PESQUIMAT 26(1): 71–87 (2023) ISSN:1560-912X/ ISSN-E:1609-8439
https://doi.org/10.15381/pesquimat.v26i1.25523 Facultad de Ciencias Matemáticas – UNMSM

Teoría de Tipos Dependientes: Una primera aproximación

Fernando Chu Rivera [1]

Resumen: Presentamos la Teoría de Tipos Dependientes, que uniformiza los conceptos de proposiciones y de conjuntos en uno solo más general, el de "tipos". Desarrollamos las nociones principales de esta teoría, y observamos que esta puede funcionar como un fundamento para estudiar las matemáticas, reemplazando la teoría de conjuntos. Ejemplificamos esto estudiando el caso de los números naturales, y explicamos cómo el uso de los asistentes de prueba nos permite tener una base más sólida para los resultados matemáticos.

Palabras clave: Teoría de Tipos, Tipos dependientes, Fundamentos de las matemáticas, Lógica, Asistentes de prueba.

Dependent Type Theory: a first approximation

Abstract: We present Dependent Type Theory, which combines the concepts of propositions and sets in a more general one, that of "types". We develop the main notions of this theory, and observe that it can serve as a foundation of mathematics, replacing set theory. We exemplify this studying the natural numbers, and explain how the use of proof assistants gives us a more rigorous foundation for mathematical results.

Keywords: Type Theory, Dependent types, Mathematical foundations, Logic, Proof assistants.

Recibido 20/02/2023 *Aceptado 09/04/2023* *Publicado online 30/06/2023*

[1]UNMSM, Facultad de Ciencias Matemáticas. e-mail: fernando.chu@unmsm.edu.pe

1. Introducción

La mayoría de la matemática actual está basada en la combinación de Teoría de Conjuntos y la Lógica Clásica. Una propuesta alternativa es la llamada Teoría de Tipos Dependientes, presentada por Per Martin-Löf [9] a comienzos de los 80s, la cual buscaba solucionar algunos problemas de impredicatividad y unificar los lenguajes de programación con la matemática.

Esta reemplaza las nociones de conjuntos y proposiciones por una sola: la de tipos. La unificación de estos constructos permite interrelacionarlos en un solo nivel, lo que tiene una serie de ventajas, enfatizamos en dos. Primero, permite plantear teorías de una forma diferente a la tradicional, véase por ejemplo la homotopía sintética presentada en [13]; y, segundo, permite la formalización de los resultados en un programa de computadora.

La Teoría de Tipos es todavía una teoría en desarrollo, con múltiples vertientes de estudio en diferentes ramas del saber, como las matemáticas, computación, lingüística, filosofía, entre otros. Aún así, ya existen múltiples lenguajes de programación que permiten redactar pruebas matemáticas utilizando esta teoría. Por ejemplo, en el lenguaje Lean [1] ya se ha formalizado gran parte del material que se ve en pregrado en los estudios de matemáticas [12], así como otros resultados más recientes (ver Sección 5).

En este artículo presentaremos los principales conceptos de esta teoría alternativa, desde la perspectiva matemática, y ejemplificamos el uso de estos para el estudio de los números naturales. Culminamos con una discusión de los asistentes de prueba.

2. Teoría de Tipos Dependientes

Así como en teoría de conjuntos, un conjunto está definido como aquello que satisface los axiomas de ZFC [6], un tipo es aquello que satisface las siguientes reglas por introducir. Sin embargo, una primera aproximación a los tipos es entenderlos como una colección de objetos matemáticos con cierta estructura interna común. De esta forma, existe el tipo de los naturales, el tipo de los espacios vectoriales, o el tipo de espacios métricos, por ejemplo.

2.1. Tipos, elementos y universos

Una de las nociones básicas en teoría de tipos es la de *pertenencia*. Similarmente a cómo en teoría de conjuntos escribimos $x \in X$ para expresar que el objeto x es un elemento del conjunto X, en la presente teoría diremos que un **objeto** x es un elemento del **tipo** X, y escribimos esto por $x : X$.

No obstante, estas nociones son diferentes en varios aspectos. Primero, en teoría de tipos, todo objeto tiene un tipo asociado al cual este pertenece (con una solo exepción, ver Regla 2.3). Los tipos, asimismo, también son objetos de otro tipo.

Definición 2.1. El tipo de un tipo es llamado un **universo**.

Asumiremos la existencia de una jerarquía ascendente e infinita de universos, esto se ve reflejada en la siguiente regla.

Regla 2.2. *Existe un universo $\mathcal{U}_0$ y para cada universo $\mathcal{U}_i$, existe un universo $\mathcal{U}_{i+1}$ tal que $\mathcal{U}_i : \mathcal{U}_{i+1}$.*

Así, una vez introducido el tipo de los naturales, tendremos que $0 : \mathbb{N}$, $\mathbb{N} : \mathcal{U}_0$, $\mathcal{U}_0 : \mathcal{U}_1$, etc; por ejemplo. Adicionalmente, los universos son cumulativos; es decir, tenemos la siguiente regla.

Regla 2.3. *Si para un tipo X se tiene que $X : \mathcal{U}_i$, entonces también se tiene que $X : \mathcal{U}_{i+1}$.*

Utilizamos la palabra "regla" en vez de "axioma", pues esta denota un significado ligeramente distinto. En matemática, introducir un axioma presupone la existencia de algunas reglas ya establecidas. En matemática clásica, usualmente suponemos como dadas las reglas de la lógica proposicional. Dado que la presente teoría subsume también la lógica, el resto de supuestos en esta sección también serán llamados "reglas".

Omitiremos los subíndices de los universos en la mayoría de escenarios, por lo que interpretaremos expresiones sin sentido como $\mathcal{U} : \mathcal{U}$ agregando índices adecuados, obteniendo $\mathcal{U}_i : \mathcal{U}_{i+1}$.

2.2. Igualdades por definición

La presente teoría posee dos conceptos de igualdad: **igualdades por definición** e **igualdades proposicionales**. Ahora presentamos la primera de estas.

Definición 2.4. Dados dos objetos x y y, diremos que estos son iguales por definición si es que podemos reemplazar uno de estos donde sea que aparezca el otro, sin modificar la verdad o falsedad de la proposición. Escribiremos $x \equiv y$ cuando este sea el caso.

Nótese que dada esta definición, es inmediato que $\equiv$ es una relación de equivalencia. Además, como se esperaría, un caso particular de esto son las definiciones usuales, entendidas como abreviaciones. Escribiremos $a :\equiv b$ para definir el símbolo 'a' como una abreviación de la expresión 'b'.

2.3. El tipo de funciones

Mientras que en matemática clásica el conjunto de funciones de X a Y está definido como un subconjunto del conjunto $X \times Y$ que cumple ciertas condiciones, en la presente teoría tenemos un tipo que captura de manera precisa nuestras intuiciones respecto a cómo se comportan las funciones.

Regla 2.5. *Para todo par de tipos X y Y, existe el **tipo de funciones** de X a Y, denotado por $X \to Y$. Elementos de este tipo son llamados **funciones**.*

Entonces con la notación que hemos introducido, escribimos $f : X \to Y$ para decir que f es una función de X a Y, lo que concuerda con la notación usual. En esta caso, llamaremos a X como el **dominio** y a Y como el **codominio** de f.

Para formar una función, solo necesitamos una regla de correspondencia; es decir, tenemos la siguiente regla.

Regla 2.6. *Sean X y Y tipos. Dado un x arbitrario en X, si podemos generar un elemento y de Y a partir de x, entonces tenemos una función $f : X \to Y$. Escribiremos $\lambda(x : X).\, y$ para referirnos a la función f dada por esta regla.*

La función $\lambda(x : X).\, y$ puede entenderse como la función $f : X \to Y$ definida por $f(x) = y$, solo que se ha omitido el nombre f. Por este motivo, a veces en la literatura estas funciones se llaman funciones anónimas. En este documento nosotros las llamaremos funciones lambda, el nombre original que les dio Alonzo Church [5].

En la expresión $\lambda(x : X).\, y$, se dice que x es una **variable ligada**. Esto es similar a cómo las expresiones ''$\forall x$' o ''$\int - \mathrm{dx}$' ligan la variable x dentro de estas. Si una variable no es ligada, se dice que es una **variable libre**.

Ejemplo 2.7. Para todo tipo X, podemos definir su función identidad por

$$\mathsf{id}_X ::\equiv \lambda(x : X).\, x$$

Las reglas hasta ahora solo nos indican cómo construir nuevas funciones, también es necesario reglas que describan como aplicarlas en elementos.

Regla 2.8. *Sean X y Y tipos. Dada una función $\lambda(x:X).\, y : X \to Y$ y dado un elemento $c : X$, entonces $f(c)$ es igual por definición a $y[c/x]$, donde la expresión $y[c/x]$ indica que todas las apariciones libres de x fueron reemplazadas con c.*

Ejemplo 2.9. Continuando el ejemplo previo, se tiene entonces que

$$\mathsf{id}_X(c) \equiv (\lambda(x:X).\, x)(c) \equiv x[c/x] \equiv c$$

Ejemplo 2.10. Dados tipos X, Y, y Z, y dadas funciones arbitraria $f : X \to Y$ y $g : Y \to Z$, podemos definir una función

$$h : X \to Z$$
$$h :\equiv \lambda(x:X).\, g(f(x))$$

Pero como g fue arbitrario, esto nos da una función

$$f^* : (Y \to Z) \to (X \to Z)$$
$$f^* :\equiv \lambda(g:Y \to Z).\, \big(\lambda(x:X).\, g(f(x))\big)$$

Pero, nuevamente, el f fue arbitrario, por lo que tenemos una función

$$\mathsf{comp}_{X,Y,Z} : (Y \to Z) \to ((X \to Y) \to (X \to Z))$$
$$\mathsf{comp}_{X,Y,Z} :\equiv \lambda(f:X \to Z).\, \big(\lambda(g:Y \to Z).\, \big(\lambda(x:X).\, g(f(x))\big)\big)$$

Esta última función es la función de composición de funciones usual, pues se tiene que

$$\mathsf{comp}_{X,Y,Z}(f)(g)(x) \equiv f(g(x)),$$

de esto se deriva que la composición es asociativa.

Con esta composición, tenemos una categoría Type, cuyos objetos son tipos y los morfismos son funciones entre tipos. Nótese, además, que la composición depende de X, Y, y Z; es decir, tenemos en realidad una familia de funciones de composiciones indexadas por triples de tipos. Veremos cómo podemos simplificar esto en la siguiente subsección.

La última regla de funciones es de unicidad respecto a "abstracciones".

Regla 2.11. *Sea $f : X \to Y$ una función, entonces tenemos que $f \equiv \lambda(x:X).\, f(x)$.*

Esta regla indica que si formamos una nueva función lambda, que recibe x y devuelve $f(x)$ entonces esta es la misma función que la función f original.

2.4. El tipo de funciones dependientes

En MC, una práctica común es utilizar sucesiones infinitas de elementos x_i de un conjunto X. Formalmente, esto corresponde a una función $f : \mathbb{N} \to X$. Similarmente, a veces es necesario indexar no solo elementos, sino conjuntos con cierta estructura por otros conjuntos.

Por ejemplo, a cada punto p en una variedad M, le corresponde su espacio tangente T_pM. Se entiende que esta última construcción está asociada a cierta función $g : M \to \bigcup_{p\in M}\{T_pM\}$. En este último caso, es común decir que $\{T_pM\}_{p\in M}$ es una familia de conjuntos indexada por $p \in M$. Presentamos el constructo análogo en teoría de tipos.

Definición 2.12. Si tenemos una función $P : X \to \mathcal{U}$ cuyo codominio es un universo, entonces diremos que P es una **familia de tipos** indexada por X.

Ejemplo 2.13. Cada punto x en un espacio topológico X tiene asociado un grupo, su grupo fundamental. Es decir, $P :\equiv \lambda(x : X).\, \pi_1(x, X)$ es una familia de tipos indexada por el tipo de los espacios topologicos.

Ejemplo 2.14. Sea $Y : \mathcal{U}$ un tipo, entonces tenemos una familia de tipos $P :\equiv \lambda(x : X).\, Y$ indexada por X. En este caso, decimos que P es una **familia constante**.

Nótese que en el caso de familias constantes, una función $f : X \to Y$ asigna a cada $x : X$ un elemento $f(x) : Y$. Esto se puede generalizar para familias arbitrarias.

Ejemplo 2.15. Sea X un espacio topológico, para cada $x : X$ podemos escoger un elemento de $\pi_1(x, X) : \mathcal{U}$, el elemento neutro de ese grupo $0_{\pi_1(x,X)}$.

En este último ejemplo tenemos una regla de correspondencia, por lo que se esperaría que podamos formar una función f con el tipo $X \to \pi_1(x, X)$. Sin embargo, este tipo previo no está bien tipado, pues el x que aparece en el codominio no está definido. El problema es que el codominio depende del $x : X$ escogido en el dominio, para estos casos necesitaremos una generalización del tipo de funciones.

Regla 2.16. *Para todo tipo X y familia de tipos $P : X \to \mathcal{U}$, existe el **tipo de funciones dependientes** de X en P, denotado por $\prod_{(x:X)} P(x)$. Elementos de este tipo son llamados **funciones dependientes**.*

Al igual que para funciones no dependientes, basta una regla de correspondencia para formar elementos de este tipo.

Regla 2.17. *Para todo tipo X y familia de tipos $P : X \to \mathcal{U}$, dado un x arbitrario en X, si podemos generar un elemento $y : P(x)$ a partir de x, entonces tenemos una función $f : \prod_{(x:X)} P(x)$. Escribiremos $\lambda(x : X).\, y$ para referirnos a la función f dada por esta regla.*

Ejemplo 2.18. Dado un espacio topológico X, existe el tipo $\prod_{(x:X)} \pi_1(x, X)$, y una de las funciones pertenecientes a este tipo es $\lambda(x : X).\, 0_{\pi_1(x,X)}$.

Finalmente, al igual que para las funciones no dependientes, existen reglas análogas que definen el comportamiento de las funciones y su unicidad.

Regla 2.19. *Para todo tipo X y familia de tipos $P : X \to \mathcal{U}$. Dada una función $\lambda(x : X).\, y : \prod_{(x:X)} P(x)$ y dado un elemento $c : X$, entonces $f(c)$ es igual por definición a $y[c/x]$.*

Regla 2.20. *Sea $f : \prod_{(x:X)} P(x)$ una función, entonces tenemos que $f \equiv \lambda(x : X).\, f(x)$.*

Veamos unos ejemplos, y algunas convenciones que facilitarán la siguiente discusión.

Ejemplo 2.21. Existe una función id que asigna a cada tipo $X : \mathcal{U}$, su función identidad $\mathsf{id}_X : X \to X$.

$$\mathsf{id} : \prod_{X:\mathcal{U}} (X \to X)$$
$$\mathsf{id} :\equiv \lambda(X :\mathcal{U}).\, \lambda(x : X).\, x$$

Cuando puedan ser inferidas o no sean relevantes, omitiremos el tipo de las variables dentro de una función lambda. Por ejemplo, escribiríamos $\lambda(x:X).\,\lambda(y:Y).\,\Phi$ como $\lambda x.\,\lambda y.\,\Phi$.

Además, el símbolo '$\to$' asocia hacia la derecha; es decir, se entiende por $X \to Y \to Z$ como $X \to (Y \to Z)$. Todos los demás constructos por introducir asociarán hacia la izquierda, de tal forma que $X \times Y \times Z$ es $(X \times Y) \times Z$. Por otro lado, si tenemos $f : X_1 \to \cdots \to X_{n-1} \to X_n$, escribiremos $f(x_1)(x_2)\cdots(x_n)$ como $f(x_1, x_2, \ldots, x_n)$.

Para funciones dependientes, por convención, las variables ligadas por Π tienen menor precedencia que otros operadores dentro de un tipo, por lo que $\prod_{(x:X)} Y \to Z$ se entiende como $\prod_{(x:X)}(Y \to Z)$, por ejemplo.

Finalmente, es común en MC definir funciones a partir de reglas de correspondencia; es decir, definiendo f por su comportamiento en un x arbitrario. Utilizaremos esta misma práctica, entendiendo estas definiciones como una abreviación de funciones lambda. Por ejemplo, la definición de la función identidad (Ejemplo 2.21) puede ser reescrita simplemente como

$$\begin{aligned} \mathsf{id} &: \prod_{X:\mathcal{U}} X \to X \\ \mathsf{id}(X, x) &:\equiv x \end{aligned}$$

Definición 2.22. Podemos definir la función de composición de funciones

$$\mathsf{comp}^* :\equiv \prod_{(X:\mathcal{U})} \prod_{(Y:\mathcal{U})} \prod_{(Z:Y\to\mathcal{U})} \left(\prod_{y:Y} Z(y)\right) \to \prod_{(f:X\to Y)} \prod_{(x:X)} Z(f(x))$$

dada por

$$\mathsf{comp}^*(X, Y, Z, g, f, x) :\equiv g(f(x))$$

Esta función generaliza a la función del Ejemplo 2.10, en el sentido de que

$$\mathsf{comp}_{X,Y,Z}(g, f) \equiv \mathsf{comp}^*(X, Y, Z, g, f)$$

Como es usual, escribiremos $g \circ f$ en vez de $\mathsf{comp}^*(X, Y, Z, g, f)$, dejando los tipos X, Y y Z implícitos.

Nótese que para familias $P :\equiv \lambda(x:X).\,Y$ constantes una función $f : \prod_{(x:X)} P(x)$ es lo mismo que una función $f : X \to Y$, por lo que utilizaremos ambos $\prod_{(x:X)} P(x)$ y $X \to Y$ indistintamente en estos casos.

2.5. Tipos Inductivos

Una de las principales formas de introducir nuevos tipos es a través de la definición de tipos inductivos. Estos pueden entenderse como aquellos objetos libres que satisfacen cierta estructura interna. Veamos unos ejemplos.

Definición 2.23. El **tipo de los naturales** $\mathbb{N} : \mathcal{U}_0$ es el tipo libremente generado por dos constructores[1]:

- $0 : \mathbb{N}$
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$

Definimos los símbolos usuales para los números naturales como aplicaciones repetidas del constructor succ. Es decir $1 :\equiv \mathsf{succ}(0)$, $2 :\equiv \mathsf{succ}(1)$, $3 :\equiv \mathsf{succ}(2)$, $\ldots$.

Este tipo es libremente generado en el sentido que satisface la siguiente regla.

[1]Se le llama constructores a aquellas constantes o funciones dentro de la definición de un tipo inductivo que permiten generar nuevos elementos de este tipo.

Regla 2.24. *Sea $P : \mathbb{N} \to \mathcal{U}$ una familia de tipos sobre $\mathbb{N}$. Si tenemos*

- *Un elemento $c_0 : P(0)$*
- *Y, una función $c_s : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}(n))$*

Entonces tenemos una función $f : \prod_{(n:\mathbb{N})} P(n)$, tal que

$$f(0) \equiv c_0 \qquad y \qquad f(\mathsf{succ}(n)) \equiv c_s(n, f(n))$$

Nótese que los requerimientos de la regla están basados en los constructores del tipo de los naturales, y que, además, está regla es equivalente a la existencia de una función

$$\mathsf{ind}_{\mathbb{N}} : \prod_{P:\mathbb{N}\to\mathcal{U}} P(0) \to \left(\prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}(n))\right) \to \prod_{n:\mathbb{N}} P(n)$$

Si entendemos la familia de tipos P como una propiedad que un natural puede tener, esta función es el principio de inducción usual de la matemática clásica. Debido a esto, llamaremos a estos tipos de reglas como principios de inducción también.

Ejemplo 2.25. Sea $P :\equiv \lambda(n : \mathbb{N}).\, \mathbb{N}$, y sean $c_0 :\equiv 0$ y $c_s(n) :\equiv \mathsf{succ}(\mathsf{succ}(n))$, la regla previa entonces nos da una función $\mathsf{double} : \mathbb{N} \to \mathbb{N}$. Esta satisface, por ejemplo, que

$$\begin{aligned}
\mathsf{double}(2) &\equiv \mathsf{double}(\mathsf{succ}(\mathsf{succ}(0))) \\
&\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(\mathsf{succ}(0)))) \\
&\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{double}(0))))) \\
&\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(0)))) \\
&\equiv 4
\end{aligned}$$

Es conveniente introducir estás funciones a través de una notación estandarizada, llamada **búsqueda de patrones**. Esta asigna a cada constructor de un tipo inductivo, su imagen apropiada. Por ejemplo la función double introducida anteriormente, hubiese sido definida por

$$\begin{aligned}
\mathsf{double} &: \mathbb{N} \to \mathbb{N} \\
\mathsf{double}(0) &:\equiv 0 \\
\mathsf{double}(\mathsf{succ}(n)) &:\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(n)))
\end{aligned}$$

Escrito de esta manera, es evidente de que estamos utilizando en realidad el principio de recursión. Es decir, en esta teoría el principio de recursión es un caso particular del principio de inducción. En general, el principio de recursión es el principio de inducción aplicado a una familia constante.

Veamos otros ejemplos de tipos inductivos.

Definición 2.26. El tipo $\mathbf{1} : \mathcal{U}_0$ es el tipo libremente generado por el siguiente constructor:

- $\star : \mathbf{1}$

Intuitivamente, el tipo $\mathbf{1}$ es el tipo con un solo elemento, por lo que para definir una función desde este, basta definirla en solo este elemento.

Regla 2.27. *Sea $P : \mathbf{1} \to \mathcal{U}$ una familia de tipos sobre $\mathbf{1}$. Si tenemos*

- *Un elemento $c : P(\star)$*

Entonces tenemos una función $f_c : \prod_{(x:\mathbf{1})} P(x)$, tal que $f_c(\star) \equiv c$.

Definición 2.28. El tipo $\mathbf{0} : \mathcal{U}_0$ es el tipo libremente generado por cero constructores.

Intuitivamente, el tipo $\mathbf{0}$ es el tipo vacío, por lo que siempre tenemos funciones (triviales) desde este.

Regla 2.29. *Sea $P : \mathbf{0} \to \mathcal{U}$ una familia de tipos sobre $\mathbf{0}$. Entonces tenemos una función $f : \prod_{(x:\mathbf{0})} P(x)$.*

También podemos generar tipos inductivos indexados por otros tipos.

Definición 2.30. Sean $X : \mathcal{U}$ y $P : X \to \mathcal{U}$, el **tipo de pares dependientes** $\sum_{(x:X)} P(x) : \mathcal{U}$ es el tipo libremente generado por el siguiente constructor

- Una función $\mathsf{pair} : \prod_{(x:X)} P(x) \to \sum_{(x:X)} P(x)$

Intuitivamente, el tipo $\sum_{(x:X)} P(x)$ es el tipo que tiene como elementos pares (x, y) donde $x : X$ y $y : P(x)$, y $\mathsf{pair}(x)(y)$ es justamente el par ordenado (x, y). Por este motivo, usaremos la notación usual de (x, y) en vez de $\mathsf{pair}(x)(y)$.

Dado esto, y dados tipos $X : \mathcal{U}$ y $Y : \mathcal{U}$, definimos $X \times Y :\equiv \sum_{(x:X)} P_Y$, donde $P_Y :\equiv \lambda(x : X).\, Y$ es la familia constante en Y.

Regla 2.31. *Sea $C : \sum_{(x:X)} P(x) \to \mathcal{U}$ una familia de tipos, si tenemos*

- *Una función $g : \prod_{(x:X)} P(x) \prod_{(y:P(x))} C((x, y))$*

Entonces tenemos una función $f : \prod_{(z:\sum_{(x:X)} P(x))} C(z)$ tal que $f(x, y) \equiv g(x)(y)$.

Ejemplo 2.32. Usando búsqueda de patrónes, definimos las proyeciones usuales

$$\mathsf{pr}_1 : \prod_{z:\sum_{(x:X)} P(x)} X$$
$$\mathsf{pr}_1(a, b) :\equiv a$$

y

$$\mathsf{pr}_2 : \prod_{z:\sum_{(x:X)} P(x)} P(\mathsf{pr}_1(z))$$
$$\mathsf{pr}_2(a, b) :\equiv b$$

Omitiremos a veces los paréntesis de un par cuando estos estén dentro de una aplicación de una función o dentro de otro par. Por ejemplo, escribiremos $C(a, b)$ en vez de $C((a, b))$, y (a, b, c) en vez de $(a, (b, c))$ o $((a, b), c)$.

Otro ejemplo importante de tipos inductivos indexados es el coproducto.

Definición 2.33. Sean $X : \mathcal{U}$ y $Y : \mathcal{U}$, el **coproducto** $X + Y$ es el tipo libremente generado por los siguientes constructores

- Una función $\mathsf{inl} : X \to (X + Y)$
- Una función $\mathsf{inr} : Y \to (X + Y)$

Intuitivamente, el tipo $X + Y$ es la unión disjunta de X y Y.

Regla 2.34. *Sea $P : (X + Y) \to \mathcal{U}$ una familia de tipos, si tenemos*

- *Una función $f_1 : \prod_{(x:X)} P(\mathsf{inl}(x))$*
- *Una función $f_2 : \prod_{(y:Y)} P(\mathsf{inr}(y))$*

Entonces tenemos una función $f : \prod_{(z:X+Y)} P(z)$ tal que $f(\mathsf{inl}(x)) \equiv f_1(x)$ y $f(\mathsf{inr}(y)) \equiv f_2(y)$.

Cómo mencionamos al comienzo de este artículo, los tipos pueden representar proposiciones y, en particular, relaciones. Posiblemente la más importante de estas es cuando dos objetos son iguales entre sí.

Definición 2.35. Sean $X : \mathcal{U}$ un tipo y $x, y : X$ objetos de este. El **tipo de identidades** $x =_X y : \mathcal{U}$ es el tipo libremente generado por el siguiente constructor

- Una función $\mathsf{refl} : \prod_{(x:X)}(x =_X x)$

Intuitivamente, el tipo $x =_X y$ representa la proposición de que x y y son iguales. Omitiremos el subíndice X cuando este no sea relevante, o pueda ser inferido, escribiendo solo $x = y$.

Regla 2.36. *Sea $P : \prod_{(x,y:X)}(x =_X y) \to \mathcal{U}$ una familia de tipos, si tenemos*

- *Una función $g : \prod_{(x:X)} P(x, x, \mathsf{refl}_x)$*

Entonces tenemos una función $f : \prod_{(x,y:X)} \prod_{(p:x=y)} P(x, y, p)$ tal que $f(x, x, \mathsf{refl}_x) \equiv g(x)$.

Expresada de otra forma, esta regla nos indica que para definir una función dependiente $f : \prod_{(x,y:X)} \prod_{(p:x=y)} P(x, y, p)$ basta asumir que y es x y p es refl_x. Veamos algunos ejemplos de esto.

Lema 2.37. *Para todo tipo A, $x, y : A$, y $p : x = y$, existe una función*

$$\mathsf{sym} : x = y \to y = x$$

Demostración. Asumiendo que $y \equiv x$ y el camino es refl_x, ponemos $\mathsf{sym}(\mathsf{refl}_x) :\equiv \mathsf{refl}_x : x = x$. □

Lema 2.38. *Para todo tipo A y todo $x, y, z : A$ existe una función*

$$\mathsf{trans} : (x = y) \to (y = z) \to (x = z)$$

Demostración. Podemos asumir que y es x y que el primer camino es refl_x, por lo que reducimos la tarea a encontrar una función $\mathsf{trans}(\mathsf{refl}_x, -) : (x = z) \to (x = z)$.

Nuevamente, podemos asumir que z es x y el segundo camino también es refl_x, con lo que solo es necesario encontrar un elemento de $(x = x)$, pero para esto podemos tomar refl_x. □

Con la definición de refl_x y con los lemas anteriores, vemos que la igualdad es una relación de equivalencia. Como se esperaría, esta se preserva bajo funciones y propiedades.

Lema 2.39. *Sea $f : X \to Y$ una función, para todo $x, y : X$ existe una función*

$$\mathsf{cong}_f : (x =_X y) \to (f(x) =_Y f(y))$$

Demostración. Asumiendo que el camino del dominio es refl_x, necesitamos encontrar un camino $f(x) =_Y f(x)$, pero tenemos que $\mathsf{refl}_{f(x)}$ es un elemento de este tipo. □

Lema 2.40. *Sea P una familia de tipos sobre X, y sea $p : x =_X y$ un camino, entonces existe una función*

$$\mathsf{subst}^P(p, -) : P(x) \to P(y)$$

Demostración. Podemos asumir que p es refl_x, en cuyo caso necesitamos una función $P(x) \to P(x)$, tomamos a la función identidad $\mathsf{id}_{P(x)}$ para esto. □

Veamos ahora como los conceptos introducidos hasta ahora permiten a la teoría de tipos dependientes formar una base para el desarrollo matemático.

3. La interpretación lógica

3.1. Proposiciones como tipos

En la introducción habíamos mencionado que las proposiciones podían ser representadas como tipos. Comenzaremos analizando cada uno de los constructos lógicos principales en matemática clásica.

- Para mostrar que se cumple $P \wedge Q$, es suficiente mostrar que se cumplen ambos P y Q. Podemos entonces considerar una prueba de $P \wedge Q$ como un par de pruebas de P y Q.
- Para mostrar que se cumple $P \vee Q$, es suficiente mostrar que se cumple alguno de P o Q. Podemos entonces considerar una prueba de $P \vee Q$ como la unión disjunta de pruebas de P y Q.
- Para mostrar que se cumple $P \implies Q$, es suficiente mostrar que dado P, se puede mostrar que se cumple Q. Podemos entonces considerar una prueba de $P \implies Q$ como una función que toma una prueba de P y devuelve una prueba de Q.
- Para mostrar que se cumple $\neg P$, es suficiente mostrar que si es que tenemos P, podemos llegar a una contradicción. Podemos entonces considerar una prueba de $\neg P$ como una función que toma una prueba de P y devuelve una prueba de una contradicción.
- Para mostrar que se cumple $\forall x \in A, P(x)$, es suficiente mostrar que, dado un x arbitrario en A, se puede mostrar que este satisface $P(x)$. Podemos entonces considerar una prueba de $\forall x \in A, P(x)$ como una función que toma cualquier elemento $x \in A$ y devuelve una prueba de $P(x)$.
- Para mostrar que se cumple $\exists x \in A, P(x)$, es suficiente mostrar que existe un $a \in A$ tal que $P(a)$. Podemos entonces considerar una prueba de $\exists x \in A, P(x)$ como un par, el cual consiste de un elemento $a \in A$ y una prueba de $P(a)$.

La interpretación de estos constructos de esta manera es llamada la **interpretación BHK**, debido a los nombres de sus principales proponentes L. E. J. Brouwer, Arend Heyting, y Andrey Kolmogorov. Notemos su similitud con las reglas de introducción para los tipos que ya hemos introducido.

Por ejemplo, la regla de introducción para el tipo del producto indica que para construir un elemento de $P \times Q$ es suficiente mostrar dos elementos $a : P$ y $b : Q$, y el nuevo elemento construido sería el par (a, b) de estos. Así, los tipos se interpretan como proposiciones y los elementos de estos tipos se interpretan como pruebas o evidencia de estas proposiciones. Esta correspondencia entre lógica y tipos es llamada el **isomorfismo de Curry-Howard**.

Matemática Clásica	Teoría de Tipos Dependientes
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$A \implies B$	$A \to B$
$A \iff B$	$(A \to B) \times (B \to A)$
$\neg A$	$A \to \mathbf{0}$
$\forall x \in A, P(x)$	$\prod_{(x:A)} P(x)$
$\exists x \in A, P(x)$	$\sum_{(x:A)} P(x)$

Cuadro 1: Isomorfismo de Curry-Howard.

La correspondencia es relativamente simple por lo que no la explicaremos más a detalle, solo el caso de $\neg A$ merece una mayor atención. En teoría de tipos, el $\mathbf{0}$ toma el rol de un absurdo, puesto que su mismo principio de inducción indica que es posible formar una función con cualquier codominio dado un elemento de $\mathbf{0}$. Por este motivo, interpretamos $A \to \mathbf{0}$ como $\neg A$.

Definición 3.1. Diremos que dos tipos A y B son lógicamente equivalentes cuando se tiene $A \iff B$; es decir, tenemos dos funciones $f : A \to B$ y $g : B \to A$, como indicado en el Cuadro 1.

Veamos algunos ejemplos de lógica proposicional y lógica predicativa.

Ejemplo 3.2. Una de las leyes de Morgan indica que para todo par de proposiciones A y B tenemos

$$\neg A \vee \neg B \implies \neg(A \wedge B).$$

El equivalente en teoría de tipos es la existencia de una función

$$f : \prod_{(A:\mathcal{U})} \prod_{(B:\mathcal{U})} (A \to \mathbf{0}) + (B \to \mathbf{0}) \to (A \times B \to \mathbf{0})$$

En efecto, combinando la búsqueda de patrones en el coproducto y en el producto, podemos definir f como

$$f(A, B, \mathsf{inl}(f_1), (a, b)) :\equiv f_1(a)$$
$$f(A, B, \mathsf{inr}(f_2), (a, b)) :\equiv f_2(b)$$

A veces omitiremos la mención explícita de los Π tipos involucrados, cuando estos toman el rol lógico de "para todo", entendiendo que la prueba que estamos construyendo es una función dependiente con un dominio apropiado.

Ejemplo 3.3. Consideremos la siguiente implicación usada comúnmente

$$\exists x \in A, \neg P(x) \implies \neg(\forall x \in A, P(x))$$

Esta es representada por la existencia de una función

$$f : \textstyle\sum_{(x:A)}(P(x) \to \mathbf{0}) \to (\prod_{(x:A)} P(x)) \to \mathbf{0}$$

Esta puede ser definida por

$$f((a, g), h) :\equiv g(h(a))$$

Ejemplo 3.4. El axioma de elección en MC indica que dada una colección de conjuntos no vacíos X, es posible crear una función $f : X \to \bigcup X$ que selecciona un elemento de cada conjunto. Es decir,

$$\left(\forall x \in X, \exists y \in \bigcup X, (y \in x)\right) \implies \left(\exists f : X \to \bigcup X, \forall x \in X, (f(x) \in x)\right)$$

En teoría de tipos, esto podría ser entendido[2] como un caso particular de

$$\mathsf{ac} : \left(\textstyle\prod_{(x:X)}\sum_{(y:Y)} R(x, y)\right) \to \left(\sum_{(f:X\to Y)}\prod_{(x:X)} R(x, f(x))\right)$$

[2]La verdadera formalización del axioma de elección como visto de manera clásica, es ligeramente distinta a la presentación actual, ver [13, Sección 3.8].

Podemos definir esta función como

$$\mathsf{ac}(h) :\equiv \Big(\lambda(x:X).\,\mathsf{pr}_1(h(x)), \lambda(x:X).\,\mathsf{pr}_2(h(x))\Big)$$

Esta función está bien definida puesto que

$$\lambda(x:X).\,\mathsf{pr}_1(g(x)) : X \to Y,$$
$$\lambda(x:X).\,\mathsf{pr}_2(g(x)) : \textstyle\prod_{(x:X)} R(x, \mathsf{pr}_1(g(x))).$$

como es requerido por el tipo del codomino de ac.

Definición 3.5. Dado un tipo X, si existe un elemento c tal que $c : X$, diremos que X es un tipo **habitado**.

Así, se dice una proposición P es **verdadera** cuando es un tipo habitado. El ejemplo previo muestra que el axioma de elección es verdadero en esta teoría.

3.2. Grupos y teorías algebraicas

Veamos cómo los conceptos introducidos nos permiten formalizar el concepto de un grupo. Recordemos que en matemática clásica un grupo es par $(G, *)$, tal que G es un conjunto y $*$ es una operación binaria en G tal que

- $*$ es asociativa,
- existe un elemento neutro e, tal que para todo $g \in G$ se tiene que $e * g = g * e = g$, y
- para todo $g \in G$ existe un $g^{-1} \in G$ tal que $g * g^{-1} = g^{-1} * g = e$.

Esta estructura puede ser reflejada directamente en la siguiente definición

Definición 3.6. Dado un tipo A, diremos que este tiene una **estructura de grupo** si es que el tipo

$$\begin{aligned}\mathsf{GroupStr}(A) :\equiv \sum_{m:A\to A\to A} \prod_{x,y,z:A} (m(x, m(y,z)) = m(m(x,y),z)) \\ \times \sum_{(e:A)} \prod_{(g:A)} (m(e,g) = g) \times (m(g,e) = g) \\ \times \sum_{(i:A\to A)} \prod_{(g:A)} (m(g, i(g)) = e) \times (m(i(g), g) = e)\end{aligned}$$

está habitado.

Cada línea de la definición previa corresponde a uno de los puntos de la definición clásica. Con el concepto de estructura ya definido, podemos definir lo que es un grupo.

Definición 3.7. Un **grupo** es un tipo junto con una estructura de grupo. Es decir, un elemento del tipo

$$\mathsf{Group} :\equiv \sum_{A:\mathcal{U}} \mathsf{GroupStr}(A)$$

Nótese que, a diferencia de en matemática clásica, sí tenemos un tipo de todos los grupos, el cual es Group.

Es claro que una formalización similar aplica para la gran mayoría de teorías algebraicas, como las de los anillos, módulos, categorías, etc.

3.3. Axiomas

En matemática clásica, cuando dos funciones $f, g : A \to B$ son iguales en cada punto del dominio, estas son iguales. Entonces, uno esperaría que el siguiente tipo esté habitado

$$\mathsf{FE} :\equiv \left(\prod_{x:A} (f(x) =_{B(x)} g(x))\right) \implies (f = g)$$

Sin embargo, no es posible demostrar esto con las reglas introducidas. Para esto, podría introducirse el siguiente axioma:

Axioma 3.8 (Extensionalidad de funciones)**.** *El tipo* FE *está habitado*

Este axioma entonces, postula la existencia de un elemento funext en el tipo FE. Todos los axiomas en teoría de tipos son de esta forma; es decir, indican la existencia de un elemento en algún tipo.

Existen varios axiomas adicionales que se pueden añadir a la teoría, pero no ahondaremos en estos pues no los necesitaremos. Enfatizamos que los axiomas se utilizan aquí para probar nuevos resultados, no para expresara una teoría. Para eso están los pares dependientes, como ejemplificamos para el caso de grupos.

4. Aplicaciones

A modo de ejemplo, veamos cómo podemos utilizar los conceptos introducidos para el estudio de los números naturales.

Definición 4.1. La operación de suma está definida

$$\begin{aligned} \mathsf{add} &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ \mathsf{add}(0, n) &:\equiv n \\ \mathsf{add}(\mathsf{succ}(m), n) &:\equiv \mathsf{succ}(\mathsf{add}(m)(n)) \end{aligned}$$

Como es común, utilizaremos $m + n$ para referirnos a $\mathsf{add}(m)(n)$.

Definición 4.2. La operación de multiplicación está definida

$$\begin{aligned} \mathsf{mul} &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ \mathsf{mul}(0, n) &:\equiv 0_{\mathbb{N}} \\ \mathsf{mul}(\mathsf{succ}(m), n) &:\equiv m + (\mathsf{mul}(m)(n)) \end{aligned}$$

Como es común, utilizaremos $m \times n$ para referirnos a $\mathsf{mul}(m)(n)$.

También podemos definir propiedades, como la de ser un divisor o un número primo.

Definición 4.3. La relación de divisibilidad $\mathsf{divisib}$ está definida por

$$\begin{aligned} \mathsf{divisib} &: \mathbb{N} \to \mathbb{N} \to \mathcal{U} \\ \mathsf{divisib}(m, n) &:\equiv \sum_{k:\mathbb{N}} m \times k = n \end{aligned}$$

Como es común, utilizaremos $m \mid n$ para referirnos a $\mathsf{divisib}(m)(n)$.

Definición 4.4. La propiedad de ser un número primo está definida por

$$\begin{aligned}\mathsf{prime} &: \mathbb{N} \to \mathbb{N} \to \mathcal{U}\\ \mathsf{prime}(m) &:\equiv ((m = 1) \to 0) \times \prod_{k:\mathbb{N}} (k \mid m) \to ((k = 1) + (k = m))\end{aligned}$$

Finalmente, mostramos cómo podemos demostrar proposiciones a través de un ejemplo.

Proposición 4.5. *La suma en los naturales es conmutativa.*

Demostración. La proposición puede ser expresada como el tipo

$$\prod_{(m:\mathbb{N})} \prod_{(n:\mathbb{N})} m + n = n + m$$

Para generar una función dentro de este tipo, necesitaremos unas funciones previas. Definimos

$$\begin{aligned}\mathsf{right}{-}0_{\mathbb{N}} &: \prod_{n:\mathbb{N}} n = n + 0\\ \mathsf{right}{-}0_{\mathbb{N}}(0) &:\equiv \mathsf{refl}_0\\ \mathsf{right}{-}0_{\mathbb{N}}(\mathsf{succ}(n)) &:\equiv \mathsf{cong}_{\mathsf{succ}}(\mathsf{right}{-}0_{\mathbb{N}}(n))\end{aligned}$$

Nótese que la función está bien definida pues $\mathsf{right}{-}0_{\mathbb{N}}(\mathsf{succ}(n))$ pertenece al tipo $\mathsf{succ}(n + 0) = \mathsf{succ}(n)$, pero por definición, $\mathsf{succ}(n + 0) \equiv \mathsf{succ}(n)$, por lo que $\mathsf{cong}_{\mathsf{succ}}(\mathsf{right}{-}0_{\mathbb{N}}(n))$ efectivamente pertenece a este tipo.

Similarmente, tenemos

$$\begin{aligned}\mathsf{right}{-}\mathsf{succ}_{\mathbb{N}} &: \prod_{(m:\mathbb{N})} \prod_{(n:\mathbb{N})} \mathsf{succ}(m + n) = m + \mathsf{succ}(n)\\ \mathsf{right}{-}\mathsf{succ}_{\mathbb{N}}(0, n) &:\equiv \mathsf{refl}_n\\ \mathsf{right}{-}\mathsf{succ}_{\mathbb{N}}(\mathsf{succ}(m), n) &:\equiv \mathsf{cong}_{\mathsf{succ}}(\mathsf{right}{-}\mathsf{succ}\mathbb{N}(m, n))\end{aligned}$$

Nuevamente, nótese que $\mathsf{right}{-}\mathsf{succ}_{\mathbb{N}}(\mathsf{succ}(m), n)$ pertenece al tipo $\mathsf{succ}(\mathsf{succ}(m)+n) = \mathsf{succ}(m)+\mathsf{succ}(n)$, pero este es igual por definición a $\mathsf{succ}(\mathsf{succ}(m + n)) = \mathsf{succ}(m + \mathsf{succ}(n))$, por lo que $\mathsf{cong}_{\mathsf{succ}}(\mathsf{right}{-}\mathsf{succ}\mathbb{N}(m, n))$ está en el tipo correcto.

Con estas dos funciones podemos definir

$$\begin{aligned}+{-}\mathsf{com} &: \prod_{(m:\mathbb{N})} \prod_{(n:\mathbb{N})} m + n = n + m\\ +{-}\mathsf{com}(0, n) &:\equiv \mathsf{right}{-}0_{\mathbb{N}}(n)\\ +{-}\mathsf{com}(\mathsf{succ}(m), n) &:\equiv \mathsf{trans}(\mathsf{cong}_{\mathsf{succ}}(+{-}\mathsf{com}(m, n)), \mathsf{right}{-}\mathsf{succ}_{\mathbb{N}}(n, m))\end{aligned}$$

□

Nótese que nuestra definición de $+{-}\mathsf{com}(\mathsf{succ}(m), n)$ representa de una manera compacta el siguiente razonamiento:

$$\begin{aligned}\mathsf{succ}(m) + n &\equiv \mathsf{succ}(m + n) && \text{Por definición}\\ &= \mathsf{succ}(n + m) && \text{Por inducción}\\ &= n + \mathsf{succ}(m) && \text{Por el resultado previo}\end{aligned}$$

La proposición anterior ejemplifica que los resultados se pueden utilizar para definir otros resultados, y el uso de estos de esta manera es claro y sin ninguna ambigüedad. Esto, entre otras cosas, hace la teoría de tipos dependientes una teoría ideal para la formalización por computadoras.

5. Asistentes de pruebas

Incluso grandes matemáticos como Leibniz, Gauss, Andrew Wiles, entre otros, han cometido graves errores en sus demostraciones, y ellos no fueron los primeros ni serán los últimos. A fin de evitar esto, existen los llamados **asistentes de pruebas**, programas que verifican que una demostración es correcta. La mayoría de estos usan teoría de tipos en vez de teoría de conjuntos, consideremos el siguiente ejemplo tomado de [2] para ver por qué:

Proposición 5.1. *Sean U y V espacios vectoriales y $f : U \to V$ una función lineal. Entonces $f(2x + y) = 2f(x) + f(y)$.*

La proposición es fácilmente entendida por un lector que conozca estos conceptos, y pareciese que se ha presentado con precisión los variables necesarias, pero veamos la gran cantidad de información omitida:

- Se ha entendido que existe un campo K subyacente a U y a V.
- Se ha entendido que f es en realidad una función entre los conjuntos subyacentes $f : |U| \to |V|$, recordemos que un espacio vectorial U es un triple $(|U|, \cdot, +)$.
- Se ha entendido que x y y son elementos arbitrarios de $|U|$.
- Se ha entendido que el '$+$' en la izquierda de la ecuación es la suma asociada a U, mientras que el '$+$' en la derecha es el asociado a V.
- Finalmente, se ha entendido que 2 es $1 + 1$, donde 1 es el neutro de la multiplicación del campo K.

Esta gran cantidad de información no puede ser inferida correctamente por computadoras que formalicen la teoría de conjuntos. Uno de los principales problemas es que la estructura que tiene cierto constructo, como los espacios vectoriales, no es reflejada de una manera única en teoría de conjuntos. Por poner otro ejemplo, dada la construcción de los reales como cortes de Dedekind, no solo tiene sentido la proposición '$0_{\mathbb{Q}} \in 1_{\mathbb{R}}$'; peor aún, es verdadera.

En contraste, en teoría de tipos, los elementos de un tipo pertenecen a un único tipo (a excepción de los tipos mismos), y la estructura adicional es parte esencial y única de una definición (ver 3.7, por ejemplo). Estos hechos permiten la inferencia del significado de variables no completamente especificadas, en la gran mayoría de los casos. Esto hace que teoría de tipos sea una teoría más apropiada para la formalización de las matemáticas a través de programas de computadoras.

Ya existe un gran esfuerzo a nivel global de traducir la matemática clásica al lenguaje de teoría de tipos, usado por los asistentes de pruebas, y así permitir la verificación de estos resultados por computadora. Entre estos, mencionamos las formalizaciones [8] y [12], proyectos con más de 250 contribuidores en conjunto.

Gran parte de resultados que consideramos básicos, y que se enseñan a nivel de pregrado ya están formalizados. Sin embargo, también se han verificado algunos resultados más modernos y complicados. El teorema de Feit-Thompson, que indica que todo grupo de orden finito e impar es soluble, fue verificado usando el programa Coq en el 2013 [7].

Otro ejemplo es la verificación de un resultado avanzado de geometría algebraica de Peter Scholze, ganador del premio Fields 2018, en colaboración con Dustin Clausen. El proceso de formalización comenzó como un reto de Scholze a la comunidad de asistentes de pruebas [10]:

> Considero que este teorema es de una gran importancia fundacional, por lo que estar 99.9 % seguro no es suficiente... Pasé mucho del 2019 obsesionado con la prueba de este teorema, casi volviéndome loco sobre esta. Al final logramos escribir el argumento en un artículo, pero creo que nadie más se ha atrevido a ver los detalles de este, así que todavía tengo algunas pequeñas dudas.

Solo 6 meses después, con la ayuda de varios matemáticos y científicos de la computación, Scholze escribe que su reto era prácticamente un éxito: aunque todavía no se había demostrado el teorema, todos los lemas que causaban cierta duda ya estaban formalizados [11]. A través de este ejercicio, él comenta, no solo encontró múltiples errores (que afortunadamente fueron posibles de solucionar), también profundizó el entendimiento de su propia prueba.

Esta es la propuesta de los proponentes de los asistentes de pruebas: introducir a las computadoras como una herramienta más en el arsenal que posee un matemático, y obtener una mejor comprensión del tema de estudio, así como la posibilidad de no volver a equivocarnos nunca más[3].

6. Conclusión

Hemos introducido la teoría de tipos dependientes, en donde los conceptos principales son los tipos y los elementos de tipos. Explicamos el proceso de formación de los tipos inductivos, y definimos algunos de estos, como los pares dependientes, el coproducto, las igualdades y los naturales. A modo de ejemplo, vimos como los conceptos introducidos permiten desarrolar la teoría de números naturales.

Detallamos también el uso de asistentes de pruebas para formalizar la matemática. Estos ya están siendo utilizados a nivel global para formalizar desde los resultados más básicos, hasta los últimos resultados publicados actualmente. Exhortamos al lector en ahondar en el uso de estos, resaltamos los lenguajes Lean [1], Coq [3] y Agda [4] por tener múltiples grupos de investigación activos que los utilizan.

La teoría de tipos todavía es una rama en desarrollo, y existen múltiples grupos que están traduciendo la matemática clásica a este nuevo lenguaje. Creemos que en el futuro la teoría de tipos dependientes, o una variante similar a esta, será la teoría principal sobre la cual se desarrollará las matemáticas, y la teoría de conjuntos ya no será la herramienta principal para formalizar conceptos.

[3]Podría preguntarse uno, ¿qué garantiza que el programa no cometa un error? La respuesta es: matemática. Se puede razonar de una forma similar a la de inducción, y verificar que cada paso del programa es correcto, y por lo tanto, el algoritmo entero.

Referencias bibliográficas

[1] Avigad, J., de Moura, L., y Kong, S. (2015). *Theorem proving in Lean*. Microsoft Research.

[2] Bauer, Andrej (2020). *What makes dependent type theory more suitable than set theory for proof assis tants?* Recuperado el 20 de febrero del 2023 de `https://mathoverflow.net/q/376973`.

[3] Bertot, Y., y Castéran, P. (2013). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[4] Bove, A., Dybjer, P., y Norell, U. (2009). *A Brief Overview of Agda - A Functional Language with Dependent Types*. En TPHOLs, Vol. 5674, pp. 73-78.

[5] Church, Alonzo (1932). *Set of Postulates for the Foundation of Logic*. The Annals of Mathematics 33.2, pág. 346.

[6] Enderton, H. (1977). *Elements of set theory*. New York: Academic Press, isbn: 978-0-12-238440-0.

[7] Gonthier, Georges et al. (2013). *A Machine-Checked Proof of the Odd Order Theorem*. Interactive Theorem Proving. Vol. 7998. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, págs. 163-179

[8] Mahboubi, Assia y Tassi, Enrico. (2021). *Mathematical Components*. doi: 10.5281/ZENODO.4457887.

[9] Martin-Löf, Per (1984). *Intuitionistic type theory: notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Studies in proof theory 1. Napoli: Bibliopolis.

[10] Scholze, Peter (2020). *Liquid Tensor Experiment*. Recuperado el 20 de febrero del 2023 de `https://xenaproject.wordpress.com/2020/12/05/liquid-tensor-experiment/`.

[11] Scholze, Peter (2021). *Half a year of the Liquid Tensor Experiment: Amazing developments*. Recuperado el 20 de febrero del 2023 de `https://xenaproject.wordpress.com/2021/06/05/half-a-year-of-the-liquid-tensor-experiment-amazing-developments/`.

[12] The mathlib Community (2020). *The lean mathematical library*. Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. New Orleans LA USA: ACM, págs. 367-381.

[13] The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.