Apéndice

En esta parte de la tesis damos una breve descripción del paquete YangBaxter del sistema GAP y de las funciones que utilizamos a lo largo del trabajo, y proveemos los códigos de las funciones que definimos en GAP y una breve descripción de cada una.

Por medio del paquete YangBaxter, desarrollado por Vendramin y Konovalov, GAP representa a las soluciones conjuntistas de la ecuación de Yang-Baxter como el tipo de dato YBType que consiste de 2 matrices, <l_actions> y <r_actions>, que representan las acciones a izquierda y a derecha, respectivamente, escritas como la imagen de las permutaciones inducidas. Veamos un ejemplo. Consideremos la solución SmallIYB(4,4) dada por $X = \{1, 2, 3, 4\}$ y r de tabla.

x	1 2	3 4
$\mathcal{L}_x$	e	(34)
$\mathcal{R}_x$	e	(34)

GAP codifica a esta solución como el objeto

$$YB([[1,2,3,4],[1,2,3,4],[1,2,4,3],[1,2,4,3]],[[1,2,3,4],[1,2,3,4],[1,2,4,3],[1,2,4,3]]).$$

A lo largo del trabajo los ejemplos los obtuvimos por medio de las funciones SmallIYB(n,m) y SmallSkewbrace(n,m). Las mismas devuelven la solución m-ésima de cardinal n de la base de datos del paquete YangBaxter. Cabe mencionar que la función SmallSkewbrace(n,m) no devuelve una solución per sé sino la braza torcida m-ésima de cardinal n de la base de datos. Por lo cual, debemos utilizar la función Skewbrace2YB() para obtener las solución SS(n,m).

En el Capítulo 4 utilizamos la función GeneratorsOfAlgebra($\mathfrak{g}$) que devuelve los generadores del álgebra de Lie $\mathfrak{g}$ escritos en notación matricial respecto de la base canónica $\{v_1, \dots, v_n\}$ de V_X.

Veamos las funciones que definimos, las ordenamos por capítulo en que fueron utilizadas y agregamos una sección con adaptaciones y modificaciones de algunas funciones existentes en el paquete YangBaxter.

Capítulo 1

- La siguiente función toma como argumento la solución `obj` y devuelve el conjunto $\overline{X}$ formado por las clases de equivalencia en forma de la lista `c`.

```
TildeX := function(obj)
local pairs, e, c, x, y;
pairs := [];
for x in [1..Size(obj)] do
for y in [1..Size(obj)] do
if (LPerms(obj)[x] = LPerms(obj)[y]) and
(RPerms(obj)[x] = RPerms(obj)[y]) then
Add(pairs, [x, y]);
fi;
od;
od;

e := EquivalenceRelationByPairs(Domain([1..Size(obj)]),
   pairs);
c := EquivalenceClasses(e);


return c;
end;
```

- Definimos la función ProjectorOfYB() que toma como input la solución `obj` y devuelve la lista `projectors` con las matrices de las proyecciones a los subespacios generados por las clases de equivalencia (i.e. $p_{\overline{x}}$). YBProjector() es una función cuyo input es el natural `n` y una sublista `subSpaceList` de la lista $\{1, 2, \ldots, n\}$ y cuyo output es la matriz `m` de $n \times n$ diagonal con 1's en las entradas (i, i) para cada $i \in$ `subSpaceList`.

```
YBProjector := function(subSpaceList, n)
local m,i;
m := NullMat(n,n);
for i in subSpaceList do
m[i,i]:=1;
od;
return m;
end;

ProjectorsOfYB := function(obj)
local pairs, e, c, projectors, x, y, j;
projectors := [];
pairs := [];
for x in [1..Size(obj)] do
for y in [1..Size(obj)] do
if (LPerms(obj)[x] = LPerms(obj)[y]) and
(RPerms(obj)[x] = RPerms(obj)[y]) then
Add(pairs, [x, y]);
```

```
fi;
od;
od;

e := EquivalenceRelationByPairs(Domain([1..Size(obj)]),
    pairs);
c := EquivalenceClasses(e);

for j in [1..Size(c)] do
Add(projectors, YBProjector(Elements(c[j]),Size(obj)));
od;

return projectors;
end;
```

- Definimos las funciones LeftMatsOfYB() y RightMatsOfYB() cuyas entradas son la solución `obj` y cuyas salidas son las matrices dadas por $LPerms(\texttt{obj}) =< l\_actions >$ y $RPerms(\texttt{obj}) =< r\_actions >$, respectivamente.

```
LeftMatsOfYB := function(obj)
return List(LPerms(obj), x-> PermutationMat(x,Size(obj)))
    ;
end;

RightMatsOfYB := function(obj)
return List(RPerms(obj), x-> PermutationMat(x,Size(obj)))
    ;
end;
```

- Definiremos la función IsOpConmut() cuya entrada es la solución `obj` y que devuelve el booleano `bool_final` que dependerá de si los operadores de la solución conmutan entre sí.

  Primero definimos la función IsMatConmut() que tiene como entrada la lista de matrices `lista` y devuelve el booleano `bool_final` cuyo valor dependerá de si las matrices de la lista conmutan entre sí.

  Precondición: `lista` es una lista de matrices cuadradas.

```
IsMatConmut:=function(lista)
local bool_final, i, j;
bool_final:=true;
for i in [1..Size(lista)] do
for j in [1..Size(lista)] do
if(lista[i]*lista[j]=lista[j]*lista[i]) then
bool_final:=bool_final and true;
else
```

```
bool_final:=bool_final and false;
fi;
od;
od;
return bool_final;
end;
```

Ahora definimos la función IsOpConmut(). Necesita las funciones LeftMatsOfYB() y RightMatsOfYB() del Capítulo 3.

```
IsOpConmut:=function(obj)
local LPerm_mat, RPerm_mat, Mats, bool_final;
LPerm_mat:=LeftMatsOfYB(obj);
RPerm_mat:=RightMatsOfYB(obj);
Mats:=[];
Append(Mats,LPerm_mat);
Append(Mats,RPerm_mat);
bool_final:=IsMatConmut(Mats);
return bool_final;
end;
```

- La función que sigue tiene como input la solución `obj` y devuelve el booleano `bool_final` según si la solución tenga retracción trivial o no. Como usamos RetractNotInv() la función es independiente de que la solución no sea involutiva. Necesita la función RetractNotInv() de la sección Adaptaciones de funciones del paquete YangBaxter.

```
IsRetTrivial := function(obj)
local LPerm_ret,RPerm_ret, bool_final, i, j;
bool_final:=true;
LPerm_ret:=LPerms(RetractNotInv(obj));
RPerm_ret:=RPerms(RetractNotInv(obj));
for i in [1..Size(LPerm_ret)] do
if (LPerm_ret[i]=()) then
bool_final:=bool_final and true;
else
bool_final:=bool_final and false;
fi;
od;
for j in [1..Size(RPerm_ret)] do
if (RPerm_ret[j]=()) then
bool_final:=bool_final and true;
else
bool_final:=bool_final and false;
fi;
od;
```

```
return bool_final;
end;
```

- Definiremos la función Invariantes() cuya entrada es la solución `obj` y cuya salida son los subconjuntos r-invariantes de la solución en forma de la lista `lista_final`. Debemos definir antes las funciones Ordenar_lista_de_listas() y DFS().

  En primer lugar definimos la función Ordenar_lista_de_listas() cuya entrada es la lista de listas `lista_input` y el natural `largo_max` y cuya salida es la misma lista pero ordenada por cardinalidad de cada uno de sus elementos. Si sabemos el largo máximo `largo_max` de las listas de `lista_input` esta técnica funciona.

```
Ordenar_lista_de_listas := function(lista_input,
    largo_max)
local lista_cardinalidades, i;
lista_cardinalidades:=[];
for i in [1..largo_max] do
Add(lista_cardinalidades,[]);
od;
for i in lista_input do
Add(lista_cardinalidades[Size(i)],i);
od;
return lista_cardinalidades;
end;
```

  Definimos ahora la función DFS = Depth First Search (o búsqueda en profundidad) que tiene como input la lista `lista_input` y cuyo output es la lista `lista_output` de todas las sublistas de la `lista_input`. Lo hacemos por backtracking. Tenemos que definir primero el paso iterativo porque en GAP debemos programar en orden.

```
DFS_iterativo := function(lista_input, lista_aux)
local lista_output, lista_aux_1, lista_aux_2, i;
lista_output:=[];
lista_aux_1:=[];
lista_aux_2:=[];
i:=1;
while i < Size(lista_input)+1 do
lista_aux_1:=ShallowCopy(lista_input);
lista_aux_2:=ShallowCopy(lista_aux);
lista_aux:=Concatenation(lista_aux,[Remove(lista_input,i)
    ]);
Add(lista_output,lista_aux);
lista_output:=Concatenation(lista_output,
DFS_iterativo(lista_input, lista_aux));
```

```
i:=i+1;
lista_aux:=ShallowCopy(lista_aux_2);
lista_input:=ShallowCopy(lista_aux_1);
od;
for i in lista_output do
Sort(i);
od;
return Unique(lista_output);
end;
```

Ahora definimos la función general DFS.

```
DFS := function(lista_input)
return DFS_iterativo(lista_input, []);
end;
```

Definimos Invariantes().

```
Invariantes := function(obj)
local lista_DFS, lista_output, lista_aux, lista_final, i,
    j;
lista_output:=[];
lista_final:=[];
lista_DFS:=DFS([1..Size(obj)]);
for i in lista_DFS do
if IsInvariant(obj, i) then
Add(lista_output,i);
fi;
od;

lista_aux:=Ordenar_lista_de_listas(lista_output, Size(obj
   ));
for i in lista_aux do
for j in i do
Add(lista_final,j);
od;
od;

return lista_final;
end;
```

- Definimos la función NoTrivialClass() que tiene como entrada la solución `obj` y cuya salida son las clases de equivalencias C para las que $r|_{CxC}$ no es una solución trivial, en forma de lista.

  Primero tenemos que definir la función auxiliar ListEquivalenceRelYB() que tiene como input la solución `obj` y como output la lista `lista_output` formada por las listas de los elementos relacionados entre sí, i.e. la lista de las clases de equivalencia.

```
ListEquivalenceRelYB := function(obj)
local pairs, e, c, lista_output, x, y;
pairs := [];
for x in [1..Size(obj)] do
for y in [1..Size(obj)] do
if (LPerms(obj)[x] = LPerms(obj)[y]) and
(RPerms(obj)[x] = RPerms(obj)[y]) then
Add(pairs, [x, y]);
fi;
od;
od;

e := EquivalenceRelationByPairs(Domain([1..Size(obj)]),
    pairs);

lista_output:=Unique(Successors(e));
return lista_output;
end;
```

Ahora NoTrivialClass().

```
NoTrivialClass := function(obj)
local lista, lista_2, i;
lista:=ListEquivalenceRelYB(obj);
lista_2:=[];
for i in lista do
if IsInvariant(obj,i) then
Add(lista_2,i);
fi;
od;
return Filtered(lista_2,i->IsTrivialYB(RestrictedYB(obj,i
    ))=false);
end;
```

- La siguiente función es una adaptación de la función anterior, de forma que tenga como input el natural `n` y devuelva la lista `lista_final` donde cada elemento es NoTrivialClass(SmallIYB(n,j)) para todo $1 \leq j \leq$ NrSmallIYB(n). Es decir, esta función ahorra el trabajo de probar una por una las soluciones cargadas en SmallIYB(`n`). Necesita la función NoTrivialClass() anterior.

```
NoTrivialClass_for_all := function(n)
local lista_final, i;
lista_final:=[];
```

```
i:=1;
while i<NrSmallIYB(n)+1 do
Add(lista_final, [NoTrivialClass(SmallIYB(n,i)),i]);
i:=i+1;
od;
return Unique(lista_final);
end;
```

Capítulo 2

- La siguiente función tiene como entrada la solución `obj` y devuelve el álgebra $A(X,r)$ como la salida `A/I`.

```
AlgX:=function(obj)
local A, lista_gen, lista_rel, LL, RR, I, i, j;
A:=FreeAssociativeAlgebraWithOne(Rationals, Size(obj), "x
   ");
lista_gen:=[];
lista_rel:=[];
LL:=LPerms(obj);
RR:=RPerms(obj);
for i in [1..Size(obj)] do
lista_gen[i]:=GeneratorsOfAlgebra(A)[i];
od;
for i in [1..Size(obj)] do
for j in [1..Size(obj)] do
Add(lista_rel,lista_gen[i]*lista_gen[j]-lista_gen[j^LL[i
   ]]*lista_gen[i^RR[j]]);
od;
od;
I:=Ideal(A,lista_rel);
return A/I;
end;
```

Capítulo 3

- La siguiente función toma la solución `obj` y devuelve el álgebra de Lie $\mathfrak{g}(X,r)$.

  Debemos definir primero la función auxiliar TensorMatricesOfYB() que tiene como input la solución `obj` y como output la lista `mats` que representa el conjunto de todas las matrices de la forma $L{\cdot}p$ y $R{\cdot}p$ con L en LeftMatsOfYB, R en RightMatsOfYB y p en ProjectorsOfYB del Capítulo 1.

```
TensorMatricesOfYB := function(obj)
local mats, pp, ll, rr, i, j;
```

```
mats := [];
ll := LeftMatsOfYB(obj);
rr := RightMatsOfYB(obj);
pp := ProjectorsOfYB(obj);
for i in [1..Size(ll)] do
for j in [1..Size(pp)] do
Append(mats,[ll[i]*pp[j], rr[i]*pp[j]]);
od;
od;
return Unique(mats);
end;
```

Ahora LieAlgebraOfYB().

```
LieAlgebraOfYB := function(obj)
return LieAlgebra(Rationals, TensorMatricesOfYB(obj));
end;
```

- La función que sigue LieSubalgebraOfYB() tiene como argumentos la solución `obj` y el subconjunto r-invariante `lista` en forma de lista y tiene como salida el álgebra $\mathfrak{g}(Y, r_Y)$ asociada a una subsolución Y pero pensada como subálgebra de $\mathfrak{g}(X, r)$.

  Pero antes necesitamos definir una adaptación de la función ProyectorsOfYB() para que tome como input la solución `obj` y el subconjunto r-invariante `lista` de la solución en forma de lista, y devuelva la lista `projectors` formada por los proyectores de la subsolución inducida por `lista`, pensados como operadores en el espacio total V_X. Necesita la función YBProjector() del Capítulo 1.

  Pre: `lista` es un subconjunto r-invariante del conjunto subyacente X de la solución `obj`.

```
ProyectorsOfYBSubsol := function(obj,lista)
local LPerms_Y, RPerms_Y, pairs, e, c, projectors, x, y,
   j;
LPerms_Y:=LPerms(obj){lista};
RPerms_Y:=RPerms(obj){lista};
projectors := [];
pairs := [];
for x in [1..Size(lista)] do
for y in [1..Size(lista)] do
if (LPerms_Y[x] = LPerms_Y[y]) and
(RPerms_Y[x] = RPerms_Y[y]) then
Add(pairs, [x, y]);
fi;
od;
od;
```

```

e := EquivalenceRelationByPairs(Domain([1..Size(lista)]), pairs);
c := EquivalenceClasses(e);

for j in [1..Size(c)] do
Add(projectors, YBProjector(Elements(c[j]),Size(obj)));
od;

return projectors;
end;
```

Ahora sí, LieSubalgebraOfYB().

```
LieSubalgebraOfYB := function(obj, lista)
local gen_Y,LPerms_Y,RPerms_Y,LPerms_Mat_Y,RPerms_Mat_Y,
Proyectors_Y, lista_aux, i, j, k;
LPerms_Y:=LPerms(obj){lista};
RPerms_Y:=RPerms(obj){lista};

#Defino las matrices de |X|x|X| asociadas a los op de r_Y.

LPerms_Mat_Y:=List(LPerms_Y,x->PermutationMat(x,Size(obj)));
RPerms_Mat_Y:=List(RPerms_Y,x->PermutationMat(x,Size(obj)));
Proyectors_Y:=ProyectorsOfYBSubsol(obj,lista);
lista_aux:=[];
for i in LPerms_Mat_Y do
for j in RPerms_Mat_Y do
for k in Proyectors_Y do
Add(lista_aux,i*k);
Add(lista_aux,j*k);
od;
od;
od;
gen_Y:=Unique(lista_aux);
return LieAlgebra(Rationals, gen_Y);
end;
```

- La siguiente función, IdealsOfLieYB(), toma la solución `obj` y devuelve la lista `lista_final` formada por las subsoluciones para las que $\mathfrak{g}(Y, r_Y)$ no sólo es subálgebra de $\mathfrak{g}(X, r)$ sino que además es un ideal.

  Pero primero necesitamos definir la función IsIdealOfLieYB() que tiene como argumentos la solución `obj` y el subconjunto r-invariante Y en forma de la lista `lista`, y como salida el booleano `bool_final` cuyo valor dependerá de si $\mathfrak{g}(Y, r_Y)$ es ideal de $\mathfrak{g}(X, r)$. Necesita las funciones LieAlgebraOfYB(), ProyectorsOfYBSubsol() y LieSubalgebraOfYB() de este Capítulo.

```
IsIdealOfLieYB := function(obj, lista)
local g_X, g_Y, gen_X, gen_Y, bool_final, i, j, k;
g_X:=LieAlgebraOfYB(obj);
gen_X:=GeneratorsOfAlgebra(g_X);
g_Y:=LieSubalgebraOfYB(obj, lista);
gen_Y:=GeneratorsOfAlgebra(g_Y);
bool_final:=true;
for i in gen_X do
for j in gen_Y do
if i*j in g_Y then
bool_final:=bool_final and true;
else bool_final:=false;
fi;
od;
od;
return bool_final;
end;
```

Ahora sí, IdealsOfLieYB(). Necesita la función Invariantes() del Capítulo 1.

```
IdealsOfLieYB := function(obj)
local i, lista_final, lista_aux;
lista_final:=[];
lista_aux:=Invariantes(obj);
for i in lista_aux do
if IsIdealOfLieYB(obj, i) then
Add(lista_final, i);
fi;
od;
return lista_final;
end;
```

Adaptaciones de funciones del paquete YangBaxter

- Las siguientes son adaptaciones de las funciones Retract(), Multipermutation-Level() e IsMultipermutation() para soluciones no necesariamente involutivas. Las mismas tienen como input la solución `obj` y como output la retracción de `obj`, el nivel de multipermutación de `obj` y un booleano que dependerá de si la solución es de multipermutación (es decir, si existe $n \in \mathbb{N}$ tal que $Ret^n(X, r)$ tiene cardinal 1), respectivamente.

```
RetractNotInv := function(obj)
local e, c, s, pairs, x, y, z, ll, rr;

pairs := [];
for x in [1..Size(obj)] do
for y in [1..Size(obj)] do
if (LPerms(obj)[x] = LPerms(obj)[y]) and
(RPerms(obj)[x] = RPerms(obj)[y]) then
Add(pairs, [x, y]);
fi;
od;
od;

e := EquivalenceRelationByPairs(Domain([1..Size(obj)]),
   pairs);
c := EquivalenceClasses(e);
s := Size(c);

ll := List([1..s], x->[1..s]);
rr := List([1..s], x->[1..s]);

for x in [1..s] do
for y in [1..s] do
z := YB_xy(obj, Representative(c[x]), Representative(c[y
   ]));
ll[x][y] := Position(c, First(c, u->z[1] in u));
rr[y][x] := Position(c, First(c, u->z[2] in u));
od;
od;
return YB(ll, rr);
end;


MultipermutationLevelNotInv := function(obj)
local r,s,l;

l := 0;
r := ShallowCopy(obj);

repeat
s := ShallowCopy(r);
r := RetractNotInv(s);
if Size(r) <> Size(s) then
l := l+1;
else
return fail;
fi;
```

```
until Size(r) = 1;
return l;
end;


IsMultipermutationNotInv := function(obj)
if not MultipermutationLevelNotInv(obj) = fail then
return true;
else
return false;
fi;
end;
```

- Ahora definiremos adaptaciones de las funciones YB_ij(), IS_YB(), YB(), DisplayTable() e IsInvolutive() para que tomen como entradas la solución `obj` y dos listas de permutaciones `l` y `r` que representan las acciones a izquierda y a derecha de la solución, respectivamente, con: YB_ij() tiene como entrada las matrices <l_actions> y <r_actions>, un vector v de largo $|X|$ con X el conjunto subyacente a la solución obj inducida por <l_actions> y <r_actions>, y dos coordenadas i, j, y devuelve el valor de la solución obj actuando en las coordenadas i, j de v; IS_YB() tiene como argumento las matrices <l_actions> y <r_actions> y como salida un booleano que depende de si esas matrices inducen una solución; YB() tiene como entradas las matrices <l_actions> y <r_actions> y como salida la solución inducida por esas matrices; DisplayTable() tiene como input una solución obj y como salida la tabla formada por la imagen del operador r; e IsInvolutive() tiene como argumento una solución obj y como salida un booleano cuyo valor dependerá de si obj es involutiva.

```
YB_ij_by_perm:=function(l, r, v, i, j)
local w;
w := ShallowCopy(v);
w[i] := v[j]^l[v[i]];
w[j] := v[i]^r[v[j]];
return w;
end;


IS_YB_by_perm:=function(l, r)
local x, y, z, v;

if Size(r) <> Size(l) then
return false;
fi;

for x in [1..Size(l)] do
for y in [1..Size(l)] do
```

```
for z in [1..Size(l)] do
v := [x,y,z];
if YB_ij_by_perm(l, r,
YB_ij_by_perm(l, r,
YB_ij_by_perm(l, r, v, 2, 3), 1, 2), 2, 3)
\<>
YB_ij_by_perm(l, r,
YB_ij_by_perm(l, r,
YB_ij_by_perm(l, r, v, 1, 2), 2, 3), 1, 2) then
return false;
fi;
od;
od;
od;
return true;
end;


YB_by_perm :=function(l, r)
local ll, rr, x, y;
if not IS_YB_by_perm(l, r) then
Error("this is not a solution of the YBE\n");
fi;

ll := List([1..Size(l)], x->[1..Size(l)]);
rr := List([1..Size(l)], x->[1..Size(l)]);
for x in [1..Size(l)] do
for y in [1..Size(l)] do
ll[x][y] := y^l[x];
rr[y][x] := x^r[y];
od;
od;
return YB(ll, rr);
end;


DisplayTable_by_perm := function(l,r)
local m, x, y;
m := NullMat(Size(l), Size(l));
for x in [1..Size(l)] do
for y in [1..Size(l)] do
m[x][y] := [y^l[x],x^r[y]];
od;
od;
return m;
end;
```

```
IsInvolutive_by_perm := function(l,r)
local table,x,y,s;
table:=DisplayTable_by_perm(l,r);
for x in [1..Size(l)] do
for y in [1..Size(l)] do
s :=  table[x][y];
if table[s[1]][s[2]] <> [x,y] then
return false;
fi;
od;
od;
return true;
end;
```

Bibliografía

[1] R. J. Baxter. *Partition function of the eight-vertex lattice model.* Ann. Physics, 70:193–228, 1972.

[2] R. J. Baxter. *Exactly solved models in statistical mechanics.* Academic Press, Inc. [Harcourt Brace Jovanovich, Publishers], London, 1989. Reprint of the 1982 original.

[3] V. G. Drinfel'd. *On some unsolved problems in quantum group theory.* In Quantum groups (Leningrad, 1990), volume 1510 of Lecture Notes in Math., pages 1–8. Springer, Berlin, 1992.

[4] P. Etingof, T. Schedler and A. Soloviev. *Set-theoretical solutions to the quantum Yang-Baxter equation.* Duke Math. J., 100(2):169–209, 1999.

[5] L.D. Faddeev, N.Yu. Reshetikhin and L.A. Takhtajan. *Quantization of Lie groups and Lie algebras* (in Russian), Algebra i Analiz 1: 178–206, 1989; English translation in Leningrad Math. J. 1: 193–225, 1990.

[6] T. Gateva-Ivanova and P. Cameron. *Multipermutation solutions of the Yang-Baxter equation.* 2009.

[7] T. Gateva-Ivanova and S. Majid. *Quantum spaces associated to multipermutation solutions of level two*, Algebr. Represent. Theory, 14(2): 341–376, 2011.

[8] T. Gateva-Ivanova and M. Van den Bergh. *Semigroups of I-type.* J. Algebra, 206(1):97–112, 1998.

[9] K. R. Goodearl and R. B. Warfield Jr. *An Introduction to Noncommutative Noetherian Rings*

[10] L. Guarnieri. *La ecuación conjuntista de Yang-Baxter.* 2016.

[11] L. Guarnieri and L. Vendramin. *Skew braces and the Yang-Baxter equation.* Accepted for publication in Math. Comp. DOI:10.1090/mcom/3161.

[12] J. E. Humphreys. *Introduction to Lie Algebras and Representation Theory.* Springer-Verlag. Third printing, revised, 1980.

[13] N. Jacobson. *Structure of rings*. American Mathematical Society Colloquium Publi- cations, Vol. 37. Revised edition. American Mathematical Society, Providence, R.I., 1964.

[14] E. Jespers and J. Okniński. *Monoids and Groups of I-Type*, Algebr. Represent. Theory 8: 709–729, 2005.

[15] E. Jespers and J. Okniński. *Noetherian Semigroup Algebras*, Springer, Dordrecht 2007.

[16] J.-H. Lu, M. Yan, and Y.-C. Zhu. *On the set-theoretical Yang-Baxter equation.* Duke Math. J., 104(1):1–18, 2000.

[17] W. Rump. *A decomposition theorem for square-free unitary solutions of the quantum Yang-Baxter equation.* Adv. Math., 193(1):40–55, 2005.

[18] W. Rump. *Braces, radical rings, and the quantum Yang-Baxter equation.* J. Algebra, 307(1):153–170, 2007.

[19] A. Solotar, M. Farinati and M. Suárez Alvarez. *Anillos y sus categorías de representaciones*. 2006.

[20] J. Tate and M. Van den Bergh. *Homological properties of Sklyanin algebras.* Invent. Math. 124 (1996), 619–647.

[21] A. Weinstein and P. Xu. *Classical solutions of the quantum Yang-Baxter equation*. Comm. Math. Phys., 148(2):309–343, 1992.

[22] C.N. Yang. *Some exact results for the many-body problem in one dimension with repulsive delta-function interaction.* Phys. Rev. Lett., 19:1312–1315, 1967.