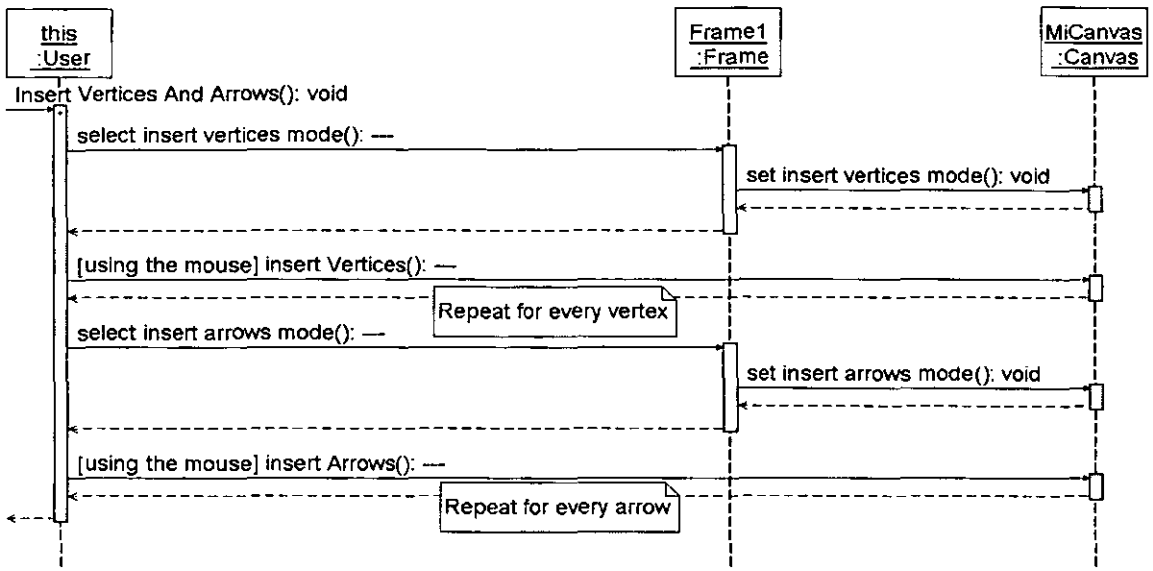
G, respectivamente) módulos de cuerda o de banda. Si $A = A(F, G)$ *denota el conjunto de todos las ternas admisibles en* $U(F, G)$ *entonces existe un isomorfismo entre Hom*(X, Y) *y* $\bigoplus_{a \in \mathcal{A}} H_a$.

Capítulo 4
Funcionamiento del programa

Este capítulo trata detalles del uso del programa. Cuando hablamos de un proyecto se refiere a todo el conjunto, el álgebra, representada por sus vértices, sus flechas y sus relaciones, las cuerdas y los homomorfismos. Para mayor entendimiento se usan diagramas de secuencia. Estos diagramas forman parte de UML (Unified Modelling Language) que es un lenguaje de modelos que sirve como herramienta de comunicación entre el programador y el usuario.

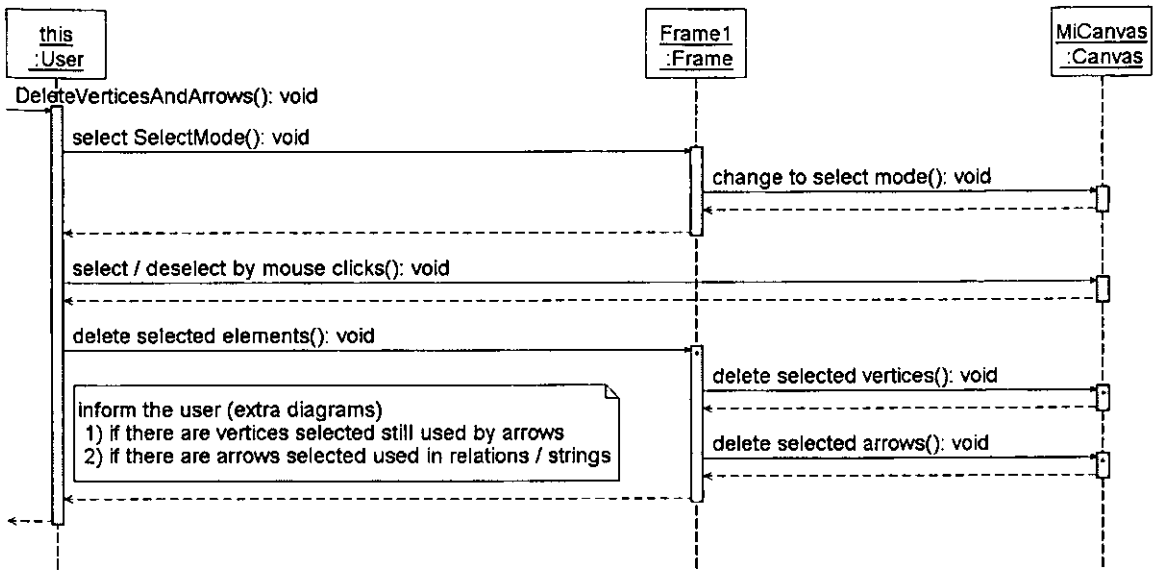
En los diagramas de secuencia los objetos están ordenados horizontalmente, aunque el orden no tiene importancia. Los objetos tienen los mismo nombres que en el código, con dos puntos [:] separando el nombre de la clase. En el eje vertical se expresa el tiempo. Las flechas tienen el nombre del mensaje y van del objeto transmisor al objeto receptor; los objetos pueden mandar mensajes a sí mismos. Opcionalmente el mensaje puede tener una condición entre corchetes [ ]. Las barras aclaran la duración de una acción, además se puede ver si un objeto es activado, está actuando o esperando. Las casillas con un doblez en la parte derecha superior contienen explicaciones adicionales.

Abriendo el programa el usuario encuentra una ventana con un menú en la parte de arriba y un canvas abajo.Este canvas está en el modo para insertar vértices, e.d. el usuario puede insertar vértices directamente presionando el botón izquierdo del ratón, vea detalles en Figura 3. Los demás elementos, tanto las flechas como los vértices, se puede insertar según el modo que seleciona bajo "Algebra" e "Insert".



3. Inserción de Elementos

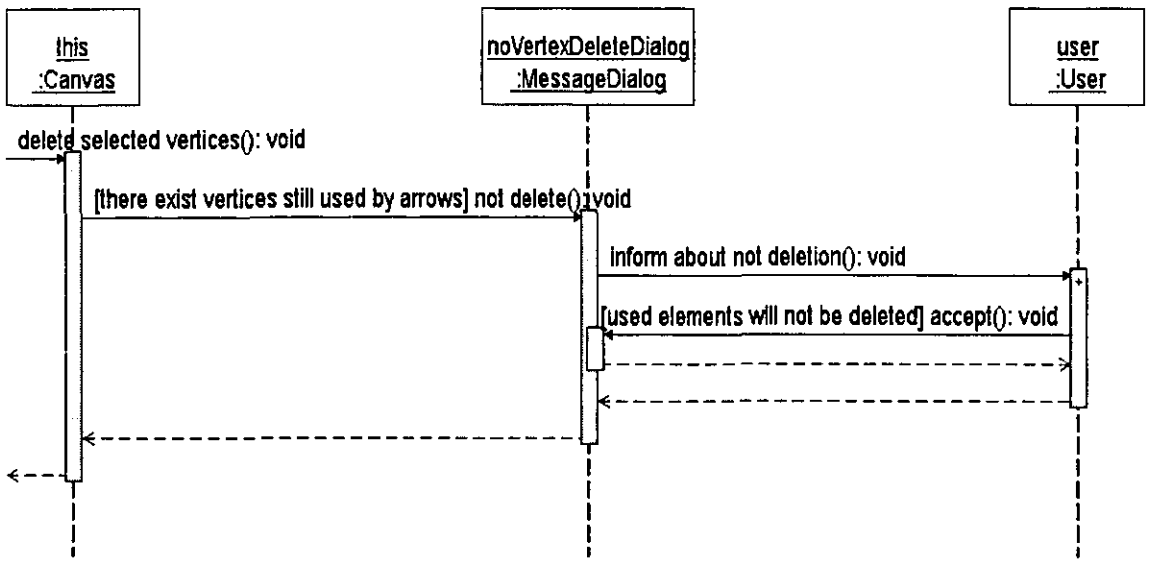
Para eliminar un vertice ó una flecha, primero hay que selecionarlo para posteriormente borrarlo. En la Figura 4 se visualiza el proceso de eliminar elementos del álgebra.



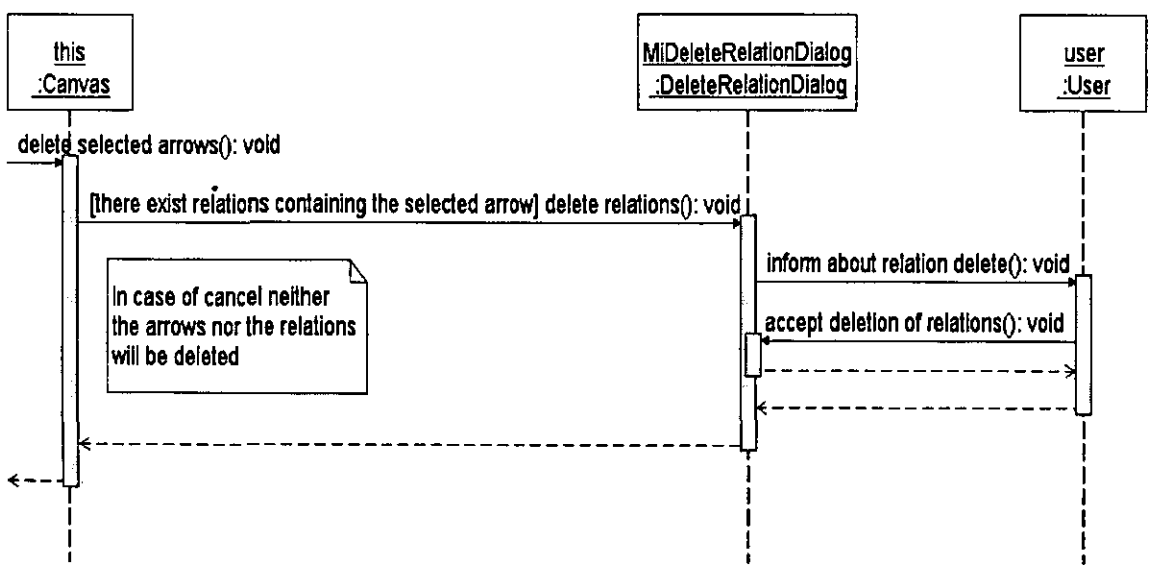
4. Eliminación de Elementos

Las Figuras 5 y 6 muestran los casos cuando hay conflictos entre elementos seleccionados para eliminarlos y elementos que dependen de ellos.

En el primer caso (Figura 5) se intenta borrar un vértice que todavía forma parte de una flecha.



5. Eliminación de Vértices



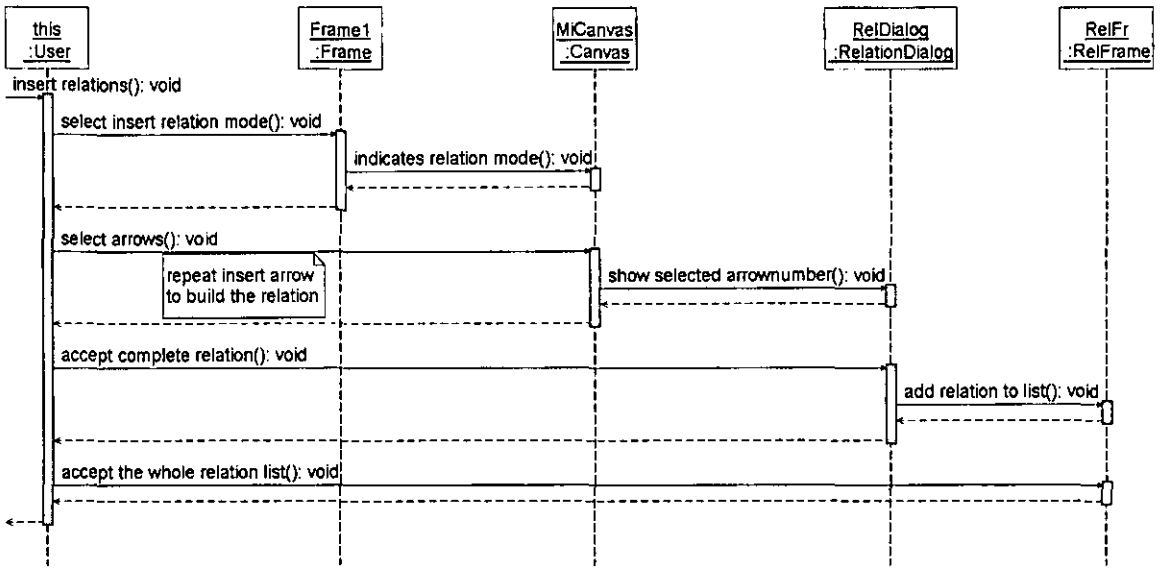
6. Eliminación de Flechas

En la Figura 6 la flecha seleccionada forma parte de una relación.

En caso de que alguna cuerda contenga flechas que están selecionadas para eliminarse, el procedimiento es similar al anterior.

Se puede acomodar la gráfica, escogiendo en "Algebra" la opción "Move Vertex", y ahora con el botón izquierdo del ratón se pueden acomodar los vértices arrastrándolos.

Para insertar relaciones el usuario escoge en "Relations" el punto "Show & Insert" y inserta las relaciones como en la Figura 7.



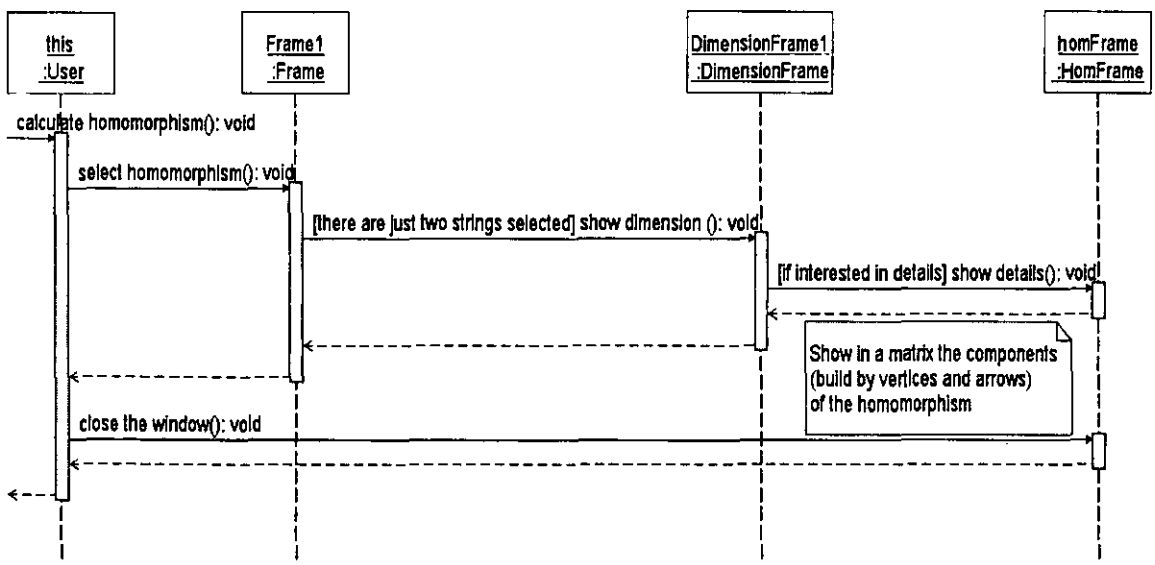
7. Inserción de Relaciones

Acceptando la primera relación se abre una nueva ventana; en esta misma ventana ("RelFr") se pueden escoger relaciones para editarlas o removerlas. Con el botón "RemoveAll" se eliminan todas las relaciones.

Escogiendo la opción "Stringalgebra" del menú "Algebra", el programa verifica si el álgebra construida con los vértices, flechas y relaciones cumple con las condiciones de álgebras de cuerda.

Las cuerdas se insertan y modifican de manera similar que las relaciones, eligiendo la opción "Strings" en el menú principal.

En la opción "Homomorphisms", "Show Dimension" el programa calcula la dimensión de la base del homomorfismo entre dos cuerdas. Si hay más de dos cuerdas, el programa ofrece al usuario que escoja de una lista dos cuerdas, para calcular el homomorfismo entre ellas. En la Figura 8 se muestra cómo el usuario puede ver la dimensión de un espacio de homomorfismos entre dos cuerdas, y si desea más detalles de los componentes de la base de este espacio.



8. Homomorfismos entre Cuerdas

Los homomorfismos no son persistentes, se calculan cada vez que lo pida el usuario.

Los proyectos se pueden guardar y abrir en "File" como de costumbre.

Capítulo 5
Detalles de la programación

En este capítulo se explican con más detalle algunos de los algoritmos que destacan en este programa y decisiones de diseño que se tomaron para la interfaz de usuario.

5.1 Elección de Java como lenguaje anfitrión

En un intento anterior se hizo el programa en C. Eso incluyó dos desventajas grandes: escasez de portabilidad y falta de una interfaz de usuario. En Java fue posible generar el programa en una PC y correrlo principalmente en estaciones de trabajo con sistema operativo Solaris o Linux. Además en vez de usar un entorno de línea de comandos como en C, se construyó una interfaz gráfica para la comodidad del usuario.

5.2 Diseño de la interfaz del usuario

La interfaz del usuario se construyó según los algoritmos que los matemáticos usan para resolver el problema con papel y lapiz.

5.3 Algoritmos

5.3.1 Búsqueda de caminos infinitos

La última condición de un álgebra de cuerda **Definición 25** afirma que no debe de haber caminos con longitud infinita. Es suficiente considerar los caminos que empiecen en todos los vértices, ya que los que terminan están incluidos por consideraciones de dualidad.

En el primer paso buscamos si una flecha ya fue visitada; si no, buscamos el camino empezando con esta flecha y en cada paso buscamos si hay una relacion de longitud dos y elegimos así un único camino. Hay dos posibilidades para el camino:

- El camino no contiene ciclos y termina en una flecha fija. Entonces marcamos todas las flechas que constituyen el camino como visitadas.
- El camino contiene un ciclo o dos (que se pueden "desdoblar" y de esta manera se entiende como un ciclo). Después de haber pasado por él regresamos con la misma flecha. Entonces hay que buscar si alguna de las relaciones con longitud mayor que 2 empieza en alguna flecha del ciclo. Como aseguramos construyendo las relaciones que ninguna relación contiene alguna otra, cualquier relación que empieza en alguna de las flechas del ciclo tiene que estar sobre el ciclo.

En el caso que existiera un ciclo y no contiene ninguna relación, la última condición para una álgebra de cuerda no se cumplira y no se trata de un álgebra de cuerda. En cualquier otro caso es una álgebra de cuerda.

Veamos el pseudocódgo:

Pseudocódigo

```
clase Algebra {
    . . .
    public boolean cond3()
    . . .
}
```

La clase Algebra contiene las diferentes condiciones con las que tiene que cumplir una álgebra para comprobar si es un álgebra de cuerda.

Veamos con más detalle la condición 3:

Pseudocódigo

```
public boolean cond3()
{
    boolean isCond3 = true;  /* supongamos que se cumple la condición
    /* Vacía el vector que contiene todos las flechas de un camino */
    visitedArrs.removeAllElements();
    for int i=0...sizeOfArrows { /* busca en todas las flechas */
        boolean infinitePath = false; /* supongamos que no sea infinito el camino */
            if (todavía no visitamos la flecha) {
              /* busca si camino empezando en flecha i es infinito */
                  infinitePath = pathInfinite(flecha i);
        }
            if (camino es infinito) {
              isCond3 = false;  /* no se cumple la cond3 */
        }
      }
      return isCond3;
}
```

Consideremos el método pathInfinite (para una flecha) con más detalle. Si ya visitamos la flecha s, nos encontramos en un círculo. Para verificar que no sean infinitas las vueltas en el círculo, basta con que encontremos una relación más larga que 2 que empiece en alguna flecha del camino.

Si es la primera vez que pasamos por esta flecha, la marcamos como visitada, la añadimos al camino y buscamos si forma una relación con alguna de las siguientes flechas

que empiezan en su vértice final. Si existe una siguiente flecha que no forma una relación con la flecha s, aplicamos el método pathInfinite a esta flecha. Seguimos, hasta que ya no haya flecha, o encontremos un círculo infinito.

Pseudocódigo

```
boolean pathInfinite(flecha s) {
      /* primero busca, si ya se visitó la flecha s */
      if (ya se visitó la flecha s) { /* estamos en un círculo */
            infinite = true; /* supongamos que sea infinito */
            for int i=0...sizeOfFlechasVisitadas{ /* busca entre las flechas visitadas */
                  /* si alguna es flecha inicial para una relación con longitud mayor que 2 */
                  for int j=0...sizeOfRelacionesMayorQue2{
                        if(el primer elemento de la relación es flecha i) {
                              infinite = false; /* no es infinito */
                        return infinite;
                  }
               }
          }
            return infinite; /* el camino es infinito */
       }
     else{
            s.seVisito(true); /* pongamos flecha s como visitada */
            visitedArrs.add(fs); /* añande la f1 al vector del camino */
            /* buscamos la flecha f1 que empieza en el final de s */
            if(flechas f1 y s forman una relación) {
                  /* busca otra flecha f2 que empieza en el final de s */
                  if (flechas f2 y s no forman una relación) {
                        pathInfinite(f2); /* iteración busca camino en f2*/
                }
                  else{ /* f1 y s no forman una relación */
                        pathInfinite(f1); /* iteración que busca camino en f1*/
                  }
          }
     }
       return infinite;
}
```

5.3.2 La dimensión del espacio de homomorfismos entre dos módulos

Por el **Teorema 8** los componentes de la base del homomorfismo entre dos módulos se pueden calcular con el siguiente algoritmo. Cada paso tiene una explicación y después sigue el pseudocódigo. Como los nombres de las clases concuerdan con los nombres en el código en Java se pueden ver detalles en el código.

Si se busca el homomorfismo entre la cuerda str1 y la cuerda str2, se forma una matriz M($n \times m$) con n=(número de flechas de str2 + 1), y m=(número de flechas de str1 + 1).

- Primero, insertamos los vértices en M, después las flechas que conectan algunos vértices.
- Por último se determina cuáles de estos componentes forman parte de la base del homomorfismo.

Pseudocódigo

```
clase HomMatrix {
    Vector vertices; /* la matriz de los vértices */
    Vector arrows; /* el vector de las flechas */
    HomMatriz(Vector str1, Vector str2); /* construye la matriz dependiendo de las dos cuerdas */
public void searchVertices() { /* busca los vértices en la matriz */
. . .
}
public void searchArrows() { /* bucsca las flechas en la matriz */
. . .
}
public void setComponents() { /* determina las componentes */
. . .
}
}
```

Consideramos los métodos en detalle. En el método searchVertices se comparan los vertices iniciales de las flecha i en str2 y de la flecha j en str1. Si los vértices son iguales,

en (i, j) en la matriz de los homomorfismos el vértice es visible como punto.

Pseudocódigo

```
searchVertices() {
Vertex sv1;
Vertex sv2;
    for int i=0...sizeOfStr2{
        for int j=0...sizeOfStr1{
        /* Se toma los vértices de inicio de cada flecha en str1 */
        Arrow start1 = string1.elementAt(j);
         /* busca el vértice inicial dependiendo si se trata de una flecha directa o inversa */
            sv1= searchStartVertex(start1);
        /*los mismo con sv2 en str2*/
            . . .
        fillMatrix(i,j,sv2,sv1); /* compara los vértices e insértalos en la matriz*/
        . . .
        }
    }
```

Para el último renglón y la última columna de la matriz, hay que considerar vértices finales, pero el algoritmo funciona igual.

En el método fillMatrix se genera un MatrixVertex mv, que es un punto con diferentes propiedades, como son el si los vértices concuerdan, un valor boolean que indica si mv es visible en la matriz de los homomorfismos o no. Al final se añade mv a la matriz vértices, definidos arriba.

Pseudocódigo

```
fillMatrix(int i1, int i2, Vertex s1, Vertex s2) {
    MatrixVertex mv; /* genera un MatrixVertex */
    /* determina si los vértices s1 y s2 son iguales*/
    if (s1.equals(s2)) {
        mv.setInMatriz(true); /* mv es visible */
    }
    else{
        mv.setInMatriz(false); /* mv no es visible */
    }
    vertices.addElement(mv); /* añade mv a la matriz vértices */
```

}

Similarmente se determinan las flechas en la matriz de homomorfismos.

Buscamos flechas que concuerdan y comparamos sus direcciones, que pueden ser los dos directos, los dos inversos, o uno inverso y el otro directo. Conforme generamos los MatrixArrow con su dirección, dependiendo de las instrucciones anteriores, lo añadimos al Vector arrows. Este vector contiene solamente flechas visibles. Al final determinamos vértices (MatrixVertex) iniciales y finales de esta flecha y indicamos en estos vértices que esta flecha inicia y termina, respectivamente alli.

Pseudocódigo

```
searchArrows() {
    for int i=0...sizeOfStr2{
        for int j=0...sizeOfStr1{
        Arrow arr1 = string1.elementAt(j);
        Arrow arr2 = string2.elementAt(i);
        if (arr1.equals(arr2) ){ /* busca si son iguales */
            /* entero que indica la combinación de las direcciones*/
            int matArrDir;
            /* determina las direcciones y así matArrDir */
            if(arr1.getSign() && arr2.getSign() ){
                matArrDir = 1;
            }
            if ...{ /* los demás casos similares */
            ... }
            MatrixArrow ma; /* genera un MatrixArrow */
            ma.setInMatrix(true); /* es visible */
            arrows.addElement(ma); /* añadelo a arrows*/
            /* según las direcciones busca los vértices inicial */
            if(matArrDir = 1) {
                MatrixVertex msVer = searchMatrixVertexAt(i,j);
                /* y terminal de la flecha */
                MatrixVertex mtVer = searchMatrixVertexAt(i+1, j+1);
                /* Indica en los MatrixVertex ma que es la flecha iniciando */
                msVer.setStartArrow(ma);
                /* respectivamente como flecha terminando */
                mtVer.setEndArrow(ma);
```

```
                }
                /* los demás casos similares */
                ...
            }
        }
    }
```

Por último determinamos los componentes del homomorfismo. Un componente se constituye de un solo vértice o varios vértices conectados con flechas como las determinamos arriba.

Buscamos el primer vértice v1 visible de la matriz; si sale o termina una flecha en v1 determinamos el vértice en el otro extremo de la flecha y buscamos si existe otra flecha, hasta que ya no hay otra más. A estos vértices y flechas los marcamos como ya visitados (i.e. ya no se consideran otra vez en el algoritmo) y forman una componente. Al final verificamos si esta componente está en el espacio del homomorfismo, y si es así cambiamos el color de sus elementos a azul.

Una componente está en el espacio del homomorfismo si cumple con las siguiente condiciones:

- Para el vértice inicial del componente que está en M(i, j) en la matriz, se verifica que la flecha j en str1 (i.e. elementAt(j) en str1) sea directa y la flecha j-1 sea inversa, si existe.

- Para el vértice final en M(k ,l) se verifica que la flecha k en str2 sea inversa y la flecha k-1 sea directa, si existe.

Pseudocódigo

```
setComponents() {
    Vector component; /* contiene los componentes del homomorfismo */
    for int i=0...sizeOfVertices {
        MatrixVertex visVer = vertices.elementAt(i);
        if (visVer está en la matriz y todavía no ha sido visitado) {
            Vector path = searchPath(visVer); /* Busca el camino empezando en visVer */
            if (startComponent(path) y endComponent(path) ){
                for int j=0...sizeOfPath { /* cambia los elementos de path a azul*/
                    elementAt(j).setColor(blue);
                    /* y añade path al vector components */
                }
            }
            for int j=0...sizeOfPath { /* en todo caso cambia los elementos a visitado*/
                elementAt(j).visit(true);
            }
        }
    }
}
```

Contando el número de componentes en el espacio del homomorfismo se calcula su dimensión y se termina el algoritmo.

Capítulo 6
Trabajos a futuro

6.1 Posibles modificaciones en el programa

6.1.1 Cambios dentro del programa

Como el programa es portatil dependiendo del fondo de la pantalla, el usuario pudiera desear cambiar los colores. En el futuro se podría agregar una opción para escoger los colores tanto del fondo del programa (o más bien del canvas en el cuál se insertan los vértices y las flechas) como de los diferentes estados de los elementos. Eventulamente se podría ofrecer al usuario la posibilidad de insertar el límite en el número de vértices o flechas. Eso podría aparecer en alguna ventana al principio o como una opción en el menú, p.ej. bajo la opción "Algebra". Otro elemento útil para el usuario podría ser un archivo de texto en el cual el usuario puede anexar notas al proyecto. Por último, una opción interesante podría ser generar todas las cuerdas con cierta longitud para una álgebra dada con sus relaciones.

6.1.2 Cambios relacionados con otros programas

Tomando en cuenta otros programas relacionados con teoría de representaciones, se podría hacer una conexión hacia otros programas o paquetes. Un ejemplo interesante podría ser conectarlo con CREP (Combinatorial REPresentation Theory), un paquete de pro-

gramas para trabajar con clases particulares de álgebras de dimensión finita o estructuras relacionadas, respectivamente. Además cubre aspectos combinatorios de sus representaciones de dimensión finita. Este programa que ofrece como interfaz de usuario el ambiente MAPLE, fue desarollado por un grupo de investigadores de teoría de representaciones de álgebras en Bielefeld/Alemania. Se podría extender el programa "Stringalgebra" del presente trabajo hacia MAPLE o extender su interfaz gráfica para usarla con CREP.

6.2 Uso del programa como API

Como Java es un lenguaje orientado a objetos, el programa presente nos permite reusar y extender el conjunto de clases que nos ofrece como ambiente de desarollo (API). La encapsulación nos permite hacer cambios y extenciones sin meternos al código, ya que nos basta conocer el funcionamiento de las clases y sus métodos.

Para un futuro proyecto que requiera p.ej un canvas para insertar vértices y flechas, el programador puede usar el ambiente presentado y extenderlo añadiendo nuevos métodos a las clases generadas. En la documentación (javadoc) del programa, anexada en el disco, se encuentran las clases con sus métodos en todo detalle como lo documentamos en el código.

Tenemos la esperanza que este trabajo sirva no solamente como herramienta a interesados en el área de teoría de representaciones de álgebras, sino tambíen a programadores de API en sus futuros proyectos.

Apéndice A
Puntos básicos de la programación

Java es el lenguaje de programación más usado en este tiempo debido a las cosas que es posible hacer en él. Puede añadirse animación a una página Web, escribir juegos y utilerías, diseñar software y crear programas que presenten una interfaz gráfica de usuario. En este trabajo se aprovechó mucho de esto último.

Java es un lenguaje interpretado: el compilador de Java genera código en bytes (byte-codes) para la maquina virtual de Java(JVM Java Virtual Machine), en vez de código que depende de la máquina (native machine code). Un programa de Java corre, ejecutando el intérprete de Java que executa el código compilado en bytes.

Como los códigos de bytes de Java son independientes de la plataforma, un programa de Java corre en cualquier plataforma que soporta JVM. Eso es importante especialmente para aplicaciones distribuidas por el Internet. En el trabajo la intención es que el programa corra tanto en PC´s como en estaciones de trabajo de Unix, así que resulta muy útil la neutralidad de la arquitectura.

A.1 Programación orientada objetos

En la programación procedural se aplica a un problema una sucesión de instrucciones que son ejecutadas según una regla definida. El modelo detrás de esto es el *la programación lineal* que esta basada en algoritmos. Las instrucciones que van juntas se agrupan en *funciones* ó *procedimientos*. La estructura de los datos y las instrucciones están separados y

van en diferentes niveles en la jerarquía del programa. Una gran desventaja está en mantenerlo. Un pequeño cambio en los datos puede causar muchos cambios en diferentes partes del programa.

Por otro lado se encuentra la *programación orientada a objetos* . Instrucciones y datos correspondientes forman un entidad independiente. Eso son los objetos.

Los *Objetos* consisten en general de dos partes: *datos* y *métodos*.

Los datos son las propiedades de un objeto, mientras que los métodos representan a los elementos operacionales (como antes las funciones y procedimientos) al nivel orientado a objetos y realizan la funcionalidad de un objeto. Se supone que los datos de un objeto son invisibles, cada interacción entre objetos se realiza mediante los métodos. Este proceso de juntar los datos de un objeto con sus métodos se llama *encapsulación*. Los beneficios de la encapsulación son, como mencionamos arriba, la invisibilidad de los datos, pues así estarán protegidos contra cambios no deseados desde afuera. Por otro lado garantizamos *modularidad*, que significa que a cada objeto se pueden agregar cambios, independientemente de los demás objetos. Cuando se requiere otra función adicional para cierto objeto, nada más se escribe otro método en la clase correspondiente.

Objetos similares se encuentran juntos para facilitar su clasificación. Se guardan sus propiedades en *clases* para generar después objetos reales. Cada clase puede tener muchas subclases y muchos objetos. Una *subclase* tiene las mismas propiedades de la llamada *superclase* aparte de otras propiedades adicionales. Cada objeto en una subclase hereda los atributos y métodos del objeto correspondiente en la superclase.

La *herencia* permite que se desarollen diversas clases relacionadas sin que hagan trabajo redundante. Las subclases o clases derivadas adoptan los atributos y los métodos de su superclase y, en general, aplican ciertos cambios y elementos adicionales. Cuando un método está definido en una subclase y en su superclase, se usa el método de la subclase. Esto permite que una subclase cambie, reemplace o elimine por completo algunos de los métodos o atributos de sus superclases. La creación de un nuevo método en una subclase para cambiar el método heredado de una superclase se conoce como *sobreposición del método* (overriding). Se necesita sobreponer un método en el caso de que el método heredado produzca un resultado no deseado.

Otro de los argumentos propios de la programación orientada a objetos se conoce como *polimorfismo* (polymorphism) . Esto corresponda a la propiedad de una variable que le permite estar a disposición para objetos de diferentes clases.

La *ligadura* (Binding) une la llamada a un método con su archivo fuente. En lenguajes de programación no orientadas a objetos el compilador (compiler) o el ligador (linker) generan esta unión estáticamente, así que ya no hay cambios de la ligadura a la hora de ejecutar el programa. Debido al polimorfismo en lenguajes de programación orientados a objetos puede usarse una variable para objetos de diferente clases, por eso hay que *ligar dinámicamente* el programa, i.e. en momento de ejecución del programa

Con polimorfismo y ligadura dinámica se evita el manejo de diferentes casos (con if-else) y teniendo un archivo fuente más claro se facilita extender el programa .

A.2 Orientación a Objetos en Java

Java es un lenguaje de programación orientada a objetos. Para dearollar programas en Java se usa el JDK (Java Development Kit), un paquete de software que incluye el interpréte, clases de Java y herramientas para el desarollo: el compilador, el depurador (debugger), el appletviewer, etc.

La interfaz de la programación de Java (Java API Application Programming Interface) que se distribuye con el JDK, consiste de diferentes clases e interfaces que se usan para programar en Java.

A un grupo de clases e interfaces tambíen se llama *paquete* (Packages). Los paquetes son análogos a las bibliotecas de muchas otros lenguajes de programación. El Java API mismo está implementado como grupo de paquetes. Otros ejemplos son: una clase que crea componentes para una interfaz gráfica de usuario (GUI Graphical User Interface) el java.awt package, clases que manejan entada y salida (java.io package), y clases que soportan la funcionalidad para trabajar en redes (networking) en el java.net package. La clase Object (en el paquete java.lang package) sirve como raíz de la jerarquía de clases en Java.

Con *serializasión* se pueden hacer persistentes los objetos, se logra así en este trabajo que las álgebras elaborados por el usuario pueden estar guardadas en archivos locales de la computadora del usuario.

Seguridad fue un factor muy importante en el desarollo de Java. El modelo de seguridad de Java protege en especial tanto el disco duro y la memoria principal del cliente como el sistema operativo y el estado de los programas contra ataques del siguiente tipo:

- Daño del software en la computadora del cliente.
- Robo de información de la computadora del cliente.
- Toma de la identidad de la máquina del cliente.
- Uso de la computdadora del cliente desde afuera, por ejemplo para correr un proceso en la máquina del cliente y así occupar memoria principal.

Hay diferentes mecanismos de seguridad que impiden que los applets causen daños durante su ejecución. Por ejemplo no se permite a un applet corriendo en un navegador ejecutar operaciones sobre datos de la computadora local. Esa fue una de las razones para implementar este trabajo como aplicación. Se requiere salvar y cargar datos insertados por el usuario.

En algunos lenguajes de programación orientada a objetos una clase puede tener más de una superclase de la cuál herede métodos y métodos adicionales, pero no en Java. En Java se resuelve este problema con interfaces. Una *interfaz* (interface) es un conjunto de métodos abstractos, i.e. métodos que todavía no están implementados, y constantes. La clase implementa la interfaz concretizando sus métodos. Objetos de la clase tienen los métodos del interfaz definidos en la clase.

Java permite *sobrecargar* (overloading) sus métodos. Eso significa que se pueden crear en una misma clase diferentes métodos con el mismo nombre que solamente difieren

en los parámetros. Un ejemplo en este trabajo son los constructores en la clase Vertex.

```
   public Vertex() {
   this.x1 = 0; this.y1 = 0;
...
   }
   public Vertex(int a1, int b1) {
   this.x1 = a1; this.y1 = b1;
   ...
   }
   public Vertex(Vertex v1) {
   this.x1 = v1.x1; this.y1 = v1.y1;
...
   }
```

En el primer caso se construye un vértice vacío con x1=y1=0, en el segundo caso se construye un vértice con dos coordenadas enteras x1=a1, y y1=b1; en el último caso las dos coordenadas son las coordenadas de v1.

Una de las características más sofisticadas del lenguaje Java es su habilidad para escribir programas que pueden hacer multitareas. En java, cada una de las tareas simultáneas que maneja la computadora se cononce como *hilo de ejecución* (Thread), y el procesamiento general se conoce como *hilos de ejecución múltiple* (Multithreading). Los hilos de ejecución múltiple son útiles en la animación y en muchos otros programas.

Java ofrece con las *excepciones* (Exceptions) una posibilidad para manejar errores y excepciones en un programa. Una excepción está generada por el programa o la JVM. Las excepciones generada por la JVM son tratadas directamente por Java, como por ejemplo quedarse sin memoria (OutOfMemoryError). En cambio las excepciones del programa son tratadas por condiciones dadas por el programador, como muestra el siguiente ejemplo

del trabajo tomada de la clase RelFrame:

```
public RelFrame(){
try {
jbInit();
}
catch (Exception e){
e.printStackTrace();
}
}
private void jbInit() throws Exception{
...
}
```

throws Exception en el método jbInit() indica que este método puede tirar una advertencia. El constructor contiene el manejo de excepciones (exception handling). La construcción

```
try
{...
}
catch (Exception e)
{...
}
```

trata la excepción. Se supone que se ejecuta el método jbInit() en el bloque try y solamente en caso de surgiera algún problema lo recibe catch y manda con e.printStackTrace() información del estado del stack.

A.3 Funcionamiento de los programas Java

Una distinción que hay que hacer en la programación Java es en dónde se supone que se ejecutará el programa. Se pretende que algunos programas trabajen en la computadora

propia, mediante un comando o al hacer clic en un icono para ejecutarlos. Se pretende que otros programas se ejecuten como una parte de World Wide Web. Los programas Java que se ejecuten en la computadora propia son llamados *aplicaciones* (applications). Los programas que se ejecutan en páginas Web son se llaman *applets*.

A.3.1 Las Aplicaciones

Aunque Java ha llegado a ser muy conocido debido a que puede usarse junto con páginas de World Wide Web, también se le puede usar para escribir cualquier tipo de programa de computación.

A.3.2 Los Applets

A diferencia dc las aplicaciones, los applets Java compilados no pueden probarse usando el intérprete de `java`. Hay que ponerlos en un página de Web y verla de alguna de dos formas:

- Mediante un navegador Web que pueda manejar applets Java, tal como Netscape Navigator Microsoft Internet Explorer.
- Mediante la herramienta appletviewer que viene con el kit para desarrolladores de Java.

En este trabajo se implementa el programa como aplicación por los problemas de seguridad mencionados arriba, y como se ve más adelante, porque existen además ciertas ventajas en el diseño de la interfaz gráfica del usuario.

A.4 Hilos de ejecución múlitple (Multithreading)

La mayoria de los programas tienen un solo hilo de ejecución. Cada programa procede secuencialmente, una instrucción después de otra, hasta que completa su proceso y termina.

Las programas con hilos de ejecución múltiple son similares a los de un solo hilo de ejecución. Difieren en el hecho de que soportan más de un hilo de ejecución concurrente, esto es, están habilitados para ejcecutar simultáneamente múltiples secuencias de instrucciones. Cada secuencia de instrucciones tiene un propio y único flujo de control que es independiente de los otros. Estas secuencias de instrucciones ejecutadas independientemente se conocen como *hilo de ejecución* (thread).

Apéndice B
Abstract Windowing Toolkit (AWT)

Las programas actuales que usan una interfaz gráfica de usuario (GUI Graphic User Interface) y un control de ratón se conocen como *software basado en ventanas*. En Java se pueden crear programas basados en ventanas con el uso de un grupo de clases llamado *Abstract Windowing Toolkit (AWT)*. Debido a que Java es un lenguaje de plataformas cruzadas que permite que se escriban programas par muchos sistema operativos diferentes, su software basado en ventanas debe ser flexible. En vez de apegarse únicamente al estilo basado en ventanas del sistema Windows de Microsoft o al Solaris de Sun, debe manejar ambos, junto con los de otras plataformas. El Abstract Windowing Toolkit (AWT) tiene este nombre debido a que es un conjunto de clase que se pretende que funcionen con cualquier plataforma que ejecute programas de Java. Se necesitó este enfoque para que los programadores pudieran ofrecer sus applets en World Wide Web, que es usada por personas con docenas de tipos diferentes de computadoras, sistemas operativos y navegadores de Web.

Con el AWT se puede crear una GUI que incluya todo lo siguiente:

- Botones (Buttons), casillas de verificación (Checkboxes), etiquetas (Labels) y otros componentes simples
- Cuadros de texto (Text Area) y componentes más complejos
- Cuadros de diálogo (Dialog) y otras ventanas
- Menús desplegables

B.1 Herramientas para organisar una interfaz de usario

B.1.1 Administrador de diseño (Layout Manager)

Hay varios administradores de diseño que se pueden usar par modificar la manera en que se muestran los componentes.

B.1.2 Responder a eventos de usuario

La interfaz gráfica de usuario que proporciona un programa tiene que hacer que pasen cosas cuando sucede un clic del ratón o una entrada de teclado. Las áreas de texto y otros componentes deben ser actualizados para mostrar lo que está sucediendo mientras se ejecuta el programa. Esos cosas son posibles cuando el programa Java puede responder a eventos de usuario. Un *evento* (Event) es un suceso, ya sea provocado por el sistema o por el usuario, que entiende el objeto y que es responsabilidad del programador especificar cómo se responderá. La respuesta a eventos de usuario se llama *manejo de eventos* (event handling). La siguiente sección revisa este tema con más cuidado.

B.2 Ventanas, Marcos (Frames), Diálogos

Además de las gráficas, eventos, componentes de la interfaz del usuario y otros mecanismos para la representación gráfica, AWT ofrece posibilidades para construir elementos de la interfaz del usuario fuera de un Applet o un browser: ventanas, menús, diálogos. Con eso se pueden construir aplicaciones como parte de un Applet o una aplicación independiente de Java.

B.2.1 Clases de ventanas en AWT

Las clases en AWT para construir ventanas y diálogos heredan de una sola clase Window. Esta clase que hereda de Container que es un Component, provee el comportamiento general para todos los elementos parecidos a ventanas.

La clase marcos (Frame) ofrece una ventana con una barra de títulos, un área para cerrar (en general un botón) y otros propiedades de una ventana dependiendo de la plataforma. En estos marcos también se pueden insertar barras para menú. En cambio, el diálogo (Dialog) es una forma restringida de Frame, que normalmente no tiene título. FileDialog, una subclase de Dialog, provee un área estandar para eligir un archivo.

B.2.2 Marcos (Frames)

Un Frame es una ventana que depende de su platafórma, con un título, una barra de menús y botones para minimizar, maximizar y cerrar, junto con otras funciones de una ventana.

Se construye un Frame con uno de los siguientes constructores:

- new Frame() construye un marco básico sin título.
- new Frame(String) construye un marco básico con título.
- Frame se deriva de Window y éste de Container así que se construye y inserta Frame en general como cualquier otro componente de AWT.

B.2.3 Diálogos (Dialog)

Un diálogo se parece a Frame en el sentido que aparece una nueva ventana en la pantalla. Pero es una ventana de transición que sirve para informar el usuario de ciertos eventos o para exigir del usuario que inserte alguna información. En general no tiene títulos (pero es posible que los tenga como sucede en este trabajo), ni se puede cambiar el tamaño de la ventana. Un diálogo puede ser modal, i.e. no se puede insertar nada en otra ventana hasta que se cierra la de diálogo.

El AWT proporciona dos tipos de diálogos: la clase Dialog que contiene un diálogo general y FileDialog que es un browser de archivos.

Objetos del diálogo en general (Dialog)

Para la construcción de un diálogo se usa cualquiera de los siguientes constructores:

- Dialog(Frame, boolean) genera un diálogo invisible que está incluido en el marco actual, modal (true) o no modal (false).
- Dialog(Frame, String, boolean) genera un diálogo invisible con un título (String), modal (true) o no modal (false).

Dialog se puede hacer visible y invisible como Frame con el metodo setVisible(boolean).

A diferencia de un marco, no se puede generar un diálogo directamente en un canvas, se tiene que generar un marco en dónde colocar el diálogo.

En el código para la creación de un diálogo que se usó en un canvas se genera un Frame anchorpoint, en este se coloca un diálogo noSelectedDialog que es una instan-

cia de MessageDialog, la clase de un diálogo en el cual se puede insertar el texto tanto del diálogo como de un botón que esta colocado en el mismo diálogo.

```
MessageDialog noSelectedDialog = new MessageDialog((Frame)anchorpoint,
 "NoSelectedElements", true, "There are no elements selected.\n" + "Nothing to delete.", "OK");
noSelectedDialog.setLocation(75, 75);
noSelectedDialog.setVisible(true);
```

noSelectedDialog tiene el título *"NoSelectedElements"*, es modal y el botón tiene escrito *"OK"* que ocasiona que se despliege el texto:

"There are no elements selected.

Nothing to delete."

La clase MessageDialog, extiende la clase Dialog e implementa la clase ActionListener que cierra la ventana cuando se aprieta el botón.

```
import java.awt.*;
import java.awt.event.*;
public class MessageDialog extends Dialog implements ActionListener
{
protected MultiLineLabel label; /* texto desplegado en el diálgo */
public MessageDialog(Frame parent,String title,boolean modal,String text,
String button)
{
super(parent,title,modal);
/* Crea y usa un BorderLayout manager con espacio 15 pixeles */
this.setLayout(new BorderLayout(15, 15));
/* Crea el componente del mensaje y lo anexa a la ventana */
label = new MultiLineLabel(text, 20, 20);
this.add("Center", label);
/* Inserta el botón en el Panel; anexa el Panel a la ventana */
Panel buttonPanel = new Panel();
Button b = new Button(button);
b.addActionListener(this);
buttonPanel.add(b);
```

```
add("South",buttonPanel);
pack();
}
/* Cierra la ventana después de usar el botón */
public void actionPerformed(ActionEvent e)
{
setVisible(false);
dispose();
}
}
```

B.2.4 Objetos de un diálogo de archivos (FileDialog)

La clase FileDialog provee un diálogo estandar para abrir/salvar archivos. Este se puede aplicar a cualquier sistema de archivos local. La clase FileDialog es independiente del sistema según de la plataforma se abre el diálgo normal para abrir y salvar archivos.

Con los siguientes constructores se puede construir un diálogo para archivos:

- FileDialog(Frame, String) generar und diálogo para archivos que está subordenado el Frame dado con el titulo dado. Se crea un diálogo para abrir un archivo.
- FileDialog(Frame, String, int) genera tambien un diálogo para archivos, con el entero se decide si es un diálogo para abrir o para salvar un archivo. Los optiones para el argumento del modo son FileDialog.LOAD y FileDialog.SAVE.

Se instancia un FileDialog y se le da el modo de abrir archivos.

```
openFileDialog1 = new java.awt.FileDialog(this);
openFileDialog1.setMode(FileDialog.LOAD);
openFileDialog1.setTitle("Open");
```

B.3 El modelo de Eventos del API AWT de Java

Modelo de delegacion

En el modelo de eventos, los eventos proveen un camino para que un componente notifique a otros componentes que occurió algo interesante. Además, asume que la fuentes o "sources" generan los eventos. Para que uno o más receptores o "listeners" sean notificados de la ocurrencia de los eventos generados por un objeto fuente en particular, se deben registrar ante ese objeto fuente. Cuando un objeto fuente detecta que algo interesante ocurrió llamará al método adecuado en el receptor. En ocasiones este modelo es llamado " Modelo de delegación", porque permite al programador delegar la autoridad del manejo de eventos a cualquier objeto que implemente la interfaz listener apropiada (JavaSoft, http://java.sun.com/products/jdk1.1/docs/guide/awt/designspec/events.html)

Conceptualmente, los eventos son mecanismos para propagar las notificaciones de un cambio de estado, entre un objeto fuente y uno o más objetos receptores.

Los detalles del modelo de eventos se describen a continuación:

- Las notificaciones de la ocurrencia de eventos se propagan de los objetos fuente a los receptores mediante la invocación de algunos métodos de los objetos receptor o destino.

- Cada clase de notificación diferente, se define como un método distinto. Estos métodos se agrupan en las interfaces EventListener que se heredan de la clase java.util.EventListener.

- Los objetos receptor se identifican a sí mismos como interesados en un conjunto particular de eventos, al implementar algún conjunto de interfaces EventListener.
- El estado asociado con una notificación normalmente es encapsulado en un objeto *estado del evento* que se hereda de la clase java.util.EventObject y el cual se pasa como argumento al método del evento correspondiente.
- Los objetos fuente se identifican así mismos como una fuente particular de eventos, definiendo métodos de registro y aceptando referencias a instancias de interfaces particulares de EventListener. Los objetos fuente, mantienen una lista de objetos receptor y los tipos de eventos a los que están suscritos. El programador crea esa lista utilizando llamadas a los métodos add<TipoEvento>Listener().
- Una vez que la lista de receptores ha sido creada, el objeto fuente la utiliza para notificar a cada objeto receptor que ha sucedido un evento del tipo que controla: Esto es lo que se conoce como registro de objetos *receptores* especifícos para recibir la notificación de eventos determinados.

Objetos estado del evento

La información asociada con la notificación de un evento es encapsulada normalmente en un objeto *" estado del evento"* que es una subclase de java.util.EventObject. Por convención estas clases del estado de un evento tienen nombres que terminan en "*Event*" Por ejemplo:

MouseEvent que indica un evento ocasionado por un clic del ratón.

Típicamente, los objetos del estado de un evento se deben considerar inmutables. En consecuencia se recomienda, primero no otorgar el acceso público a los campos, y segundo, usar métodos de ac eso para exponer los detalles de los objetos. Sin embargo, cuando alguna de las propiedades de un objeto "estado de evento" requiera de modificación (por ejemplo el traslado de las coordenadas relativas de una ventana), se sugiere que tales modificaciones se encapsulen dentro de un método apropiado, quien efectuará la operación requerida, como se observa en el código anterior.

El AWT define un conjunto determinado de eventos, aunque el programador también puede definir sus propios tipos de eventos, derivando clases de EventObject, o desde una de las clases de eventos del AWT.

B.3.1 Interfaces EventListener

El nuevo modelo de eventos de Java se basa en la invocación de métodos. Esto implica que se debe contar con una forma estándar de definir y agrupar los métodos para el manejo de eventos. Además, estos métodos serán definidos dentro de las interfaces EventListener, que son heredadas de java.util.EventListener .Por convención, las interfaces EventListener se designan con nombres terminados en *"Listener"*.

Aquella clase que desee manejar un conjunto de eventos definidos en la interfaz EventListener, deberá implementar los métodos de esa interfaz. Por ejemplo

```
public class MouseMovedExampleEvent extends java.util.EventObject.....

interfaz MouseMovedExampleListener extend java.util.EventListener....

class ArbitraryObject implements MouseMoveExampleListener...
```

Típicamente, los métodos relacionados con el manejo de eventos, se agrupan en la misma interfaz EventListener. Por ejemplo, mouseEntered; mouseMoved, y mouse-Exit podrán agruprase en la misma interfaz EventListener.

B.3.2 Registro del receptor (Listener) de eventos

Para que los receptores potenciales EventListener, se registren a sí mismos con las instancias apropiadas de *fuentes* de eventos o *sources*, de manera que se establezca el flujo de eventos de la fuente al receptor, las clases *fuente de eventos* deben proveer de métodos para registrar y borrar del registro a los objetos *receptores de eventos*.

Apéndice C
Instalación del Programa Stringalgebras

Obtención del Software

El software y su documentacion está disponible en el sitio de web:

http://www.matem.unam.mx/~sdieck

Se instala en cualquier plataforma que soporta la máquina Virtual de Java con el JDK 1.2. Como Java es portatil el sistema operativo puede ser cualquiera, o sea Windows, Solaris, Linux, etc.

Apéndice D
Glosario

API	Application Programming Interface. Interfaz de la programación de aplicaciones
applet	Icono animado que posee su propio programa
AWT	Abstract Window Toolkit. Kit de herramienta de ventanas abstractas
browser	Navegador
byte code	Código de bytes
class loaders	Cargadores de clases
COM	Component Object Model. Componente del modelo de objetos
event handling	Manejo de eventos
GUI	Graphic User Interface. Interfaz gráfica de usuario
HTML	Hyper Text Markup Language. Lenguaje de marcado de hipertexto
IDE	Integrated Developer Environment. Ambiente de desarollo integrado
inheritance	Herencia
JDK	Java Developer Kit
Layout Manager	Administrador de diseño
Listener	Receptor
Multithreading	Hilos de ejecución múltiple
OMG	Object Management Group. Grupo de administración de objetos
OOP	Object Oriented Programming. Programación orientada a objetos
overriding	Sobrrposición
SDK	Software Development Kit. Kit de desarollo de software
source file	Archivo fuente (aquí en Java)
stream	Flujo
thread	Hilo de ejecución, se refiere a una trayectoria de ejecución en un programa
UML	Unified Modelling Laguage. Lenguaje unificado de modelado
URL	Uniform Resource Locator. Localizador uniforme de recursos

Bibliografía

H. Krause, *Maps between tree and band modules,* J. Algebra 137 (1991), 186-194.

M.C.R. Butler, C.M. Ringel, *Auslander-Reiten sequences with few middle terms and applications to string algebras,* Comm. in Alg. 15 (1&2) (1987), 145-179.

W.W. Crawley-Boevey, *Maps between representations of zero-relation algebras,* J. Algebra, 126, (1989), 259–263.

P.W. Donovan, M.R. Freislich; *The Indecommposable Modular Representations of Certain Groups with Dihedral Sylow Subgroup*, Math. Ann., 238, (1978), 207-216.

Ch. Geiss, *On components of type* $\mathbf{Z}\, A_\infty^\infty$ *for string algebra,* Comm. Algebra, 26, (1998), 749–758.

Ch. Geiss, *Maps between representations of clans,* J. Algebra, 218, (1999), 131–164.

C Riedtmann, *Algebren, Darstellungskoecher, Ueberlagerungen und zurueck,* Comment. Math. Helvetici, 55, (1980), 199-224.

C.M. Ringel, *The indecomposable representations of the dihedral 2-groups,* Math. Ann. 214 (1975), 19-34.

C.M. Ringel, *The repetitive algebra of a gentle algebra,* Bol. Soc. Mat. Mexicana (3) **2** (1997), 235–253.

C.M. Ringel, *On generic modules for string algebras,* to appear in Bol. Soc. Mat. Mexicana.

J. Schroeer, *On the Krull-Gabriel dimension of an algebra,* 1998, (Preprint) to appear in Math.Z.

J. Schroeer, *Modules without self-extensions over gentle algebras,* J. Algebra, 216, (1999), 178–189.

A. Skowronski, J. Waschbuesch, *Representation-finite biserial algebras*, J. f. Mathematik, 345, (1983), 172-181.

F.W. Anderson, K.R. Fuller, *Rings and Categories of Modules,* 2nd edition, Springer, 1992

M. Auslander, I. Reiten, S. O. Smalø, *Representation Theory of Artin Algebras,* Cambridge University Press, 1995.

C. Ciblis, F. Larrión, L. Salmerón, *Métodos Diagramáticos en Teoría de Representaciones,* Monografías del Instituto de Matemáticas 11, 1981.

F. Larrión, G. Raggi, L Salmerón, *Rudimentos de Mansedumbre y Salvajismo en Teoría de Representaciones,* Aportaciones Matemáticas, Textos, 5, Soc. Mat. Mexicana, 1995.

C.M. Ringel, *Tame Algebra and Integral Quadratic Forms*, Springer, Lecture Notes in Mathematics, 1099, 1984.

J.J. Rotman, *An introduction to homological algebra*, Academic Press Inc., New York, 1979.

R.Steyer, *Java 1.2 Schnelluebersicht,* Markt&Technik, 1998.

J. Jaworski, *Java 2 Platform Unleashed,* SAMS, 1999

D. Flanagan, *Java in a Nutshell,* O'Reilly & Associates, 1997

J. Hunt, A. McManus, *Key Java,* Springer, 1998

L. Lemay, R. Cadenhead, *Java 2 in 21 Days,* SAMS, 1999

J. Zukowsky, *Java AWT Reference,* O'Reilly & Associates, 1997

G. Krueger, *Go To Java 2,* Addison-Wesley, 1999.

G. Booch, J. Rumbaugh, J. Jacobson, *UML Benutzerhandbuch (UML User Guide),* Addison-Wesley, 1999.

C. Larman ,*Applying UML and Patterns*, Prentice Hall, 1998.

J. Martin, J.J. Odell, *Objektorientierte Modellierung mit UML: Das Fundament*, Prentice Hall, 1999.

Bernd Oesterreich, *Objektorientierte Softwareentwicklung mit der UML,* 3.Aufl. Oldenbourg , 1997.

http://www.gamelan.com Un sito útil para código en Java y preguntas acerca de Java.

http://java.sun.com El sitio oficial de Sun, alli se consigue el JDK.

http://www.omt.com El sitio del Object Modeling Group que maneja cuestiones de UML.

http://www.OTW.de El sitio dónde se encontró la herramienta para elaborar las diagramas de sequencia.

Indice