Notas del Trabajo Especial

Rocío Perez Sbarato

Abril 2025

Observación

Este documento busca ser un borrador del trabajo especial. Su objetivo es cristalizar y ordenar el flujo de ideas durante el proceso de investigación.

Índice

Índice de figuras

Índice de códigos

1. Grafos en la teoría de *hypersets*

Antes de comenzar con la aventura de estudiar las estructuras de sistemas reflexivos, es importante apropiarse de ciertos conceptos de la teoría sobre la cual se va a basar fuertemente nuestro trabajo. En particular, los modelos y prototipos de paradojas serán realizados con el capítulo 10 del libro *Vicious Circles* de Barwise y Moss.

1.1. Teoría de conjuntos no bien fundada (*ZFA*)

La teoría ***ZFC***, que es el estándar en matemática, incluye el axioma de fundación (FA), el cual impide la existencia de conjuntos que se contengan a sí mismos directa o indirectamente. Es decir, no permite circularidades en la relación de pertenencia. Este axioma garantiza una estructura jerárquica bien fundada de los conjuntos, pero excluye muchos modelos interesantes, como aquellos que permiten la autorreferencia.

La teoría ***ZFA*** (Zermelo-Fraenkel con Anti-Fundación), en cambio, reemplaza el axioma de fundación por el **AFA** (Axiom of Anti-Foundation), lo que permite conjuntos no bien fundados, es decir, conjuntos que pueden contenerse a sí mismos (de forma directa o a través de ciclos). Esta teoría es una extensión conservadora de ZFC: para demostrar su consistencia, se asume que ZFC también lo es.

Gracias a AFA, en ZFA es posible plantear ecuaciones de conjuntos que son *reflexivas*, como por ejemplo:

$$X = X$$

1.2. *Solution Lemma*

El *Solution Lemma* es una de las piedras angulares de la teoría ZFA. A grandes rasgos, garantiza que todo sistema de ecuaciones de conjuntos (incluso los que son auto-referenciales) tiene una única solución en el universo de los *hypersets*. Aunque se puede indagar mucho más en los matices de este lemma, por ahora solo debemos tener en cuenta la siguiente definición.

Definición

Todo sistema de ecuaciones en una colección de variables indeterminadas X, dentro del universo V_A (el universo de los conjuntos no bien fundados según AFA), tiene una única solución.

Una *solución* se entiende como una función que asigna a cada variable $x \in X$ un conjunto (es decir, un *hyperset*), de modo que todas las ecuaciones del sistema se satisfacen.

1.2.1. Grafos

Un grafo es simplemente un conjunto G de objetos (es decir, conjuntos o *urelementos*) llamados nodos, junto con una relación binaria $E \subseteq G \times G$. Si $(a, b) \in E$, decimos que hay una arista de a a b.

Además, como E suele entenderse por contexto, solemos escribir $a \to_G b$ (o incluso simplemente $a \to b$) en lugar de aEb. Formalmente, nuestros grafos serán pares ordenados de la forma $\langle G, \to_G \rangle$.

La *decoración* de un nodo se define como el conjunto de las decoraciones de sus hijos. En *Vicious Circles*, la decoración de los grafos significa la relación inversa de $\in$ entre los nodos.

Definición

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 $g \in G$

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

Esta definición recursiva permite representar estructuras como:

- **Ciclos auto-referenciales**, por ejemplo:

  $$q = \langle E, q, 0 \rangle \text{ equivale a } q = \{\{E\}, \{q, 0\}\}$$

  Este grafo puede interpretarse como una formalización de la *paradoja del mentiroso*.

Este principio de correspondencia fue una de las motivaciones iniciales del enfoque de Aczel: representar conjuntos mediante grafos dirigidos con una semántica bien definida (gracias a AFA y al *Solution Lemma*).

El *Solution Lemma* aplicado a grafos, dice que:

Definición

Todo grafo dirigido finito etiquetado (*labeled graph*) tiene una única *decoración*, es decir, una asignación de *hypersets* a cada nodo que respeta la estructura del grafo.

Una de las claves más intuitivas del enfoque de Aczel es que los sistemas de ecuaciones se corresponden con grafos dirigidos etiquetados. En este marco, cada nodo del grafo representa una variable, y sus conexiones representan la estructura de pertenencia ($\in$).

1.2.2. *Hypersets*

En la teoría ZFA, un conjunto (o *hyperset*) puede visualizarse como cualquier estructura que pueda representarse mediante un grafo cuya relación de pertenencia esté definida por sus aristas. Por ejemplo, el conjunto

$$X = \{X\}$$

tiene como representación un nodo con una flecha que apunta hacia sí mismo.

En la teoría de Aczel, cada grafo representa un único conjunto. Aunque distintos grafos pueden representar el mismo conjunto. De hecho, hay un criterio formal para establecer cuándo dos grafos representan el mismo conjunto: la bisimulación. Dos nodos (o grafos) son bisimilares si tienen estructuras equivalentes en términos de relaciones de pertenencia, más allá de cómo estén construidos. La bisimulación, entonces, permite definir una noción de igualdad de *hypersets* que respeta la semántica del modelo.

2. Conjuntos no bien fundados y diagramas de estructuras reflexivas

Uno de los objetivos centrales del trabajo es caracterizar los sistemas formales reflexivos y señalar algunas propiedades recurrentes. Para ello, es posible proponer una representación formal aprovechando las posibilidades que nos brinda la teoría de conjuntos ***ZFA*** para representar circularidad. Es aún más interesante establecer una relación con los análisis formales de las estructuras detrás de muchas paradojas, realizados por Grim y Rescher, plasmados en un tipo de diagramas especial para la ocasión.

2.1. La paradoja del Mentiroso

2.1.1. Diagrama estructural

En *Reflexivity: From Paradox to Consciousness*, Grim y Rescher se proponen mostrar que las distintas paradojas tienen una forma o estructura muy parecida. Este eje central que subyace al absurdo de las paradojas puede ser expresado con un tipo de diagrama que podemos ver en la Figura 2. En esta figura, tenemos la separación de una oración entre sujeto y predicado, lo cual nos permite construir la oración que se refiere a sí misma. Del lado izquierdo se encuentra el sujeto y del lado derecho el predicado. La flecha de estos diagramas representa que el predicado se aplica al sujeto. El predicado debe ser algo que pueda decirse de un sujeto. En nuestro caso, el sujeto se llama **S** y debe referirse a sí misma, es decir, tenerse a sí misma como sujeto (Figura 1).

Dicho esto, podemos notar que la paradoja semántica que se ve expresada en esa circularidad no es cualquier oración auto-referencial sino la expresión de la paradoja del Mentiroso: “esta oración es falsa”. Lo paradójico de esta oración se puede ver en la oscilación de la Figura 2, donde si S - que representa

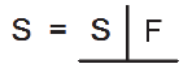

Figura 1: Expresión **S**

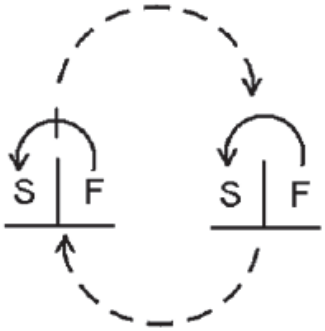

Figura 2: Diagrama de la estructura de la paradoja del Mentiroso.

"esta oración es falsa es falsa, entonces no puede ser que sea falsa. Por otro lado, si no es falsa, entonces tiene que serlo. Resumiendo, la paradoja contiene auto-referencia y negación.

Similarmente, la paradoja de Russell contiene un ciclo. Sea R es el conjunto russelliano compuesto por cualquier conjunto que no pertenezca a sí mismo. Entonces, si R no pertenece a sí mismo, entonces debe pertenecer a sí mismo. En la Figura 8, la flecha representa la relación de pertenencia. A diferencia de la paradoja del Mentiroso, la de Russell solo trabaja con componentes con una estructura única y no de dos partes.

2.1.2. Proposición, ecuación y su *labeled graph* asociado

La Teoría de Conjuntos clásica de Zermelo-Fraenkel con el Axioma de Elección (**ZFC**) no permite los conjuntos que se tienen a sí mismos como elementos. Esto se debe al Axioma de Fundación (**FA**), el cual inherentemente frena la circularidad. En cambio, la Teoría de Conjuntos **ZFC** con el Axioma de Anti-Fundación (**AFA**) sí da lugar a la reflexividad en los conjuntos. Es una extensión de ZFC, tanto que para demostrar su consistencia se parte de la base de que **ZFC** lo es. [1]

Ahora bien, uno de los conceptos clave desarrollados por Barwise y Moss es el *Solution Lemma*. Tiene distintas aristas y colores, en general este lemma significa que cada sistema de ecuaciones tiene una única solución. Esto solo vale en la Teoría de Conjuntos ***ZFA*** (con el Axioma de Anti-Fundación), no en la

[1] Demostrado en los libros de Barwise

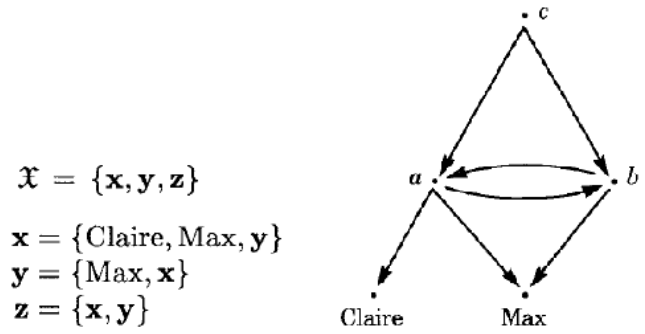

Figura 3: Relación entre decorado y soluciones.

teoría ***ZFC*** que conocemos. En respecto a los grafos, plantea que cada *labeled graph* tiene un único decorado. Podemos pensar que un conjunto es cualquier colección de objetos cuya relación de pertenencia puede ser ilustrada a través de un grafo. En esta Teoría, puede pasar que un conjunto tenga varios grafos que lo representan, pero en el fondo todos deben tener el mismo decorado.

A su vez, los grafos tienen sus nodos y sus aristas. Se trabaja con *labeled graphs* y se define el decorado de estos grafos como el conjunto de los hijos de cada nodo. Si no tiene hijos, entonces ese nodo se decora con su *label*. Luego, los nodos de los grafos son conjuntos y las aristas son establecidas por la relación de pertenencia invertida entre conjuntos. En la Teoría de Conjuntos con **AFA** incorporado, estos grafos pueden ser tanto bien fundados como no serlo. Por ejemplo, en la Figura 3 se ve un grafo no bien fundado.

Otra cosa importante a destacar es que las soluciones de sistemas de ecuaciones y las decoraciones de los grafos se corresponden. En *Vicious Circles* está demostrada la equivalencia entre el *Solution Lemma* en general y para grafos. Teniendo en cuenta estas dos ideas, es claro que las soluciones a las ecuaciones son un mapeo que asigna valores a las variables indeterminadas. En la Figura 3, esas variables indeterminadas son $X = \{x, y, z\}$. Para entender mejor este concepto ir a la sección Definiciones útiles.

La solución o las soluciones de cada ecuación del sistema se encuentran en el decorado del grafo asociado. Por ejemplo, la Figura X. muestra un grafo G_p asociado a la proposición $p = \langle E, q, 0\rangle$, la cual nos dice que la proposición **q** no tiene la propiedad E. La solución a la ecuación $p = \langle E, q, 0\rangle$ es el grafo asociado a G_p. Recordar que, en Teoría de Conjuntos, un par ordenado $\langle a, b, c\rangle$ es expresado como $\langle a, \langle b, c\rangle\rangle$, que es comúnmente escrito como {{a}, {b, c}}.

Esta estructura matemática enmarcada en ***ZFA*** permite la circularidad en estos grafos, como se ve en las Figuras 5 y 4. En esta última, la solución de

la ecuación es ella misma, lo que se puede ver en el *loop* a su propia raíz. Notar que en este caso tomamos la proposición p y la cambiamos para obtener la auto-referencialidad $q = \langle E, q, 0\rangle$. Esta no es necesariamente la paradoja del Mentiroso, pero sin lugar a dudas tiene su estructura. Si $E =$ "esta oración es verdadera", entonces q representa "esta oración no tiene la propiedad "es verdadera. O sea, la oración q es "esta oración es falsa".

2.1.3. Relación entre diagrama y ecuación

Sea S la sentencia con sujeto y predicado que vimos en la **sección 1.1.1** y q la proposición que vimos en 1.1.2. Su equivalencia surge de la reflexividad, negación y la propiedad/predicado que se aplica a la misma estructura auto-referencial. presente en ambas. Además, las propiedades $E =$ "es verdaderoƶ el predicado $F =$ "es falso"son opuestas. Es decir, $E = \neg F$ o $F = \neg E$. Entonces, "esta oración es falsa"no cumple E si y solo si cumple F. Esto es equivalente a decir que S cumple F si y solo si S cumple $\neg E$, q cumple F si y solo si q cumple $\neg E$ y por lo tanto S es equivalente a q en tanto ambas representan el mismo estado de verdad respecto de la evaluación de E. O sea, S es falsa si y solo si q no es verdadera. Cabe aclarar que esta relación entre P y E existe porque ambas son lo que se dice de la unidad auto-referencial, independientemente de si esta propiedad se aplica o no se aplica a tal unidad.

Algo similar podríamos decir para la generalización del diagrama de la paradoja del mentiroso (Figura 7) y otros valores para la propiedad E. Es crucial estudiar esto aplicado a la paradoja de Russell, paradojas semánticas como heterological o teoremas como Halting problem y el de Incompletitud. En el caso de estos dos últimos deberá tener otro enfoque, puesto que son teoremas y el absurdo surge de una construcción de prueba. Sin embargo, se mantiene la estructura de "sujeto y predicado"del Mentiroso.

2.2. Paradoja de Russell

Podemos representar la paradoja de Russell mediante una ecuación auto-referencial en nuestro marco de ecuaciones del tipo $q = \langle E, q, 0\rangle$. En este caso, definimos la expresión $E(x)$ como

$$E(x) = x \notin x$$

y construimos el par ordenado $q = \langle E, q, 0\rangle$. Este objeto representa "el conjunto de todos los conjuntos que no se contienen a sí mismos no se contiene a sí mismo". Evaluar si $q \in q$ equivale a evaluar la condición $E(q)$, es decir $q \in q \iff q \notin q$ lo cual genera una contradicción. De este modo, reproducimos en nuestro sistema la estructura lógica de la paradoja de Russell.

El diagrama de Russell propuesto por Grim y Rescher (Figura 8) presenta la relación entre conjuntos como una sola parte. La flecha indica si el de la izquierda pertenece al de la derecha. Siguiendo este diagrama, notamos que es posible modelar cualquiera de las dos expresiones que generan el ciclo extraño. Esto es posible al relacionar q con R y la fecha tachada con $\neg E$. Aunque Grim

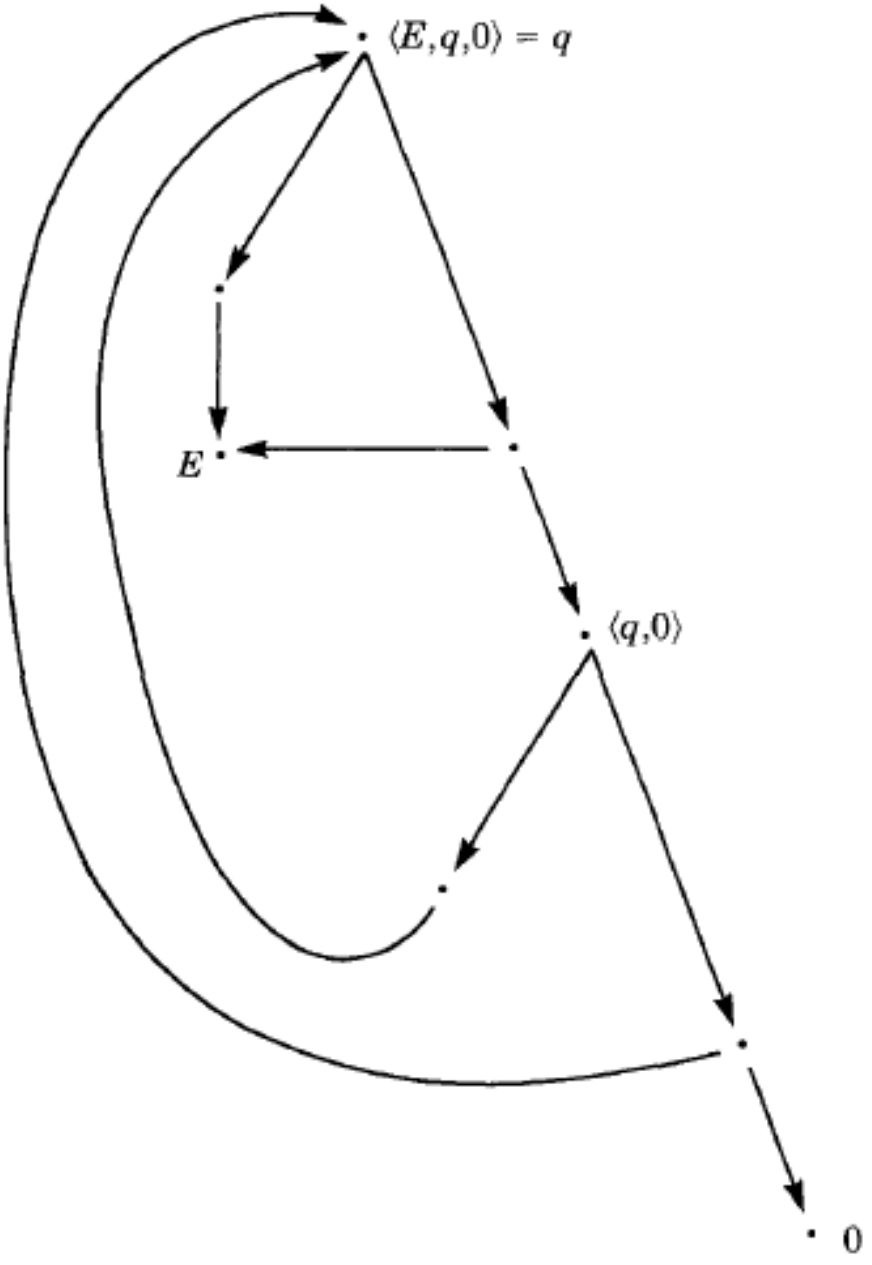

Figura 4: Grafo de $q = \langle E, q, 0\rangle$.

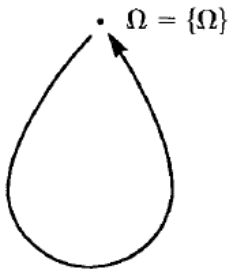

Figura 5: Grafo de la ecuación $\Omega = \{\Omega\}$.

Figura 6: Diagrama del ciclo extraño presente en la paradoja del Mentiroso que se relaciona con el *hyperset q*

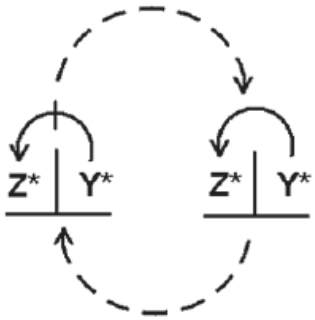

Figura 7: Diagrama de la estructura de la paradoja del Mentiroso y demás paradojas con una estructura de dos partes.

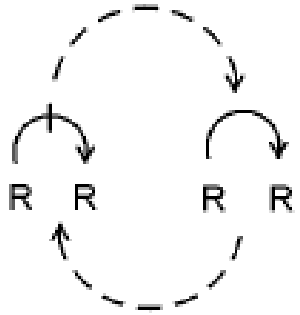

Figura 8: Diagrama de la estructura de la paradoja de Russell

y Rescher no lo propongan así, una aproximación al análisis de la paradoja de Russell mediante la estructura de dos partes con sujeto y predicado consiste en lo siguiente. Sea el conjunto ruselliano R el sujeto y la expresión P el predicado. La propiedad E indica "no pertenecer a sí mismo", que es negación de la propiedad $P(x) = x \in x$. Es decir, $E = \neg P$ y, por lo tanto, $P = \neg E$. Si $q \in q$, entonces no cumple E, y por definición de E, eso implica que $q \notin q$. Por el contrario, si $q \notin q$, entonces cumple E, lo cual implica que $q \in q$. Por lo tanto, R y q son equivalentes en tanto ambas representan una estructura auto-referencial basada en la aplicación de una propiedad a sí misma, y ambas conducen a una contradicción lógica al evaluarse.

2.3. Paradojas de Grim y Rescher

Podemos generalizar las aproximaciones en el modelo de paradojas como la del Mentirosos y la de Russell. Tomamos el análisis de los diagramas (Figuras 7, 9) y el grafo auto-referencial (Figura 4). También, tomamos las relaciones que hemos establecido entre ellas. Esperamos que estas aproximaciones puedan servir como una guía para seguir intentando comprender las estructuras paradójicas. Consideramos que esto es útil para abordar la circularidad y aprovechar las posibilidades que brinda.

Antes de comenzar, haremos notar algunas preocupaciones entorno a las hipótesis que hemos presentado.

Una diferencia entre los diagramas y los grafos es que la Figura 4 representa la oración que causa una paradoja, mientras que diagramas como el de la Figura 2 presentan el "ciclo extraño"de la paradoja, es decir, la dinámica de auto-referencia que genera la contradicción. Es por eso que podemos establecer la relación anterior, donde la propiedad E y el predicado P son opuestos: $E(x) = \neg P(x)$. En el caso del Mentiroso, E puede ser "es verdaderaz P = "es falsa", mientras que en la paradoja de Russell, $E(x) = x \notin x$ y $P(x) = x \in x$. En ambos casos, la paradoja emerge cuando una entidad se evalúa respecto de una propiedad que la niega al cumplirse, generando así una estructura lógicamente

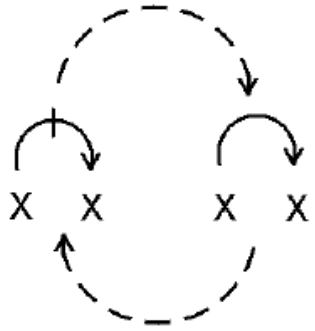

Figura 9: Diagrama de la estructura de las paradojas de Russell, del Barbero, de la palabra hetereological y demás

inestable.

Surgen las interrogantes de cómo lograr que el diagrama solo represente la oración y cómo lograr que el grafo represente el ciclo extraño. La respuesta a la primera pregunta es simple, pues S es esa misma representación. La respuesta a la segunda pregunta parece ser más complicada a la hora de formalizarla, el hecho de que $q = \langle E, q, 0\rangle$ y $q = \langle E, q, 1\rangle$ tienen el mismo grafo podría generar un problema. Sin embargo, podemos justificar, mediante una explicación lógica similar a Grim y Rescher, que $q = \langle E, q, 0\rangle$ y $q = \langle E, q, 1\rangle$ generan un "ciclo extraño".

Cabe señalar que Barwise y Moss trabajan la paradoja del Mentiroso mediante un modelo presente en una teoría de verdad Austiniana. Hemos decidido omitir este enfoque y esperamos poder incorporarlo una vez que comprendamos el rol de los *hypersets* en el *encoding* de las paradojas.

2.3.1. Formalización de relación entre diagramas y *hypersets*

A continuación, se encuentra una observación que actúa a modo de resumen de las secciones anteriores.

Observación

Semantic paradoxes
Y^*:= "este sujeto cumple la propiedad Y^*" $\equiv \neg E$
Z^*:= "este sujeto no cumple la propiedad Y^*" $\equiv q$
- La suposición de un predicado que se relaciona con R todas y solo aquellas oraciones que tienen un predicado que no se relaciona con R con su sujeto, conduce a la oscilación.
- Lleva a un ciclo infinito en el caso de la paradoja del mentiroso.
Set-therorical paradoxes
X:= conjunto de elementos que no cumplen la propiedad Y^* $\equiv$ conjunto de elementos que cumplen $\neg E$
$X \to X$:= "este conjunto X cumple la propiedad Y^*" $\equiv q$
- Nada tiene relación R con todo y solo con aquellas cosas que no tienen relación R consigo mismas.
- Lleva a un ciclo infinito en el caso de la paradoja de Russell, del barbero y de la palabra *hetereological*.

2.4. Prototipos de sistemas reflexivos

En esta sección exploramos distintos enfoques para modelar sistemas reflexivos. En particular, nos enfocamos en desarrollar representaciones formales de la Paradoja de Russell, la Paradoja del Mentiroso y otras construcciones con estructuras similares, con el objetivo de proponer un *template* general que permita capturar este tipo de circularidad de manera sistemática.

Para este propósito, utilizamos el lenguaje funcional Haskell como herramienta principal de modelado. Haskell, con su fuerte soporte para estructuras recursivas y su orientación a la definición declarativa de datos, resulta especialmente adecuado para expresar la circularidad inherente a estas paradojas.

2.4.1. Paradoja de Russell

En un principio, buscamos modelar el diagrama de la Figura 10. También es posible el camino de modelar el par ordenado $q = \langle E, q, 0 \rangle$ mediante *hypersets*. Aunque no son excluyentes, dividimos el desarrollo del primer enfoque en las secciones 2.4.2, 2.4.3 y del segundo en la sección 2.4.4. Como se verá a continuación, el primer enfoque tiene mayor nivel de abstracción y de verbosidad mientras que el segundo es más directo y convincente. Es por eso que propongo que el primer enfoque tiene potencial más alto de escalabilidad.

2.4.2. Intento con bug

Es importante destacar que existe un bug en Haskell[2], donde GHC tiene el error `ghc: panic! (the 'impossible' happened)` al ejecutar el siguiente programa:

[2]Código fuente: `https://okmij.org/ftp/Haskell/impredicativity-bites.html`

Listing 1: Modelo en Haskell de la Paradoja de Russell

```haskell
{-# LANGUAGE GADTs, KindSignatures, EmptyDataDecls #-}

data False -- Fantasma
data J c = J (c ())

{- Si el conjunto no pertence a si mismo, entonces pertenece al conjunto
    russelliano R -}
data R :: * -> * where
   MkR :: (c (J c) -> False) -> R (J c)

{- La funcion f toma como argumento el mismo R (J R) que construye -}
condFalse :: R (J R) -> False
condFalse x@(MkR f) = f x

absurd :: False
absurd = condFalse (MkR cond_false) -- Ciclo de self-reference

main = do
    print (absurd`seq` ())
```

En este caso, `f` es el predicado $P(x) := x \notin x$. Mientras que P construye el conjunto R, el predicado se aplica a este mismo conjunto.

```
condFalse (MkR condFalse)
    condFalse (MkR f)
        == f (MkR f)
    == condFalse (MkR condFalse)
```

La condición `condFalse` es similar a preguntar "¿El conjunto R pertenece a sí mismo?". Más formalmente, se trata de evaluar la propiedad P aplicada a R, es decir, determinar si $P(R)$ es verdadera o falsa, o bien, si $R \in \{x : x \notin x\}$. En el ejemplo, se asigna el valor de falsedad a esta pregunta, lo cual como ya vimos lleva a un ciclo infinito.

Es posible establecer la relación con este predicado `condFalse` y $q = \langle E, q, 0 \rangle$. Ambos proponen que el conjunto ruselliano no pertenece a sí mismo, si tomamos la definición de E que dimos en **Proposición, ecuación y su labeled graph asociado** (sección 2.1.2). Notar que P es exactamente este E, q es exactamente `condFalse`. Por esto mismo, podemos decir que es similar a la Figura 10.

2.4.3. Modelos de autoreferencialidad en Haskell sin bugs

Este programa[3] es muy similar al anterior, lo que cambia es el uso de `inline`. Aunque no termina de correr, puesto que genera un ciclo infinito, permite modelar la circularidad de la paradoja sin bugs.

[3] Código fuente en uno de los comentarios del post en `https://www.reddit.com/r/haskell/comments/5nzxf6/is_the_following_encoding_of_russels_paradox/`

Figura 10: Diagrama de $R \notin R$

Listing 2: Otra versión del modelo en Haskell de la Paradoja de Russell

```haskell
data False

-- Conjunto ruselliano
data R = MkR {proj :: R -> False}

-- R pertenece a si mismo?
f :: R -> False
f = \x -> proj x x
-- La clave: evitar optimizaciones de GHC
{-# noinline f #-}

omega :: False
omega = f (MkR f) -- Ciclo de self-reference

main = do
  print (omega `seq` ())
```

En el código `data False` define un tipo vacío, sin constructores. Representa una proposición lógicamente falsa. El tipo `R` representa un conjunto russelliano, es decir, un conjunto que contiene elementos que no se contienen a sí mismos. Cada valor de tipo `R` tiene una función llamada `proj` de tipo `R ->False`.

El selector de campo `proj` se genera automáticamente por Haskell, y tiene tipo `proj :: R ->(R ->False)`. Así, `proj x` es una función que toma otro `R` y produce un `False`.

Por otro lado, la función `f` toma un valor `x :: R` y aplica su propio `proj` a sí mismo: `proj x x`. Es decir, pregunta si `x` "se contiene a sí mismo", y se aplica a sí mismo para decidirlo. Esta forma es directamente análoga a la construcción del conjunto de todos los conjuntos que no se contienen a sí mismos pero tiene el componente de la auto-referencia.

El pragma {`-# noinline f #-`} es crucial: le indica al compilador que no optimice ni expanda la definición de `f`, para evitar que el compilador descubra anticipadamente el ciclo infinito. Esto fuerza a que la auto-referencia se mantenga en tiempo de ejecución. Notar que el código anterior se arregla usando la misma estrategia.

Listing 3: Arreglo del primer intento de modelo en Haskell de la Paradoja de Russell

```haskell
condFalse :: R (J R) -> False
```

```
2   condFalse x@(MkR f) = f x
3   {-# noinline condFalse #-}
```

La expresión `omega = f (MkR f)` construye una instancia de `R` con la función `f`, y luego se aplica a sí misma. El resultado es un ciclo de auto-aplicación que nunca termina...

$$\begin{aligned}\omega &= \texttt{f (MkR f)} \\ &= \texttt{proj (MkR f) (MkR f)} \\ &= \texttt{f (MkR f)} \\ &= \ldots\end{aligned}$$

Finalmente, en `main`, se intenta forzar la evaluación de `omega` usando `seq` para imprimir `()`. Sin embargo, como `omega` nunca termina, el programa entra en un ciclo infinito ($\perp$).

2.4.4. Intento con `data HFS`

En pos de simular la estructura de los *hypersets*, utilizamos una estructura[4] que nos permite simular la relación de pertenencia entre nodos. En cierto modo, estamos llevando los análisis de Grim y Rescher al mundo de los *non well founded sets*, ya que permitimos que haya circularidad para poder representar la paradoja de Russell.

Listing 4: Modelo de la paradoja de Russell en Haskell usando `data HFS`

```haskell
-- Tipo de conjunto hereditarimente finito con soporte para ciclos
data HFS t = U t | S [HFS t] deriving (Eq, Show)

-- Funcion para saber si un conjunto esta en otro
elemHFS :: Eq t => HFS t -> HFS t -> Bool
elemHFS x (S xs) = x `elem` xs
elemHFS _ _    = False

-- Creamos un elemento para el conjunto R
x :: HFS String
x = S [] -- No se contiene a si mismo

-- Proponemos que R no se contiene a si mismo
r :: HFS String
r = S [ y | y <- [x, r], not (y `elemHFS` y) ] -- Circularidad

main :: IO ()
main = do
    print ((r `elemHFS` r) `seq` ()) -- Ciclo de self-reference
```

[4] Idea sacada de `https://arxiv.org/pdf/0808.0754`

3. 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 conceptualmente 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 — 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[5] 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[6] con su debida documentación.

La organización de la explicación seguirá el esquema de la Figura 11. 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.

3.1. Representación de *hypersets* mediante grafos en Haskell

3.1.1. Antecedentes de prototipos exploratorios de Teoría de conjuntos

En el borrador de Tarau, se representa la biyección entre conjuntos en ***ZFC*** y números naturales [7]). Para representar los conjuntos se usan el tipo de datos

[5] Disponible en `https://github.com/rocio-perez-sbarato/strangelooplab`

[6] `https://hackage-content.haskell.org/`

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

Figura 11: Esquema general de la implementación del módulo HyperSetGraph

`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` [8] de la librería `Data.Graph`. Es importante aclarar que en tal paper se modelan grafos dirigidos acíclicos (DAGs).

Listing 5: Tipo de datos `Graph`

```
1  type Vertex = Int
2  type Table a = Array Vertex a
3  type Graph = Array Vertex [Vertex]
4  type Bounds = (Vertex, Vertex)
5  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 6: Versión de Tarau de la función de decorado

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

```haskell
    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.

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*.

3.1.2. 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 12.

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 7: `Graph` y `LabGraph`

```haskell
-- === 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

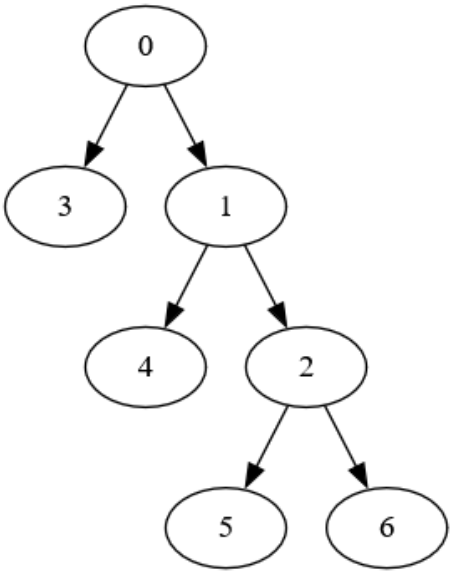

Figura 12: Grafo obtenido con el conjunto $q = \{p, \{s, \{t, u\}\}\}$ como parámetro de entrada de la función `setToLabGraph`

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 12. Usando nuestra implementación, el conjunto † 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 8: 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")]
```

3.1.3. 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

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)])
```

3.1.4. Visualización de grafos mediante Graphviz

Ponemos a prueba nuestra implementación con el ejemplo a la izquierda de la Figura 13. 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 13 está generada con el archivo *.dot*, el cual es el *output* generado por la función `showGraphViz` partiendo del *labeled graph* como input[9].

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

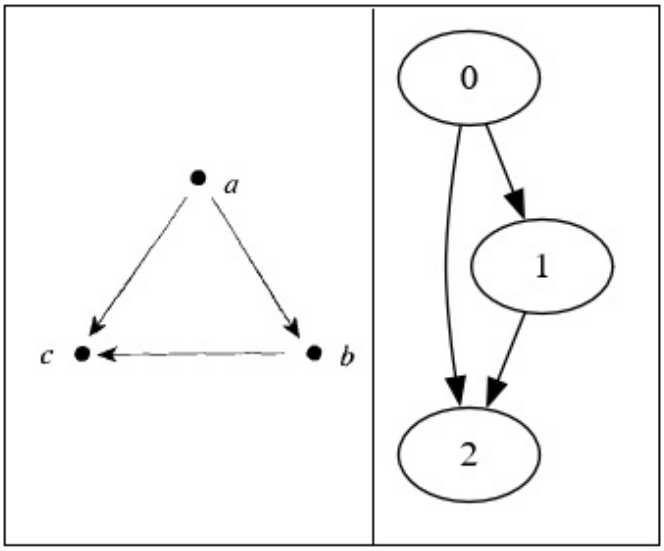

Figura 13: Grafo sacado de *Vicious Circles* y visualización de grafo generado con `setToLabGraph`

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.

3.1.5. 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.

Listing 9: `decorate` con operaciones de conjuntos HFS

```
1   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*.

3.2. 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 10: Tipo `System` compuesto por `Equation` y `SetExpr`

```
-- === Tipo System ===
type Variable = String
data SetExpr t = Ref Variable | Expr t | SetOf [SetExpr t]
  deriving (Show)
data Equation t = Equation Variable (SetExpr t)
  deriving (Show)
type System t = [Equation t]
```

3.2.1. 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)]]]
```

4. Definiciones útiles

4.1. Solución

- The Liar, página 50.

> By an *assignment* for $\mathbb{X}$ in V_A we mean a function $f : \mathbb{X} \to V_A$ which assigns an element $f(\mathbf{x})$ of V_A to each indeterminate $\mathbf{x} \in \mathbb{X}$. Any such assignment f extends in a natural way to a function $\hat{f} : V_A[\mathbb{X}] \to V_A$. Intuitively, given some $a \in V_A[\mathbb{X}]$ one simply replaces each $\mathbf{x} \in \mathbb{X}$ by its value $f(\mathbf{x})$. Rather than write $\hat{f}(a)$, we write $a[f]$, or even more informally, $a(\mathbf{x}, \mathbf{y}, \ldots)$ and $a(f(\mathbf{x}), f(\mathbf{y}), \ldots)$.

Definición

Una asignación $f : V_A[\mathbb{X}] \to V_A$ es una *solución de una ecuación* $\mathbf{x} = a(\mathbf{x}, \mathbf{y}, \ldots) \in V_A[\mathbb{X}]$ si

$$f(\mathbf{x}) = a(f(\mathbf{x}), f(\mathbf{y}), \ldots) \in V_A$$

Notar que $V_A[\mathbb{X}] = V_{A'}$
- f es una solución de un sistema de ecuaciones en $\mathbb{X}$ si es una solución de cada ecuación del sistema.

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.