Hypersets

Rocío Perez Sbarato

Julio 2025

Índice

Índice de códigos

1. Prototipos de circularidad con *hypersets* en Haskell

Implementé un módulo de *hypersets* en Haskell para modelar sistemas reflexivos y analizar sus propiedades formales, con el objetivo de desarrollar herramientas computacionales para la experimentación filosófica. Usando el lenguaje Haskell conseguí desarrollar un modelo matemáticamente leal al marco teórico construido por Peter Aczel[Acz88]. Ha resultado de gran ayuda el libro *Vicious Circles*[BM96], especialmente el capítulo 10 sobre grafos, y el uso de Haskell en las investigaciones sobre *Herederitary Finite Sets* realizadas — sin publicar —

Figura 1: Esquema general de la implementación del módulo HyperSetGraph

por Paul Tarau[Tar08]. A través de estos prototipos, busco representar *hypersets* tanto como soluciones a sistemas de ecuaciones como en forma de grafos, aplicar decoraciones y analizar propiedades fundamentales como la bisimulación.

Gracias al manejo de principios de diseño de software como modularidad, reusabilidad y corrección, el código resultante[1] posibilita el modelado de *hypersets* mediante el pasaje de la teoría matemática a un código simbólico que la refleje sin perder el rigor técnico. El resultado final permite no solo una comprensión más profunda de la noción de *hyperset* desde una perspectiva computacional, sino también una base funcional de código re-utilizable para modelar sistemas reflexivos. Es por esto que proyecto esta implementación como un futuro paquete de Haskell, disponible en Hackage[2] con su debida documentación.

La organización de la explicación seguirá el esquema de la Figura 1. En la sección Representación de *hypersets* mediante grafos en Haskell se desarrolla la relación entre *hypersets* y grafos, mientras que en la sección Representación de ecuaciones en ***ZFA*** se presenta la implementación de la relación entre sistemas de ecuaciones recursivas en el universo de los conjuntos no bien fundados y las soluciones a estas ecuaciones. Indirectamente se construye el puente entre las ecuaciones y grafos. Cabe destacar que la incorporación de *deep embedings* en los tipos de datos centrales en nuestra estructura fue una decisión de diseño que, a pesar de implicar un proceso complicado y meticuloso, facilitó el manejo sintáctico y el análisis meta-teórico.

[1] Disponible en `https://github.com/rocio-perez-sbarato/strangelooplab`
[2] `https://hackage-content.haskell.org/`

1.1. Representación de *hypersets* mediante grafos en Haskell

1.1.1. Antecedentes de prototipos exploratorios de Teoría de conjuntos

En el borrador de Tarau, se representa la biyección entre conjuntos en ***ZFC*** y números naturales [3]). Para representar los conjuntos se usan el tipo de datos `data HFS t = U t | S [HFS t]` debido a la flexibilidad y expresividad que posibilita crear tu propio tipo, en lugar de usar el paquete `Data.Set` de Haskell. Notar que `data HFS t` permite explicitar la herencia de conjuntos mientras que `Data.Set` solo almacena elementos.

Este tipo de datos de conjuntos hereditarios finitos facilita su expresión en grafos. Para ello, se usa el tipo de datos `Graph` [4] de la librería `Data.Graph`. Es importante aclarar que en tal paper se modelan grafos dirigidos acíclicos (DAGs).

Listing 1: Tipo de datos `Graph`

```
type Vertex = Int
type Table a = Array Vertex a
type Graph = Array Vertex [Vertex]
type Bounds = (Vertex, Vertex)
type Edge = (Vertex, Vertex)
```

Un dato incentivador para la investigación que nos compete es que el mismo Tarau sugiere que `compute_decorations` es similar a la función *decorations* de ***ZFA***.

Listing 2: Versión de Tarau de la función de decorado

```
compute_decoration g v =
compute_decorations g (g!v) where
    compute_decorations _ [] = 0
    compute_decorations g es =
        sum (map ((2^) . (compute_decoration g)) es)
```

La función `compute_decorations` sirve para decorar nodos en un grafo dirigido sin ciclos. `g ! v` accede a la lista de adyacencia del nodo `v`, es decir, devuelve todos los nodos a los que apunta `v`. Es una función recursiva que asigna un número a cada nodo en base a los nodos a los que apunta. Si un nodo `v` no tiene vecinos (lista vacía), su decoración es 0. Si tiene vecinos, para cada vecino `e`, se calcula `compute_decoration g e`, se le aplica `2^x`, y se suman todos esos valores.

[3] Ackermann Encoding: `https://en.m.wikipedia.org/wiki/Ackermann_function`

[4] Practical Graph Handling: `https://wiki.haskell.org/index.php?title=The_Monad.Reader/Issue5/Practical_Graph_Handling`

Este trabajo nos deja preguntas sobre lo cíclico resonando... ¿Cómo podemos adaptar `compute_decorations` a la teoría de *non well founded sets*? ¿Cuáles son las dificultades para adaptar el *Ackermann Encoding* para conjuntos no bien fundados? Profundizaremos en estas cuestiones en ese mismo orden y urgencia. Sin tener la ventaja de biyectividad entre conjuntos HFS y enteros, generaremos un grafo a partir de un HFS y calcularemos el decorado de grafos. Como una discusión teórica fructífera para futuros proyectos, ahondaremos sobre una posible codificación para *hypersets*.

1.1.2. Implementación de `setToLabGraph`

Según la teoría ***ZFA***, cualquier conjunto — ya sea bien fundado o no — puede ser representado como un grafo G con la relación de pertenencia ($\in$) o herencia ($\ni$). Un ejemplo que podemos tomar es $q = \{p, \{s, \{t, u\}\}\}(\dagger)$ y su grafo etiquetado presente en la Figura 2.

Partiendo del trabajo de Tarau, usaremos el tipo `Graph`, que representa un grafo como un par de vértices y aristas, e incorporamos grafos etiquetados con `LabGraph`, que además incluye una función de etiquetado que asigna a cada vértice un conjunto de etiquetas o *strings*.

Listing 3: `Graph` y `LabGraph`

```
-- === Tipos Graph y LabGraph ===
type Graph = Array Vertex [Vertex]
type Edge = (Vertex, Vertex)
type Bounds = (Vertex, Vertex)

type Labeling a = Vertex -> HFS a
data LabGraph n = LabGraph Graph (Labeling n)
```

Para modelar la teoría de *hypersets*, consideramos necesaria la transformación de conjunto a grafo mediante la función `setToLabGraph :: HFS t -> LabGraph`. Su implementación requirió tratar las *labels* que se le asignan a cada elemento y cuáles son las aristas que componen al grafo. Sobre las aristas, tenemos la opción de inventar una etiqueta por elementos de HFS o de recibir del usuario las etiquetas de cada elemento. En mi caso opté por la segunda opción. Entonces, pasamos del tipo de datos `HFS` al nuevo tipo de datos `LabelHFS` con más información incorporada.

```
type Label = String
data LabelHFS t = U (t, Label) | S Label [LabelHFS t]
setToLabGraph :: LabelHFS t -> LabGraph
```

Sobre el dilema de las aristas, debemos tener en cuenta que una arista en los grafos de nuestro interés representa herencia. Es por esto que si los elementos dentro del constructor S son los hijos del constructor, entonces hay una arista que va desde el elemento S hacia sus hijos. Para refrescar nuestra memoria, observemos la Figura 2. Usando nuestra implementación, el conjunto †

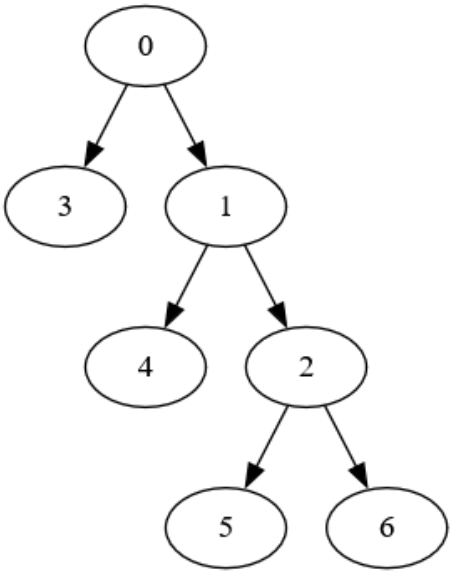

Figura 2: Grafo obtenido con el conjunto $q = \{p, \{s, \{t, u\}\}\}$ como parámetro de entrada de la función `setToLabGraph`

se representa como `setExample = S "q" [U "p", S "stu"[ U "s", S "tu" [U "t", U "u"]]]`.

La cuestión ahora es la siguiente: si buscamos *labels* de los hijos de un conjunto S con *label* x, ¿es esta una forma correcta de obtener las aristas que salen de x? Puesto que dos nodos podrían tener la misma *label*, no es la mejor manera porque generaría la respuesta incorrecta en tales casos. Para identificar unívocamente cada elemento del conjunto, surgió la ideal sinónimo de tipos `ID`.

Listing 4: Tipo de datos de conjuntos hereditarios finitos con `ID` y `Label`

```
type ID = Int
type Label = String
data RefHFS t = RefS Label ID [RefHFS t] | RefU (t, Label, ID)
```

Una ventaja de referenciar cada elemento evidente es que definir referencias y auto-referencias es accesible. En nuestra implementación, representamos una arista entre elementos como que uno de ellos tiene al otro como hijo. En otras palabras, usando la relación de herencia intrínseca al tipo `RefHFS`. En este sentido, para representar auto-referencia de un elemento, definimos a ese elemento como hijo de sí mismo. Por ejemplo, el conjunto Omega que es un único nodo con un *loop* hacia sí mismo es representado como `setExampleCyclic = S "X" 0 [ U ("X",0) ]`.

Ahora bien, analicemos la función `setToLabGraph`. Dado un conjunto con su herencia, *labels* y ciclos definidos explícitamente, se pasa a un grafo. La función

está compuesta primariamente por dos funciones `setToGraph :: RefHFS t -> Graph` y `getLabels :: RefHFS t -> Array ID Label`. Esta última utiliza la lógica de `getChildren :: RefHFS t -> ID -> [ID]`.

El código completo está disponible en el anexo, ya enlazado en la introducción. Sin entrar en detalles y en pos de despejar dudas sobre el funcionamiento de este bloque, a continuación se plantea un ejemplo. Supongamos que ejecutamos `setToLabGraph setExample`. Para generar el grafo, se llama a la función `setToGraph setExample`, allí `getChildren` es aplicada a cada elemento según su `ID`.

```
ghci> setExample = RefS "q" 0 [RefU ("p","p",1),RefS "stu" 2 [RefU
    ("s","s",4),RefS "tu" 3 [RefU ("t","t",5),RefU ("u","u",6)]]]
ghci> setToGraph setExample
ghci> array (0,6)
    [(0,[1,2]),(1,[]),(2,[4,3]),(3,[5,6]),(4,[]),(5,[]),(6,[])]
```

Ahora estos `IDs` son los vértices del grafo. De esta manera, obtenemos un elemento de tipo `Graph` generado con un arreglo de hijos indexados por vértices. Asimismo, se usa `getLabels` para extraer los *labels* de cada elemento, generando la función `Labeling`.

```
ghci> getLabels setExample
ghci> array (0,6)
    [(0,"q"),(1,"p"),(2,"stu"),(3,"tu"),(4,"s"),(5,"t"),(6,"u")]
```

1.1.3. Representación de grafos etiquetados y cíclicos

Para poder simular el decorado de *hypersets*, fue necesario incorporar un chequeo de ciclos. En caso de que haya un ciclo, se corta en la segunda iteración.

Definición 1.1 (Decoración)**.** Sea $A \subseteq \mathcal{U}$. Un *(labeled graph)* $G = \langle G, \to, l \rangle$ sobre A es una 3-upla tal que $\langle G, \to \rangle$ es un grafo, y $l : G \to \mathcal{P}(A)$. Una *decoración* de un grafo G sobre A es una función $d : G \to \mathcal{V}[A]$ tal que para todo $g \in G$

$$\boxed{d(g) \equiv \{d(h) \mid g \to h\} \cup l(g)}$$

Recordemos que siguiendo la definición de Aczel, el decorado de un nodo es el decorado de sus hijos y su propio *label*.

```
    computeDecorations :: LabGraph String e -> Array Vertex (HFS String)
    computeDecorations (LabGraph gr label) = decs
        where
            decs = listArray (bounds gr) [decorate [] v | v <- indices gr]

                decorate visited v
                    | v `elem` visited = U (label v ++ " = {" ++ label v
                        ++ "}") -- circularidad
                    | otherwise =
```

```
                        let children = map snd (gr ! v)
                            childDecs = map (decorate (v : visited))
                                children
                        in S (childDecs ++ [U (label v)])
```

1.1.4. Visualización de grafos mediante Graphviz

Ponemos a prueba nuestra implementación con el ejemplo a la izquierda de la Figura 3. Como `Vertex` es un sinónimo de `Int`, tomamos $a = 0$, $b = 1$, $c = 2$. Esto debería poder cambiarse fácilmente.

La función *label* l es $l(0) = \{x\}$, $l(1) = \emptyset$, $l(2) = \{x, y\}$. Entonces, los decorados de cada nodo son

$$d(0) = \{d(1), d(2), x\}$$
$$d(1) = \{d(1), \emptyset\}$$
$$d(2) = \{x, y\}$$

Construimos el grafo con `buildG`.

```
ghci> graph = buildG (0, 2) ["x", "0", "x,y"] [ (0, 1), (0, 2), (1, 2) ]
ghci> array (0,2) [(0,[2,1]),(1,[2]),(2,[])]
```

Luego, lo usamos para construir el *labeled graph* usando `LabGraph`. La parte derecha de la Figura 3 está generada con el archivo *.dot*, el cual es el *output* generado por la función `showGraphViz` partiendo del *labeled graph* como input[5].

El decorado generado con `computeDecorations` para el ejemplo 1 es

$$0 : \{\{x, y\}, \{\{x, y\}, 0\}, x\}$$
$$1 : \{\{x, y\}, 0\}$$
$$2 : \{x, y\}$$

La única diferencia con la salida esperada es que $d(1)$ tiene 0 en lugar de $\emptyset$. Esto es porque la función `Labeling` toma `Vertex` y devuelve un elemento, en lugar de un conjunto. Esto debería poder cambiarse fácilmente.

1.1.5. Corrección conceptual de tipo `LabGraph` e implementación de `computeDecoration`

Para mejorar la legibilidad del `print` de decoraciones, la segunda versión de `computeDecorations` asigna el conjunto vacío al vértice que ya vio anteriormente, identificando un ciclo. Esas decoraciones ficticias que cortan el ciclo son filtradas a la hora de construir las decoraciones oficiales. Además, el decorado se construye mediante la operación `unionHFS`. Esencialmente, realiza concatenación de listas de conjuntos `HFS` sin repetir elementos.

[5] GraphViz Online: `https://dreampuf.github.io/GraphvizOnline`

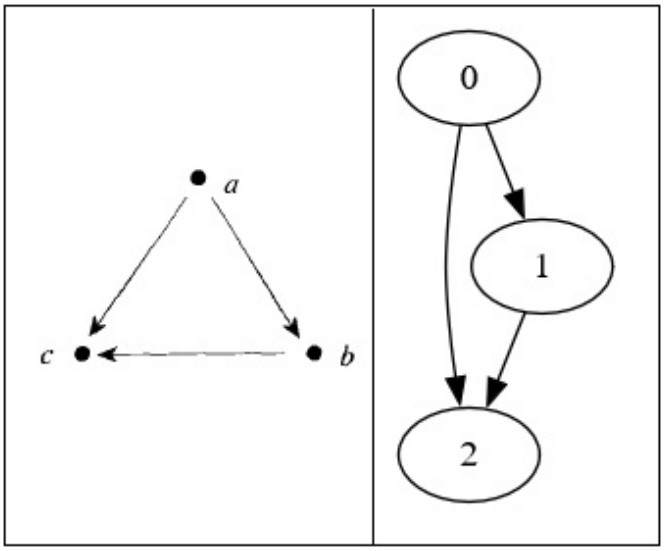

Figura 3: Grafo sacado de *Vicious Circles* y visualización de grafo generado con `setToLabGraph`

Listing 5: `decorate` con operaciones de conjuntos HFS

```
decorate :: Graph -> Labeling String -> [Vertex] -> Vertex -> HFS
    String
decorate gr label visited v
  | v `elem` visited = S [] -- Evitar recursion infinita
  | null children = label v -- Caso base: sin hijos
  | otherwise = unionHFS (label v) (S nonEmptyChildDecs) -- Caso
      recursivo: con hijos
  where
    children = gr ! v
    visited' = v : visited
    nonEmptyChildDecs = filter (not . isEmpty)
                        (map (decorate gr label visited') children)
```

Para que la función `Labeling` de `LabGraph` vaya a la par de la teoría, fue necesario cambiar el tipo a `Labeling a :: Vertex -> HFS a`. A la par de esto, es importante mantener coherencia a lo largo de la implementación. Por ello, se deberán realizar las modificaciones a `Label` como sinónimo de `HFS String`. Sobretodo la importancia surge de grafos con decorados *labels* repetidas, lo cual se soluciona aplicando la unión de conjuntos en lugar de concatenación de *strings*.

1.2. Representación de ecuaciones en *ZFA*

Modificamos el tipo `HFS` para que se pueda identificar un conjunto y sus elementos con un *ID*, y a su vez las referencias a un conjunto (mediante su *ID*). Por eso, `RefHFS` es muy similar a una estructura de grafo.

Ahora que ya quedó claro por qué usamos `RefHFS` y de qué modo lo hacemos, es momento de pasar a la implementación del bloque de las ecuaciones. La idea es implementar la parte de la teoría que refiere al *Solution Lemma*, que establece que en ***ZFA*** todo sistema de ecuaciones tiene una única solución.

Estas ecuaciones están compuestas por conjuntos, variables y constantes. Las soluciones son los valores que encajan en la disposición del sistema. En el tipo `SetExpr`, usamos `Ref` para referirnos a otra variable, posibilitando ciclos y referencias; usamos `Expr` para referirnos a una constante; y usamos `SetOf` para iniciar un conjunto. El tipo `Equation` refleja la ecuación para una variable en particular. La idea es usar `Variable` como las *labels* de la variable, aunque a futuro sería útil incorporar la creación de la función `Labeling` para cada variable y luego reutilizarla, si se desea realizar la transformación a grafo etiquetado.

Listing 6: Tipo `System` compuesto por `Equation` y `SetExpr`

```
1  -- === Tipo System ===
2  type Variable = String
3  data SetExpr t = Ref Variable | Expr t | SetOf [SetExpr t]
4    deriving (Show)
5  data Equation t = Equation Variable (SetExpr t)
6    deriving (Show)
7  type System t = [Equation t]
```

1.2.1. Implementación de `denoteSystem`

Para representar el *Solution Lemma*, usamos la función `denoteSystem`. Las soluciones a las ecuaciones son representadas simbólicamente y no se calculan de forma explícita, muchas veces porque trabajamos solo con variables sin valor asignado. Por eso, este trabajo es de carácter formal. Para referirnos a los *hypersets* de manera unívoca y sintáctica, utilizamos el sinónimo de tipos `System`. Cada *hyperset* se deriva – o es una solución – de un sistema de ecuaciones.

En el tipo `SetExpr`, las referencias a otras variables dentro del sistema se representan con el constructor `Ref`, que durante la denotación se transforma en un `RefU`, es decir, una referencia unificada con identificador y número de nodo. Esto permite representar conjuntos no bien fundados con ciclos y referencias. Para manejar correctamente los ciclos, se utilizan listas auxiliares `visited` y `expanded` dentro del algoritmo de denotación. Estas estructuras evitan que se expanda o se visite el mismo subgrafo más de una vez, lo cual sería incorrecto tanto desde el punto de vista computacional como teórico.

Una regla a respetar para el correcto funcionamiento de `denoteSystem` es el orden escalonado y hereditario de la lista de ecuaciones. Esto es porque se

construye el conjunto hereditario finito a medida que se desglosa el sistema de ecuaciones. Siguiendo el ejemplo del conjunto †, mostramos a continuación el comportamiento de las ecuaciones

```
ghci> systemExample =
  [ Equation "q" (SetOf [Expr "p", Ref "stu"])
  , Equation "stu" (SetOf [Expr "s", Ref "tu"])
  , Equation "tu" (SetOf [Expr "t", Expr "u"])
  ]
ghci> denoteSystem systemExample "q" -- Seleccionamos la variable raiz
ghci> RefS "q" 0 [RefU ("p","p",1), RefS "stu" 2 [RefU ("s","s",4), RefS
    "tu" 3 [RefU ("t","t",5), RefU ("u","u",6)]]]
```

Referencias

[Acz88] P. Aczel. *Non-well-founded sets.* CLSI Publications, Estados Unidos, 1988.

[BM96] J. Barwise and L. Moss. *Vicious Circles: On the Mathemathics of Non-Wellfounded Phenomena.* CLSI Publications, Estados Unidos, 1996.

[Tar08] P. Tarau. *A Functional Hitchhiker's Guide to Hereditarily Finite Sets, Ackermann Encodings and Pairing Functions.* University of North Texas. [Unpublished draft], Estados Unidos, 2008.