# VERSATILIDAD REPRESENTATIVA Y TRANSICIÓN HACIA LA PERSISTENCIA *PER SE* DE DATOS ESTRUCTURALMENTE COMPLEJOS CON HASKELL

Wildo Gómez, Gustavo Sosa-Cabrera y María Elena García Díaz
*Facultad Politécnica, Universidad Nacional de Asunción, Paraguay*

**RESUMEN**

En los últimos años, se ha desarrollado un creciente interés por las formas de abordar la representación y el almacenamiento de los tipos de datos considerados *estructuralmente complejos*. Sin embargo, la mayoría de los estudios en el campo del descubrimiento de conocimiento sólo se han centrado en lo que respecta a la gestión y almacenamiento de grandes volúmenes de información. En este trabajo se aborda una exploración de la versatilidad en la definición de tipos de datos en *Haskell* y su asequible empleo para la representación de datos conceptualmente *polimorfos*. Se describe los aspectos relacionados al almacenamiento de las definiciones de tipos de datos en *Haskell* y la implementación de una librería conforme a la identificación de los factores mínimos hacia una *persistencia primitiva.*

**PALABRAS CLAVES**

Versatilidad Representativa del Conocimiento, Complejidad Estructural de Datos, Paradigma Declarativo, Programación Funcional, Persistencia de Datos, Haskell

## 1. INTRODUCCIÓN

Con el propósito de construir representaciones de la realidad, se vienen utilizando desde hace muchas décadas los modelos de datos (Simon y Kazmierczak, 2004). Este proceso de poder resumir una visión del mundo, ha incrementado la demanda de distintas formas de abordar los tipos de datos más complejos. Asimismo, el almacenamiento de datos es importante porque casi todos los programas no triviales manipulan datos que son persistentes en el tiempo (Thinder, 1989).

Representar conceptos del mundo real y almacenar los modelos de datos consecuentes es un proceso que puede hacerse por medio de gestores de bases de datos o por la acción de guardar y restaurar explícitamente las estructuras de datos del lenguaje de programación en el sistema de archivos. Este último se conoce como el mecanismo de *persistencia de primera generación* (Dearle et al, 2009).

En el presente, los sistemas de gestión de bases de datos relacionales son el medio de almacenamiento predominante, donde sus aplicaciones consisten en tareas de procesamiento de datos, tales como la banca y la gestión de nóminas. Dichas aplicaciones presentan conceptualmente tipos de *datos simples*, puesto que los elementos de datos básicos son registros bastante pequeños y cuyos campos son atómicos, es decir, no contienen estructuras adicionales y por tanto dan cumplimiento a la primera forma normal (Aballay et al, 2017).

Sin embargo, la investigación en el área de las bases de datos relacionales no ha crecido tan rápido como la complejidad de los datos que tienen que almacenar (López y Gómez, 2007) Bajo este escenario, aplicaciones que trabajan con datos no `tradicionales', como temporales, espaciales, médicos, multimedia, científicos, de ingeniería o geográficos, tienden a utilizar directamente lenguajes de programación de propósito general que les permitan diseñar estructuras a medida de sus necesidades, así como también *Base de datos No Relacionales*. A lo largo de este documento, nos referimos como datos `no tradicionales' a aquellos valores para los atributos en una fila que no son atómicos y/o dependen de parámetros, variables o en todo caso vienen definidos por relaciones entre variables mediante ecuaciones o inecuaciones matemáticas.

Surge, por tanto, la necesidad de estimular el debate acerca de las estructuras de datos de los lenguajes de programación de propósito general, en cuanto a versatilidad se refiere, para la representación de datos `no tradicionales'.

En lo que a este estudio concierne, el principal objetivo de esta investigación es desarrollar un mejor entendimiento acerca de la suficiencia y amplitud de opciones para las definiciones de tipos de datos a medida del lenguaje de programación funcional (Curry, 2010). Adicionalmente, se trata una serie de cuestiones que han surgido a partir de la necesidad de persistencia de las definiciones de tipos de datos y sus valores para su preservación en el tiempo.

La organización del resto del artículo es la siguiente. *La Sección 2* presenta brevemente el paradigma de programación subyacente y la definición de tipos en *Haskell*. *La Sección 3* describe algunas de las alternativas de representación de la información consideradas de interés en la investigación. *La Sección 4* presenta los recursos de software desarrollados y puestos a disposición de la comunidad. Finalmente, la Sección 5 concluye el trabajo y presenta líneas de trabajo futuro.

## 2. PARADIGMA Y DEFINICIÓN DE TIPOS EN HASKELL

(Field y Harrison, 1988) postuló que la manera en que actualmente los programas de software son escritos *(i.e. Paradigma Imperativo)* no es la más conveniente ni la más cómoda. Ésto sugiere la idea de desarrollar nuevas formas o paradigmas para poder representar las situaciones del mundo real, y que ésto se pudiera realizar de forma más versátil y cómoda.

Por *Paradigma Declarativo* se entiende que es un estilo de programación en el que el programador especifica *qué* debe computarse y no *cómo* debe hacerse. Según este principio (García, 1997) define un programa como la unión de lógica y control *(i.e., programa = lógica + control*), donde el componente lógico determina el significado, mientras que el de control sólo su eficiencia. Así la tarea de programar se centra en la lógica, puesto que se asume el control automático a la máquina. La característica fundamental del *Paradigma Declarativo* es el uso de la lógica como lenguaje de programación.

No obstante, dependiendo del tipo de lógica existen varios estilos de programación en este paradigma, entre los principales se destacan los siguientes: *(1) Funcional (i.e. lógica ecuacional), (2) Relacional (i.e. lógica clausal) y (3) De Tipos (i.e. lógica heterogénea)* (Straneo y Amo, 2009).

La diferencia entre el enfoque *declarativo* y el *imperativo* es que el primero implica la resolución de problemas mediante la *especificación* de condiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y lo resuelven, pero no necesariamente *cómo* deben resolverse, mientras que el *imperativo* se enfoca en la construcción del *cómo* se deben de resolver los problemas, mediante un conjunto de procedimientos estructurados (Fahland et al, 2009).

Para ilustrar la diferencia entre el enfoque imperativo y el declarativo considérese el problema de construir una función *f* que reciba como argumento un número natural *n* y retome la suma de los naturales desde 1 hasta *n*, es decir:

$$f(n) = \sum n = 1$$

En un lenguaje imperativo como C, se podría definir la función de la siguiente forma:

```
int f(int n = 1) {
 int i;
 int suma = O;
 for (i = l; i <= n; i++)
     suma=+ i;
 return suma;
}
```

La *Programación* Funcional consiste en construir definiciones y usar la computadora para evaluar expresiones. El objetivo está en que el programador construya una función para resolver un problema dado. Ésta función, que puede implicar varias subfunciones, se expresa en notación que obedece a los principios matemáticos formales (Bird y Walder, 1988).

Un rasgo característico de la *programación funcional* es que el significado de una expresión es su valor y la tarea de la computadora es simplemente obtenerla. De ello se deduce que las expresiones en un lenguaje funcional pueden ser construidas, manipuladas y razonadas como cualquier otro tipo de expresión matemática (Bird y Walder, 1988). El resultado, como esperamos para justificar, es un marco conceptual para la programación que es a la vez simple, conciso, preciso y potente.

La *Programación Funcional* se basa en la solución de un problema que parte de tres principios básicos: la simplificación del objetivo en *sub-objetivos*, la reutilización de código y la reducción del tiempo en pruebas (Trejos, 2011).

Ésta forma de programar se aparta de la concepción de máquina de (Von-Neumann, 2007) ya que él se basaba en la utilización de memoria, por lo cuál los programas poseen variables. Sin embargo en la *programación declarativa*, las variables no son necesarias, ya que no se considera la memoria como tal, pudiéndose entender a un programa como una evaluación continua de funciones sobre argumentos u otras funciones. Esto quiere decir que posee un estilo de computación que sigue la evaluación de funciones matemáticas y evita los estados intermedios y la modificación de éstos (Rivadera, 2008).

La *Programación Funcional* tiene como objetivo la utilización de funciones matemáticas puras sin efectos colaterales y, por tanto, sin asignaciones destructivas. El esquema del modelo funcional es similar al de una calculadora. Se establece una sesión interactiva entre *sistema* y *usuario*: el usuario introduce una *expresión inicial* y el sistema la evalúa mediante un proceso de reducción. En este proceso se utilizan las *definiciones de función* realizadas por el *programador* hasta obtener un *valor* no reducible (Labra, 1998). El *Paradigma Funcional* se basa en un modelo llamado *Modelo de la Calculadora* (Rechenberg, 1990) como se sintetiza en la Figura 1.

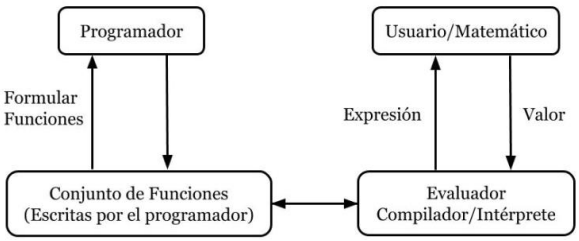


Figura 1. Modelo del esquema funcional

Es un lenguaje de programación avanzado puramente funcional. Permite el desarrollo rápido de un software robusto, conciso y correcto. Con un fuerte soporte para la integración con otros lenguajes, concurrencia y paralelismo integrados, depuradores, perfiladores, bibliotecas ricas y una comunidad activa, *Haskell* facilita la producción de software flexible, mantenible y de alta calidad (O'Sullivan et al, 2008). El lenguaje funcional por excelencia *Haskell* posee de forma nativa una sintaxis para poder crear objetos complejos, éstos se crean apartir de la palabra reservada *data*, con el cuál se pueden modelar distintos tipos de datos y con mejor versatilidad.

Dadas las formas en las cuáles los *conceptos del mundo real* pueden ser representados en *Haskell* mediante su definición versátil de tipos de datos, se consideran en este sentido, las siguientes ventajas en la que éste tipo de estructuras tendrían sobre el modelo relacional de base de datos:

1. *Soporte a objetos complejos.* Mediante estructuras flexibles para construir objetos estructuralmente complejos del mundo real, es decir, que el *experimentador* puede moldear de forma versátil cualquier interpretación que necesite representar sin pérdida de significancia (Gómez y Mariea, 2007).
2. *Polimorfismo.* Pudiendo aplicarse el mismo método sobre los distintos objetos creados *(i.e. a partir de la estructura previa creada)* de diferente tipo (sobrecarga), y dando la posibilidad de definir distintos métodos con el mismo nombre en función de sus parámetros de entrada (Buneman y Ohori1996).
3. *Extensibilidad.* Mediante la capacidad de crear nuevos tipos a partir de los ya existentes brindando un soporte al constante crecimiento de la complejidad estructural de los datos (Jones, 1995).

# 3. REPRESENTATIVIDAD DE CONCEPTOS Y SUS APLICACIONES

## 3.1 Representación de Conceptos: Modelo Relacional vs. Modelo Funcional

Muchos autores coinciden en que el modelo relacional carece de una forma de representar varios de los conceptos del mundo real considerados estructuralmente *polimorfos* (Gómez 2007; Quiroz 2003). Ésta argumentación radica en que el modelo relacional representa los conceptos mencionados con transformaciones previas o mediante la definición de entidades y relaciones poco naturales.

Por otra parte, un aspecto clave a resaltar es que el lenguaje estándar para el tratamiento de base de datos relacionales (SQL, *por sus siglas en inglés)* carece de completitud computacional, ya que no tiene definido por naturaleza instrucciones de control *IF, WHILE, DO*, es posible incorporarlo en lenguajes procedimentales tales como *Java*, pero ésta técnica produce una *no correspondencia de impedancias* por estar mezclando diferentes paradigmas de programación (Gómez, 2007). Sin embargo, *Haskell* es un lenguaje de propósito general y solo nos permite la manipulación de datos mediante estructuras de datos de tipo *registro*, a partir de esto sólo se necesitaría poder persistir los tipos que fueron creados a partir de la estructura versátil que incorpora *Haskell*.

## 3.2 Aplicaciones de la Representatividad de Conceptos con Haskell

Un programa de *software* suele estar particionado en módulos o funciones, las cuáles reciben un grupo de entradas y a su vez proporcionan una salida o resultado. Similar a una estructura de composición, de un modelo de proceso de negocio que consiste en departamentos, cada una de las cuáles pueden contener pequeños pasos (operaciones), ésto a su vez genera una serie de resultados para el negocio (Fahland y col, 2009).

Por otra parte, es importante mencionar que los programas funcionales no manejan el concepto de *variables*, una vez que se les da un valor, nunca cambian. Por lo general, los programas funcionales no tienen ningún *efecto secundario*. Una llamada de función no puede tener otro efecto que el de calcular su resultado. Esto elimina una fuente de errores posibles y también hace que el orden de ejecución sea irrelevante. Libera al programador de la carga de prescribir el flujo de control. Desde que pueden ser evaluadas en cualquier momento, se pueden sustituir libremente las variables por sus valores y viceversa, es decir, que los programas son *referencialmente transparentes*. Esta libertad ayuda a hacer que los programas funcionales sean más manejables matemáticamente que sus similares imperativas (Hughes, 1989).

### 3.2.1 Tipo de Datos del Sistema de Información Geográfica (GIS)

A día de hoy, los tipos de datos *GIS* son representados mediante el modelo relacional de datos. En la actualidad existe una demanda cada vez más creciente de la *geocomputación*. El modelo de representación *ráster* es el modelo principal para representar atributos espaciales (Qin et al, 2014).

Los datos del tipo *GIS* representan objetos reales del mundo, como calles, terrenos, elevaciones, mediante datos digitales. Actualmente éste tipo de datos se almacenan mediante mallas rectangulares o componentes geométricos. A continuación en la Tabla 1 se describe la comparación del modelado de un esquema geográfico entre *relacional y funcional:*

Tabla 1. Representación de datos geográficos en *Haskell*

| Modelo Relacional | Modelo Funcional |
|---|---|
| Se define la tabla con la forma básica que tendra el contido del dato GIS.<br>`create table Pais {`<br>`nombre VARCHAR(30),`<br>`superficie VARCHAR(30),`<br>`coordenada VARCHAR(30)`<br>`};`<br>Aquí se nota que el campo *coordenada* es una cadena de caracteres, donde ahí deben estar los datos de latitud y longitud. Para poder extraer esos datos de éste campo, se depende de algún procedimiento extra para poder realizar la extracción de la latitud y la longitud. | Se define un nuevo tipo de dato (función) País:<br>`data Coordenada = Coordenada {`<br>`latitud :: Double`<br>`longitud :: Double`<br>`} deriving (Show)`<br>`let coordenada1 = Coordenada(-32.323213, -21.323232)`<br>`let coordenada2 = Coordenada(95.3232, -31.434343)`<br>`let coordenadas = fromList [coordenada2, coordenada2]`<br>`data Pais = Pais {`<br>`nombre :: String`<br>`, superficie :: String`<br>`, departamento :: String`<br>`, coordenadas :: [Coordenada]`<br>`} deriving (Show)`<br>Se puede notar en la representación funcional que el dato *coordenada* es un tipo complejo, que a su vez se puede definir en longitud y latitud, dependiendo del sistema de coordenadas que se esté utilizando. Con ésto se demuestra que *Haskell* permite manejar de manera más versátil los datos estructurados y representarlos de forma más natural al mundo real. |

### 3.2.2 Detección de Patrones Musicales

El creador de la conocida aplicación móvil de detección de canciones denominado *Shazam*, Avery Li-Chun Wang detalla en (Wang, 2003) explícitamente como realizó la aplicación, con una explicación a nivel matemático, donde utilizó un modelo de reconocimiento de patrones de canciones utilizando la frecuencia de la canción y a partir de ésta frecuencia fue creando un gráfico del cuál quitó la información necesaria para hacer la comparación de canciones dentro de su base de datos. Mediante el manejo polimórfico de datos en *Haskell* nos permite manejar de manera más versátil éste tipo de datos, a continuación en la Tabla 2 se ve una comparación básica de la estructura.

Tabla 2. Representación de patrones musicales en *Haskell*

| Representación Relacional | Representación Funcional |
|---|---|
| Una base de datos relacional tradicional posee una escalabilidad horizontal deficiente (Pingel, 2008) el cuál impide que algunos esquemas de base de datos no se plasmen de forma natural. EÍ esquema de un modelo de patrones musicales se podría representar de forma de jerarquía, mostrando así las frecuencias de un mismo formato las cuáles provienen de frecencias padres o principales, así tambien como las canciones que representan un conjunto de frecuencias.<br>En el modelo relacional, para la representación de la estructura de datos mencionada anteriormente para modelar los datos que maneja el creador de *Shazam*, se crea una tabla con la información de la canción y posteriormente una tabla que va relacionada a ésta, para poder identificar las frecuencias que tiene.<br>Él inconveniente de ésto es que se desataría una redundancia de datos, por que las frecuencias y distancias pueden ser iguales a varias canciones:<br>`create table Cancion {`<br>`nombreCancion VARCHAR2(30),`<br>`id_frecuencia INTEGER`<br>`};`<br>`create table Frecuencias {`<br>`id_frecuencia INTEGER,`<br>`frecuencia1 VARCHAR2(30),`<br>`frecuencia2 VARCHAR2(30),`<br>`distancia INTEGER`<br>`};` | Se define la estructura de datos para la persistencia de la canción, con la cantidad de distancia entre frecuencias para la comparación posteriormente. A partir de éstos datos se realiza la comparación de distancia entre frecuencias entre la música que fue grabada y las músicas de la base de datos.<br>`data Cancion =`<br>`data Informacion1 = Informacion {`<br>`frecuencia1 :: String,`<br>`frecuencia2 :: String,`<br>`distanciaEntreFrecuencias :: Int`<br>`} \|`<br>`data Informacion2 = Informacion {`<br>`frecuencia1 :: String,`<br>`frecuencia2 :: String,`<br>`distanciaEntreFrecuencias :: Int`<br>`} \|`<br>`data Informacion3 = Informacion {`<br>`frecuencia1 :: String,`<br>`frecuencia2 :: String,`<br>`distanciaEntreFrecuencias :: Int`<br>`} deriving (Show)` |

### 3.2.3 Base de Datos con Restricciones

Este estudio aporta una alternativa de representación a lo ya expuesto por (Gómez y Teresa, 2007) donde define una Base de Datos con Restricciones como cualquier colección definida de relaciones no definidas con restricciones y presenta el *LORCDB* como un Gestor de Bases de Datos Objeto-Relacionales de Restricciones.

En el mundo de la programación funcional la forma en la que las ecuaciones e inecuaciones se expresan de manera flexible y natural, ya que ésta estilo de programar se basa en los principios algebraicos matemáticos básicos, ésto podría resultar una gran ventaja para la definición de las restricciones (Hughes, 1999). En la Figura 2 se representa un modelo de una base de datos con restricciones, donde las restricciones están escritas en *Haskell*, posteriormente éstas restricciones podrían ser persistidas.

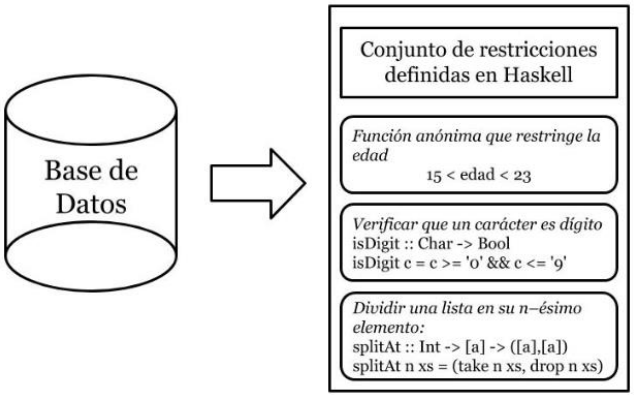


Figura 2. Representación de las restricciones en *Haskell*

# 4. ALMACENAMIENTO DE INFORMACIÓN POLIMORFA

Como se ha podido contemplar a lo largo de este estudio, el sistema de tipos de *Haskell* es uno de los más sofisticados que existen (Rodriguez et al 2008, Marlow y Jones 2004). Es un sistema *polimórfico*, que permite una gran flexibilidad de programación, pero a la vez mantiene la correctitud de los programas, *Haskell* utiliza un sistema de inferencias de tipos, es decir sabe el tipo resultante de una expresión, por lo que las anotaciones de tipo de un programa son opcionales (Rivadera, 2008).

Para lograr el almacenamiento de las definiciones de tipos de datos abordados en la sección anterior, mediante la utilización de *Haskell*, se implementó una librería agregando mecanismos conforme a la identificación de factores mínimos hacia una *persistencia primitiva*. La serialización de los datos "no tradicionales" y conceptualmente polimorfos son estructurados en el formato *JSON* (Clockford, 2022). Al momento de serializar los datos, previo a una persistencia, fue imprescindible identificar los siguientes factores:

1. *Identificación unívoca.* Ante el inconveniente de la identificación de los objetos al seralmacenados, se recurrió a la solución de establecer un campo adicional dentro de la estructura de datos que será almacenada, un campo que se utilizaría como clave para la identificación de esa entidad.
2. *Representación de datos sin variaciones significativas.* Ya que dentro del modelo relacional de datos que son las que más se utilizan actualmente, pudimos notar que al almacenar los datos, los mismo sufren una serie de transformaciones antes de ser almacenadas (Quiroz, 2003), la información debe ser almacenada de forma más natural y que al pasar por todo el procedimiento de almacenaje los datos sufran la mínima cantidad de modificaciones o transformaciones hasta llegar al archivo final.
3. *Ciclo de operaciones sobre el archivo.* Entre las operaciones sobre los datos más importantes y que no pueden faltar dentro de ningún tipo de base de datos se encuentran; la creación del objeto o instancia, consulta o recuperación, e inserción de los datos del exterior (Elmasri et al,

2017). Éstas operaciones son esenciales para el tratamiento de la información, las cuáles deben ser contemplados dentro de un mecanismo de persistencia primitiva.

Ya con la aplicación de los métodos de manipulación necesarios más los *metadatos* asociados para la búsqueda o identificación de las instancias, finalmente la estructura *JSON* compuesta de datos y *metadatos*, fueron transformados a un formato binario, *BSON (Binary JavaScript Object Notation)* el cual es una representación binaria de estructuras de datos basados en el lenguaje *JSON*, para el almacenamiento en el sistema de archivos, esto debido a que es conocido que la eficiencia y el rendimiento de los archivos binarios es superior para la persistencia que el formato *JSON* (Maeda, 2012).

## 5. CONCLUSIÓN

Durante la última década la mayoría de las investigaciones en el descubrimiento del conocimiento han hecho hincapié en el uso de técnicas de preprocesamiento de datos asumiendo que la información contenida en el conjunto de datos *(i.e. dataset)* es la representación propia del *mundo real*.

Sin embargo, en este estudio se ha dado cuenta de la versatilidad representativa del modelo funcional para la representación de conceptos *estructuralmente complejos*; como ser, tanto aquellas relacionadas con las aplicaciones en sistemas de información geográfica *(GIS)*, las restricciones basadas en ecuaciones e inecuaciones matemáticas, como así también operaciones entre patrones musicales. Para tal efecto, el nivel de versatilidad del modelo funcional ha sido expuesto a la luz mediante su contraste con el modelo relacional *(i.e. representación tabular*) generalmente conocido como única opción por parte de los experimentadores dentro de los diversos campos del saber.

Puesto que la representación del mundo real debe poder almacenarse y recuperarse para su tratamiento posterior, la segunda contribución más importante del presente trabajo, ha sido tanto la implementación de una librería disponible para la comunidad en (Gómez, 2022) para la persistencia de las definiciones *a medida* de tipos de dato en *Haskell* como así también la identificación de los *factores* a ser considerados en trabajos similares y conocido teóricamente como *persistencia de primera generación* en lenguajes de programación de propósito general.

Finalmente, sería recomendable que futuras investigaciones abordasen los efectos de la versatilidad en la representación de los conceptos del mundo real mediante la cuantificación del error propagado desde la recolección de las muestras hasta el análisis de los datos en áreas de interés para la academia y la industria. En adición, se subraya la necesidad de seguir transitando hacia la conversión de *Haskell* en un lenguaje de programación persistente.

## REFERENCIAS


Aballay, M., De Battista, A. and Gagliardi, E.O., 2017. *Uso de bases de datos espacio-temporales para la atención de eventos de emergencia.* In XXIII Congreso Argentino de Ciencias de la Computación La Plata, 2017, pp. 13-15.

Backus, J., 1978. *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*. Communications of the ACM, 21(8), pp. 613-641.

Bird, R. and Wadler, P., 1988. *Introduction to Functional Programming.* Series in Computer Science. Prentice Hall International, 20, pp. 547-559.

Buneman, P. and Ohori, A., 1996. *Polymorphism and type inference in database programming.* ACM Transactions on Database Systems (TODS), 21(1), pp. 30-76.

Clofkford D. (2022). *Introducing JSON.* Disponible en: < https://www.json.org/json-en.html> [Consultado 10 Agosto 2010].

Curry, H., 2022. *Haskell Language*. [online] Haskell.org. Available at: <https://www.haskell.org> [Consultado 13 Agosto 2019].

Dearle, A., Kirby, G.N. and Morrison, R., 2009, July. *Orthogonal persistence revisited*. In International Conference on Object Databases. Springer, Berlin, Heidelberg. pp. 1-22.

Elmasri, R., Navathe, S.B., Elmasri, R. and Navathe, S.B., 2000. *Fundamentals of Database Systems*</Title. Addison-Wesley</publisher.

Field, A.J. and Harrison, P.G., 1988. *Functional programming.* Addison-Wesley.

Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M. and Zugal, S., 2009. *Declarative versus imperative process modeling languages: The issue of understandability.* In Enterprise, Business-Process and Information Systems Modeling. Springer, Berlin, Heidelberg , pp. 353-366.

Garcıa, A.J., 1997. *La Programación en Lógica Rebatible: su definición teórica y computacional* (Doctoral dissertation, Master's thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahıa Blanca, Argentina).

Gómez, W., 2022. *GitHub - wildogomez/haskell-persist-1stGen*. [online] GitHub. Disponible en: <https://github.com/wildogomez/haskell-persist-1stGen.git> [Consultado 13 Agosto 2022].

Hughes, J., 1989. *Why functional programming matters*. The computer journal, 32(2), pp. 98-107.

Hughes, J., 1999, September. *Restricted data types in Haskell.* In Haskell Workshop (Vol. 99).

Jones, M.P., 1995, May. *Functional programming with overloading and higher-order polymorphism.* In International School on Advanced Functional Programming (pp. 97-136). Springer, Berlin, Heidelberg.

Labra, G., 1998. J. *Introducción al Lenguaje Haskell.*

Maeda, K., 2012, May. *Performance evaluation of object serialization libraries in XML, JSON and binary formats.* In 2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP) (pp. 177-182). IEEE.

Marlow, S. and Jones, S.P., 2004. *The glasgow haskell compiler*.

Milton, S.K. and Kazmierczak, E., 2004. *An ontology of data modelling languages: A study using a common-sense realistic ontology*. Journal of Database Management (JDM), 15(2), pp. 19-38.

O'Sullivan, B., Goerzen, J. and Stewart, D.B., 2008. *Real world haskell: Code you can believe in.* " O'Reilly Media, Inc.".

Quiroz, J., 2003. *El modelo relacional de bases de datos.* Boletín de Política Informática, 6, pp. 53-61.

Qin, C.Z., Zhan, L.J., Zhu, A.X. and Zhou, C.H., 2014. *A strategy for raster-based geocomputation under different parallel computing platforms*. International Journal of Geographical Information Science, 28(11), pp. 2127-2144.

López, G.Ó.M.E.Z. and Teresa, M., *LORCDB: Gestor de Bases de Datos Objeto-Relacionales de Restricciones.* Sevilla, 2007 (Doctoral dissertation, Tesis Doctoral (Doctora en Informática). Universidad de Sevilla. Departamento de Lenguajes y Sistemas Informáticos. Disponible en< http://www. lsi. us. es/docs/doctorado/tesis/Memoria-Tesis-MTGomez. pdf).

Rivadera, G.R., 2008. *La programación funcional: un poderoso paradigma*. Cuadernos de Ingeniería, (3), pp. 63-77.

Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O. and Oliveira, B.C.D.S., 2008. *Comparing libraries for generic programming in Haskell*. ACM Sigplan Notices, 44(2), pp. 111-122.

Rechenberg, P., 1990. *Programming languages as thought models*. Structured Programming, 11(3), pp. 105-116.

Straneo, H.P. and Amo, F.A., 2009, March. *A holonic model of system for the resolution of incidents in the software engineering projects*. In 2009 International Conference on Computer and Automation Engineering, pp. 79-86

Trinder PW. *A functional database* (Doctoral dissertation, University of Oxford).

Trejos, O.I., 2011. *Consideraciones sobre la Evolución del Pensamiento Humano a Partir de los Paradigmas de Programación.* Scientia et technica, 2(48), pp. 281-286.

Wang, A., 2003, October. *An industrial strength audio search algorithm.* In Ismir (Vol. 2003, pp. 7-13).