Genericidad de funciones: El *quid* para la incorporación de dominios en un Sistema Funcional Inductivo basado en Haskell

Félix Gómez, Pedro González, Derlis Salinas,
Gustavo Sosa-Cabrera, María Elena García-Díaz

Universidad Nacional de Asunción, Paraguay
{fgomez,pgonzalez,dsalinas,gdsosa,mgarcia}@pol.una.py
http://www.pol.una.py

Resumen Hoy en día, una inconmensurable cantidad de programas informáticos se encuentran en ejecución generando información propia de su comportamiento, estos tipos de historial son amplia y generalmente conocidos como *Logs*. Sin embargo, a pesar de los avances en la *inferencia funcional inductiva* para trabajar con los datos, hasta el momento se ha prestado escasa atención a la automatización del procesamiento analítico de estos tipos de registros de eventos. En este sentido, puesto que la alta expresividad de los *Lenguajes de Programación Declarativos* es una noción ampliamente aceptada, en este trabajo se aborda las implicancias prácticas de la *Programación Funcional Inductiva* aplicado en el dominio específico de los *Logs* para su inferencia asistida *ex professo*.

Palabras Claves: Funciones Genéricas · Logs · Dominio Específico · Conocimiento de Fondo · Inferencia Funcional Inductiva · Haskell

1. Introducción

A día de hoy, cada vez es más difícil ignorar la cantidad no conmensurable de programas informáticos de diverso índole que se encuentran en ejecución generando registros sobre su propio funcionamiento, ya sea informativamente o indicando anomalías. Estos tipos de historial son ampliamente conocidos como *Logs* y se constituyen como un recurso vital para la administración de los sistemas ya que permiten tanto el monitoreo periódico como así también el diagnóstico de problemas. En adición, puesto que el gran volumen de registros de eventos serializados a diario hace que sea poco práctico inspeccionar manualmente los *logs*, analizarlos de manera efectiva sigue siendo un gran desafío [10].

Aunque una importante labor se ha llevado a cabo, tanto por parte de la academia, como también por la industria; las 2 alternativas propuestas hasta el momento para el análisis de *logs* consisten básicamente en (1) la extracción de patrones mediante expresiones regulares (e.g. grok[1]) ó (2) más recientemente, por medio de aplicación de técnicas tradicionales de minería de datos (TTMD).

[1] https://logz.io/blog/logstash-grok

Sin embargo, ambos enfoques respectivamente presentan una serie de inconvenientes importantes como ser (1) la escritura manual de reglas consume mucho tiempo y es propenso a errores; (2) la fase de aprendizaje de las *TTMD* poseen una complejidad temporal en función al tamaño del conjunto de datos y a la vez requieren una considerable cantidad de registros para evitar el *sobre-ajuste*.

En este sentido, prolifera la preocupación en entornos *TICs*, puesto que ínterin los *logs* crecen en cantidad, volumen y complejidad; los administradores de sistemas deben solucionar los problemas lo más rápido posible para satisfacer a los clientes, proteger la imagen corporativa y mantener las ventas [9].

Por otra parte, estudios previos recientes han señalado que a diferencia de los enfoques estándar de aprendizaje automático utilizado en las *TTMD*, la Programación Funcional Inductiva (IFP, *por sus siglas en inglés*) aborda el problema de aprender programas pequeños pero complejos, de unos pocos ejemplos representativos de *entrada/salida*, generados a medida que el usuario transforma una o muy pocas instancias o campos particulares de los datos, como lo hacemos los humanos análogamente [2]. Asimismo, es una noción ampliamente aceptada la gran expresividad de la Programación Declarativa del cual desciende la *IFP*.

Surge por tanto, la oportunidad de dar luz a la investigación de la *IFP* en el contexto de la automatización del proceso de inferencia asistida de *logs* y de hecho, este es el primer estudio en el que se realiza la incorporación de este tipo de *dominio* en un *Sistema Inductivo Funcional* (SIF).

Los resultados de esta investigación demuestran que mediante un tratamiento adecuado de la *genericidad de funciones* como factor principal, es posible la explotación de las ventajas del *IFP* en cuanto a inferencia de *logs* se refiere.

2. Trabajos Relacionados

Una de las razones del éxito de los sistemas *IFP* es el uso de lenguajes específicos de dominio (DSL, *por sus siglas en inglés*). Así *FlashFill*[3], como ejemplo de ello, puede realizar transformaciones sintácticas de cadenas utilizando formas restringidas de expresiones regulares, condicionales y bucles en tablas de hojas de cálculo. Sistemas *IFP* que también tuvieron protagonismo son *Igor-II* [6] y *ADATE* [11]. Sin embargo, éstos no han sido actualizados recientemente ni puestos en práctica. *Igor-II* impone una restricción estricta en el conjunto de ejemplos dado como especificación y *ADATE* requiere una gran habilidad para la síntesis de programas simples. Además, debido a la ausencia de memorización, tienen una desventaja en la velocidad práctica en comparación con *MagicHaskeller*, que puede iniciar la síntesis con su tabla de memorización llena de expresiones[5], como se menciona en el apartado 3.3 de la siguiente sección.

3. Materiales y Métodos

3.1. Logs

Registro del "qué sucedió, cuándo y por quién"del sistema. Los *logs* de aplicaciones graban cronológicamente las operaciones durante el funcionamiento de

la aplicación y vienen categorizados en tres niveles principales, donde representan la información de depuración, información narrativa o eventos de la lógica del negocio. A continuación, los tipos de *logs* de interés para este estudio.

***Logs* de *Spring*.** Se refiere al proyecto *Spring Framework*[2]. Un *framework web* es un conjunto de componentes que ofrece mecanismos para la implementación de las capas del negocio y presentación web [1]. *Spring Framework* proporciona soporte para diferentes arquitecturas de aplicaciones, incluidos mensajería, datos transaccionales y persistencia en ambiente web.

***Logs* de *Apache*[3].** Para administrar un servidor web de manera efectiva, es necesario obtener comentarios sobre la actividad y el rendimiento del servidor, así como sobre cualquier problema que pueda estar ocurriendo. El servidor HTTP *Apache* proporciona capacidades de registro muy completas y flexibles, y con ellos describimos dos divisiones principales, *Error Log* y *Access Log*.

3.2. Programación Declarativa

Se caracteriza por una *alta expresividad* basado en la utilización de bloques de construcción como funciones, recursión y el uso de patrones, de manera a especificar más la solución antes que su cálculo de bajo nivel [8].

Programación Funcional. Deriva de la *Programación Declarativa*, prescinde del uso de bloques condicionales o ciclos. Su característica principal reside en el uso de expresiones y funciones, donde un programa es una función pura. Siendo función en términos matemáticos obedeciendo sus fundamentos.

Función total. Toda función relaciona dos conjuntos a través de una operación algebraica. Una función total es aquella que está definida para todos los posibles valores del conjunto dominio.

Programación Inductiva. (IP, *por sus siglas en inglés*) desarrolla métodos en base al razonamiento inductivo, esto es, de lo específico a lo general.

IFP. Una vertiente en auge de la *IP* es la Síntesis de Programas Inductivos (IPS, *por sus siglas en inglés*). Generalmente en IP, infinitos programas semánticamente diferentes cumplen con una misma especificación, por lo tanto, se necesita definir criterios para elegir entre ellos. Estos criterios se denominan sesgo inductivo, que se dividen en: (1) Sesgo de restricción: El sistema *IPS* solo es capaz de generar un subconjunto específico de las funciones de un dominio, debido a que tiene un lenguaje restringido, incapacidad computacional o sus operadores no logran alcanzar cada programa; (2) Sesgo de preferencia: El sistema depende del orden en que se explora el espacio del problema, causando que la organización de las soluciones dependa del orden de exploración.

La búsqueda en los métodos *IPS* funcionales se basa en generar-y-probar. A diferencia de técnicas de aprendizaje automático convencionales donde los ejemplos de E/S son utilizados para la construcción de modelos, en *IPS*, los ejemplos de E/S se utilizan única y exclusivamente para probar los programas generados. La *IFP* es el marco para la programación automatizada para lograr la síntesis de programas funcionales recursivos a partir de ejemplos de E/S.

[2] https://github.com/spring-projects/spring-framework
[3] https://httpd.apache.org

3.3. Domain Specific Induction (DSI)

Karmiloff-Smith (1994) define a un *dominio* como el "conjunto de representaciones que sostiene un área específica de conocimiento" (p. 23)[7]. Ejemplos de dominio podrían ser las fechas, los correos electrónicos, nombres de personas y *logs*. DSI es un sistema que pretende la automatización del procesamiento de datos o *Data Wrangling* mediante la implementación del software *MagicHaskeller*[4], sumado a un conjunto de funciones definidas para un cierto dominio y una interfaz de acceso web para la interacción humano-computador.

Por otra parte, la interfaz web permite, a partir de datos de E/S, dónde la Entrada es el valor a ser procesado y la Salida es el resultado esperado, transformarlo a un predicado para utilizarlo como valor de entrada en el *MagicHaskeller*. Éste retorna como resultado la función f que satisface el predicado. Luego la función f es aplicada al resto de entradas del *DSI* a fin de obtener los valores para cada columna de salida. Desde la interfaz web, se envían los predicados al *MagicHaskeller* a través de un puerto abierto en el servidor, donde se ejecuta una instancia del sistema para cada dominio abarcado. *MagicHaskeller* al encontrar una o varias funciones que resuelven el problema enviado en el predicado, lo devuelve en una cadena con el siguiente formato:

```
(\a -&gt; function1 a), (\a -&gt; function2 a), (\a -&gt; function3 a)
```

El proceso *data wrangling* culmina cuando el sistema *DSI* selecciona la primera función de la lista: `function1 a`, dónde `a` se reemplazará por el valor de cada entrada para evaluar la función hasta completar todos los campos de salida.

MagicHaskeller. Un sistema IFP que adopta el enfoque de generar-y-probar, llamado *MagicHaskeller*, el cual al recibir un conjunto de ejemplos de E/S, primero infiere el tipo de expresiones deseadas. Luego, genera expresiones del mismo tipo que se pueden expresar con las funciones de la biblioteca de componentes, combinadas con aplicaciones de funciones y abstracciones *lambda*. La generación se realiza de forma exhaustiva a partir del más corto. Luego se comparan con la especificación en orden y las que pasan la prueba son las funciones sintetizadas.[4]

MagicHaskeller puede sintetizar programas funcionales cortos sin restricción del espacio de búsqueda en base a cualquier conocimiento previo, gracias a su gran tabla de memorización. El sistema recibe un predicado, en este caso, una expresión *lambda* con la siguiente estructura: $\backslash f$ -> f *"entrada"*==*"salida"*. Para reducir el espacio de búsqueda, se necesita limitar la cantidad de funciones, conocido como *Conocimiento de Fondo Específico de un Dominio* (DSBK, por sus siglas en inglés), para ello, se generan los prototipos de las funciones en el archivo "primitives", con prioridades por función, donde 0 es el más prioritario; a este número de funciones se lo denomina Amplitud o *Breadth (b)*. Otro valor importante es la Profundidad o *Depth (d)*, que limita la profundidad con la cual se combinarán las funciones disponibles para solucionar un predicado. La relación entre estos dos valores se da en la cota superior asintótica, siendo una de orden potencial exponencial dado $O(b^d)$. Considerando esto, el valor de d es clave en la fase de entrenamiento, ya que con una cantidad alta de b y d, la fase

de entrenamiento podría no culminar, debido a que el computador no posee los suficientes recursos para realizar todas las combinaciones posibles y almacenarlas en su Tabla de Memorización, obedeciendo al criterio de *sesgo de restricción*.

4. Experimentos

4.1. Réplica de experimentos y verificación de resultados de *DSI*

Con el objetivo de montar el escenario experimental del presente estudio validando a la vez su correctitud, se han verificado y reproducidos los resultados obtenidos en *DSI*[2], basado en su implementación[4] se comprueba la ejecución e interacción con *MagicHaskeller* en la búsqueda de funciones, con dominios como fechas, correos, nombres y palabras. Para realizar los experimentos se utiliza una máquina virtual con una vCPU, dos gigabytes de memoria RAM y un disco de almacenamiento de 25 gigabytes. El sistema operativo utilizado es Ubuntu 18.04.2 para arquitecturas x64. Acorde a estas especificaciones, se ha empleado un aproximado de cuatro horas reloj en modificaciones para hacerlo funcionar.

4.2. Definición de nuevo dominio y la generación de funciones

En este estudio, se amplia el alcance de *DSI*[2] con la inclusión del dominio de *Logs*. Inicialmente incorporando *logs* del framework de *Spring*, ya que éste tiene una estructura básica sumamente completa, definida en la Sección 3.1.

Se determina generar las funciones para la extracción de porciones de datos como las fechas, el nivel del *log*, el nombre de la clase Java en donde se registró el *log*, y el mensaje del *log*. Basado en ello, se generan las correspondientes funciones en Haskell: (1) *extractDate*; (2) *getLogLevel*; (3) *getClass*; (4) *getMessage*.

Conforme a la estructura de un *log* de *Spring*, se generan estas funciones sin antes haber analizado a profundidad las características de las funciones ya presentes en *MagicHaskeller* ni las funciones generadas para *DSI*; el punto principal es la construcción de funciones que realicen de la manera más sencilla posible la extracción del dato solicitado a partir de una cadena de texto. Para añadir las funciones a *MagicHaskeller* se sigue el siguiente procedimiento:

- Crear un módulo en el directorio */MagicHaskeller*. Realizar las importaciones necesarias y agregar las nuevas funciones correspondientes al dominio.
- En el archivo *Individuals.hs*, se debe agregar el prototipo de cada función nueva, dentro del parámetro "totals", con el siguiente formato:
  `++ $(p [| function :: [Char] ->[Char] |] )`
- Agregar el nombre del módulo en el archivo *MagicHaskeller.cabal*, dentro del parámetro "Build-depends".
- Compilar el programa desde el directorio */MagicHaskeller*, mediante:
  `cabal install --global`
- Generar las primitivas con el comando:
  `MagicHaskeller --dump-primitives > primitives`

[4] https://github.com/liconoc/DataWrangling-DSI

En pruebas de ejecución se somete a *MagicHaskeller* con predicados como:

```
f "2017-03-03 13:02:50.608  INFO ... info message" == "IndexController"
```

Se obtiene la función `getClass` que convierte el *input* en el *output* esperado. *MagicHaskeller* puede retornar como solución una función compuesta equivalente a la buscada; la función `getClass` es equivalente a `extractDate(getClass)` ya que ante un mismo *input* retornan exactamente el mismo *output*.

Considerando lo mencionado en la Sección 3.3, una desventaja de *MagicHaskeller* es el potencial exceso de tiempo en la fase de entrenamiento. La cantidad de tiempo está directamente ligada a los parámetros *depth* y *breadth*. Se maneja como hipótesis la posibilidad de que el agregado de predicados adicionales sobre un dominio pudiera mejorar el entrenamiento y los resultados se encuentren en instancias de ejecución con menores valores de *depth*. Para tal efecto, se efectúa un experimento buscando obtener variaciones, alteración de resultados o algún comportamiento anómalo que pudiera registrarse. Se añadió diez predicados diferentes por función, totalizando cincuenta nuevos predicados. Añadidos los predicados, los resultados de las pruebas son exactamente iguales a cuando *MagicHaskeller* se ejecuta sin ningún predicado añadido. Esto confirma que los predicados adicionales y pretendidos como *DSBK* para el *MagicHaskeller* no generan alteraciones en su comportamiento en la búsqueda de funciones, obedeciendo la teoría de los sistemas *IPS* (Sección 3.2).

4.3. Ampliación del dominio de *Logs*

Se contempla la necesidad de ampliar el dominio agregando otra estructura de *logs*, mediante *logs* del servidor web *Apache* (Sección 3.1). Para tal efecto, se generan funciones para la extracción de fechas, inicialmente para *logs* de acceso de *Apache*, mediante la función `extractApacheDate`. Consultar definición en repositorio publicado[5].

En pruebas de ejecución, se someten predicados similares a:

```
f "109.24.239.55 ... "-" "-"" == "13/Nov/2018:17:30:26 -0300"
```

Efectivamente, ante un predicado así, *MagicHaskeller* responde con la función `extractApacheDate`, sin embargo, esta función representa un caso muy específico, ya que se trata de la extracción de la fecha solo para este tipo de *log*. Entonces, nace el cuestionamiento sobre la posibilidad de realizar funciones capaz de abarcar un mayor espectro de casos posibles, teniendo en cuenta la inmensa variedad de estructuras y tipos de *logs* existentes por *frameworks*.

4.4. Propuesta de caracterización de funciones: Genéricas.

Dado que los *DSBK* contienen funciones para la manipulación de un determinado tipo de dato, muchas de ellas pueden resolver problemas relacionados con otros tipos de datos. Además, si los datos pertenecen a un dominio y el problema en cuestión termina siendo una tarea muy exclusiva que pertenece a ese

[5] http://www.github.com/derlissalinas/dsi

dominio, se necesitan funciones más precisas para obtener resultados correctos. En pos de caracterizar las funciones a añadir para un dominio, se contempla la caracterización estableciendo dos límites. Primero, con funciones **genéricas**, basada en dos principios que se adoptan para representar la idea propuesta:

Especificidad. Las funciones generadas deben retornar únicamente los datos esperados, implicando validaciones en los datos de retorno. Por ejemplo, la función `extractApacheDate` de la Sección anterior, extrae la fecha solamente para los *logs* de *Apache*, del tipo Access, sin embargo, carece de una validación para afirmar que lo retornado es un dato que pertenezca al dominio de fechas.

Generalidad. Las funciones deben ser capaces de abarcar un mayor espectro de casos posibles dentro de un mismo dominio, realizando las correspondientes validaciones por medio de condicionales.

Teniendo los casos de *logs* de *Spring* y *Apache*, se generan las funciones de ejemplo basadas en la **especificidad** para la extracción de fechas, denominadas `getDateSpring` y `getDateSpringArray`. De igual forma, se generan las funciones `getDateApache` para Access y Error. Por otra parte, se genera un ejemplo de una función genérica, que se aferra al principio de **generalidad**. La implementación está presente en el *script* `C2_Logs.hs` del repositorio[6].

Mediante condicionales se evalúan todas las funciones específicas hasta determinar la que retorne un valor distinto de vacío.

4.5. Propuesta de caracterización de funciones: Concretas

Es posible generar funciones de concretas, simples y puntuales para un caso en particular. Respecto a la Sección anterior, estas funciones guardan similitud con la simplicidad del principio de *especificidad*, pero sin la validación correspondiente. Tales son los casos de las funciones generadas en la Sección 4.2 para *logs* del *framework Spring*. A continuación se muestra una función como ejemplo para la extracción de fechas.

```
Prelude>extractDate :: [Char] -> [Char]
        extractDate x = unwords (take 2 (splitStringByPunctuation x " "))
```

Estas funciones son generadas netamente para un caso en particular, es decir, obedecen sólo a predicados que posean *logs* de *Spring* como *input*, y por otra parte, en el *output*, lo que explícitamente cada función es capaz de evaluar.

5. Resultados y Discusión

La caracterización de funciones para el agregado de un nuevo dominio no es, en absoluto, determinante, ya que está sujeto a factores intrínsecos de cada dominio que lo diferencia de otros, por lo cual, no es posible decir, por ejemplo, que un conjunto de funciones concretas se corresponde como la mejor solución para todos los dominios existentes que pudiesen añadirse a *DSI*. Al definir un dominio y pretender añadirlo a *DSI*, debe considerarse lo mencionado en la

[6] http://www.github.com/derlissalinas/dsi

Sección 3.3, los parámetros *depth* y *breadth* son determinantes al momento de contemplar la caracterización ya que éstos valores determinan el tamaño de la tabla de memorización.

Depth. El parámetro más relevante dentro de la caracterización determina la cantidad de composiciones posibles entre las funciones del *MagicHaskeller*, lo cual implica un aumento exponencial en la cantidad de combinaciones posibles.

Breadth. Cantidad de funciones que generemos para nuestro dominio.

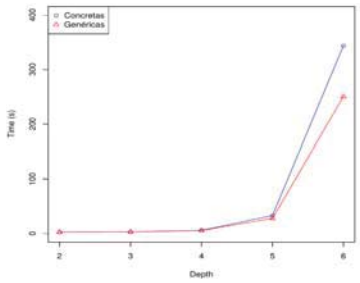
Tomando el caso del dominio de *Logs*, es sabido que existe un enorme número de estructuras diferentes de registros si se considera la cantidad de *frameworks* existentes con la consecuente posibilidad de modificación de dicha estructura básica en una aplicación, pero una vez definida la estructura del *log*, la misma está sujeta a escasas variaciones, por lo cual, es plausible considerar la creación de funciones concretas que operen directamente sobre dicha estructura de registro. Esto implica que la cantidad de funciones a definir es directamente proporcional a la necesidad de datos a obtener de un *log*, como se mencionó en la Sección 4.2, cinco funciones en el *DSBK* bastaron para satisfacer la necesidad del caso.

Por otra parte, considerando un dominio como el de fechas, existe una amplia gama de funciones a definir para el tratamiento de datos, dadas las diferentes variaciones y estándares que existen, aunque así como con el dominio de *Logs*, pueden crearse funciones concretas para manejar cada caso o cada necesidad de obtener un dato a partir de cada *input*. Es sabido que una fecha en formato *"DD/MM/AA"* puede representarse también como *"DD/month/AA"*, donde el mes se define por nombre y no por valor numérico. Ante la necesidad de funciones que abarquen más de un idioma, nace la cuestión en definir funciones de tipo concreta o genérica, ya que un listado de meses de un idioma es definido como una función, no variable, lo cual implica que se deberá definir una función para cada idioma añadido. Esto eleva proporcionalmente la cantidad de funciones disponibles aumentando el valor de *breadth*. Como ejemplos se toman a las funciones `convertMonth`, que convierte el nombre de un mes en su valor numérico y a `getMonthName`, que extrae el nombre de un mes embebido en una cadena de caracteres. Una solución al caso puede verse como la definición de funciones genéricas, que realicen las validaciones contemplando los idiomas, evitando añadir todas las definiciones de funciones dentro del archivo *Individuals.hs* y en las *primitivas* de *MagicHaskeller* reduciendo drásticamente el valor de *breadth*.

Ante una situación que requiera añadir un dominio, es deber analizar la cantidad de funciones a crear, a fin de contemplar mantenerlas como concretas, o bien, si los datos a extraer requieren de varias funciones específicas cuyas funcionalidades sean similares, implementar una función genérica que englobe todas esas funciones; este último es aplicable a las funciones que precisan un nivel de abstracción superior a las funciones concretas, como el caso de la extracción de fechas independientemente al formato de entrada, o como se menciona anteriormente, el caso de extracción del mes sin importar el idioma.

En la Figura 1, se presenta una comparación de tiempo de entrenamiento en segundos para las funciones `convertMonth` y `getMonthName` en sus versiones

concretas y genéricas, considerándose el agregado de veinte idiomas con sus respectivas lista de meses, variando el valor de *depth* en cada ejecución.



Depth	Concretas	Genéricas
2	2,61	2,57
3	2,86	2,81
4	5,78	5,37
5	32,94	27,75
6	343,87	250,56

Figura 1. Tiempos de entrenamiento con funciones genéricas y concretas, sobre *depth*.

Un requisito determinante para la creación de funciones y el posterior agregado de un dominio a *DSI* y por ende al *MagicHaskeller*, es que las funciones deben de estar definidas para todos los valores posibles de entrada, como una función total[7], evitando excepciones. Éste requisito es propio del *MagicHaskeller* para que las funciones sean contempladas en la fase de entrenamiento y agregadas en la *tabla de memorización*. Al definir funciones deben de contemplarse todos los casos posibles de *inputs* y que la función sea capaz de evaluarlo.

La *Programación Funcional Inductiva* en la síntesis de funciones implica criterios para realizar la selección. Dichos criterios son los sesgos inductivos. En al ambiente del *MagicHaskeller*, el *sesgo de preferencia* se manifiesta al momento de ampliar el *DSBK*, agregando los prototipos de las funciones de un determinado dominio al archivo de configuración *Individuals.hs*, ya que el orden en el cual dichos prototipos son colocados, influye para que sean encontradas con menor o mayor facilidad una función. Mismo sesgo se manifiesta mediante el archivo *Primitives*, en donde se indica explícitamente el orden de prioridades para las funciones antes de la ejecución de una nueva instancia del *MagicHaskeller*. Ambos casos se deben tener en cuenta para el agregado de un dominio al ambiente de *DSI*. Cabe resaltar que ha quedado como incógnita el porqué de situaciones donde habiéndose cumplido y verificado las condiciones mencionadas, *MagicHaskeller* no retorna las funciones definidas.

6. Conclusión y Líneas de Trabajo Futuro

La gran cantidad de *Logs* generándose constantemente en los programas informáticos evidencian la necesidad existente de automatizar su procesamiento y análisis a fin de generar una información útil a partir de éstos. Incorporar el dominio de Logs al entorno de *DSI* resulta un proceso poco trivial, en vista a que

[7] http://mathworld.wolfram.com/

se ha detectado como factor determinante a la *genericidad* de las funciones de un dominio. En paralelo, se ha demostrado que *MagicHaskeller* es una herramienta viable para la inferencia de *Logs*. Este aporte podría enmarcar la aceptación de la alta expresividad de los Lenguajes de Programación Declarativos para la solución de necesidades en ambientes poco explorados hasta el momento.

En términos generales, considerando futuros dominios complejos como cadenas de texto o mensajes, implicará caracterizar funciones en cierto grado de *genericidad*. En ello, el *quid* de la incorporación de dominios reside en el balanceo entre casos donde se determine la *genericidad* por función. Basado en lo expuesto, los trabajos futuros podrían centrarse en esclarecer la incógnita respecto al *MagicHaskeller* en los casos donde no retorna funciones previamente definidas. Además, la ampliación de factores para la caracterización, como el análisis de la factibilidad de variación del orden y los valores de prioridades en prototipos, siendo éstos sesgos propios de *MagicHaskeller*.

Referencias

[1] M. A. Constanzo y S. I. Casas. «Usabilidad de framework web: identificación de problemas y propuesta de evaluación». En: *XXIV Congreso Argentino de Ciencias de la Computación (La Plata, 2018)*. 2018.

[2] L. Contreras-Ochando. «General-purpose Declarative Inductive Programming with Domain-Specific Background Knowledge for Data Wrangling Automation». En: (2018). DOI: `https://arxiv.org/abs/1809.10054`. arXiv: `1809.10054`.

[3] S. Gulwani. «Automating String Processing in Spreadsheets Using Input-Output Examples». En: *POPL '11 Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2011), págs. 317-330.

[4] S. Katayama. «Systematic search for lambda expressions». En: *In Proceedings Sixth Symposium on Trends in Functional Programming*. 2005.

[5] S. Katayama. «Towards Human-Level Inductive Functional Programming». En: *Artificial General Intelligence* 10 (2015), págs. 111-120. DOI: `https://doi.org/10.1007/978-3-319-21365-1`.

[6] E. Kitzelmann. «A Combined Analytical and Search-Based Approach to the Inductive Synthesis of Functional Programs.» En: (2010).

[7] Robert Kowalski. «Algorithm= logic+ control». En: *Communications of the ACM* 22.7 (1979), págs. 424-436.

[8] John W Lloyd. «Practical Advtanages of Declarative Programming.» En: *GULP-PRODE (1)*. 1994, págs. 18-30.

[9] M. Mizutani. «Incremental mining of system log format». En: *2013 IEEE International Conference on Services Computing*. IEEE. 2013, págs. 595-602.

[10] Adam Oliner, Archana Ganapathi y Wei Xu. «Advances and challenges in log analysis». En: *Communications of the ACM* 55.2 (2012), págs. 55-61.

[11] R. Olsson. «Inductive functional programming using incremental program transformation.» En: *Artificial Intelligence* 74.1 (1998), págs. 55-81.