Proyecto 02: El Cálculo λ como fundamento del razonamiento computacional

Giovanny Cruz Martinez
Facultad de Ciencias, UNAM
Ciudad de México, México
312001690

I. Introducción

Al inicio de este curso, veía el Cálculo λ simplemente como una forma de definir funciones de forma más sencilla. Sin embargo, la pregunta central de esta investigación me obligó a cambiar esa perspectiva: ¿Cómo puede un sistema tan simple servir de base formal para razonar sobre la corrección de programas complejos?

Mi hipótesis, basada en la lectura de Philip Wadler en su publicación Propositions as Types, es que el Cálculo λ no solo modela computación, sino que es un sistema lógico aunque no lo parezca a primera vista. A través del principio de "Proposiciones como Tipos" [1], intentaré demostrar que escribir un programa con un buen control de tipos es equivalente a demostrar un teorema matemático.

II. Contexto Histórico y Paradigmas

Para entender la magnitud del problema que se abordará, es necesario mirar atrás. En la historia de la computación, siempre han existido dos corrientes de pensamiento paralelas. Por un lado, el modelo de la Máquina de Turing, que describe la computación como una serie de cambios de estado en una cinta infinita; y por el otro, el Cálculo λ, formulado originalmente por Alonzo Church (también en 1936) para estudiar la fundamentación de las matemáticas [2].

Según la clasificación de lenguajes que estudiamos en la Nota 01 [3], la mayoría de los lenguajes más comúnmente usados en la industria (como C o Java) provienen del modelo de Turing: son imperativos y se centran en el "cómo" hacer las cosas. En cambio, el tema de esta investigación se ubica en el paradigma declarativo funcional, descendiente directo de Church. Esta diferencia es muy importante, porque, como descubrí al leer a Wadler [1], la lógica formal se lleva mucho mejor con las funciones puras que con los cambios de estado arbitrarios.

III. El Cálculo λ como Lógica de Razonamiento

Para responder a cómo este cálculo es pieza fundamental para la corrección, debemos entender que no estamos ante una simple analogía. Como vimos en la Nota 09 del curso, el Cálculo λ es un modelo de computación completo [4], pero cuando le añadimos tipos, se convierte en un sistema de prueba.

La respuesta a nuestra pregunta de investigación la podemos encontrar en la correspondencia Curry-Howard. En este marco, "razonar sobre la corrección" de un programa se reduce a verificar sus tipos. Si se logra asignar un tipo T a un término M (escrito $\Gamma \vdash M : T$ como vimos en la Nota 07 [5]), habremos demostrado matemáticamente que el programa se comportará correctamente respecto a esa especificación.

Por lo tanto, el Cálculo λ sirve como base formal porque elimina la necesidad de "adivinar" si el programa fallará. El sistema de tipos actúa como una barrera lógica porque entonces así los programas incorrectos ni siquiera serían considerables como proposiciones válidas.

IV. Un Diccionario entre dos idiomas

Para entender esto, intenté verlo como una traducción. Como vimos en clase, el Cálculo λ es capaz de representar datos complejos [4]. Pero aquí noté una diferencia importante: en las notas del curso codificábamos la lógica dentro de los términos (como los Booleanos de Church), pero el principio de "Proposiciones como Tipos" me dice que la lógica debe estar en el **tipo**, no en el valor.

Un caso que me ayudó a entenderlo fue la **implicación**. En lógica, $A \supset B$ significa que si A es verdad, entonces B también. En programación, nos podemos dar cuenta de que esto es exactamente la definición de una **función** $(A \to B)$ que estudiamos al ver abstracciones lambda [6]. Lo mismo pasa con la **conjunción** $(A \wedge B)$: para probarla necesitamos una prueba de A y una de B, lo cual en el código equivale a un "par" o tupla $(A \times B)$.

En el Cuadro I podemos ver resumidas algunas equivalencias. Básicamente, programar no es solo dar órdenes, es construir demostraciones paso a paso.

Cuadro I
La Correspondencia Curry-Howard [1]

Lógica (Proposiciones)	**Programación (Tipos)**
Proposición	Tipo
Prueba	Programa
Implicación ($A \supset B$)	Función ($A \to B$)
Conjunción ($A \wedge B$)	Producto ($A \times B$)
Disyunción ($A \vee B$)	Suma / Unión ($A + B$)
Simplificación de prueba	Evaluación de programa

V. Reducción de Sujeto

Si aceptamos que los tipos son teoremas lógicos, surge una duda que puede ser crítica: ¿Qué pasa cuando el programa se ejecuta? ¿Podría la computación "romper" la lógica?

Aquí es donde entra la propiedad que da sustento a todo el razonamiento, la Reducción de Sujeto (*Subject Reduction*). Esta propiedad establece que si un término t tiene un tipo T y se reduce a t' (un paso de cómputo), entonces t' mantiene el tipo T [7].

En el contexto de la pregunta que da origen a esta investigación, esta propiedad es la respuesta: el Cálculo λ garantiza la corrección porque la ejecución (β-reducción) preserva la verdad lógica. Si partimos de una premisa verdadera (un programa con los tipos correctos), la conclusión (el resultado) necesariamente será verdadera.

Es importante analizar qué significa formalmente esta propiedad. El teorema de preservación de tipos, tal como lo define Pierce [7], no es trivial. Establece que la relación de reducción ($\rightarrow$) y la relación de los tipos ($\vdash$) deben conmutar.

Imaginemos un diagrama: si tengo un término M y sé que tiene tipo T, al continuar por el camino de la ejecución para obtener M'. La propiedad de Reducción de Sujeto me garantiza que si intento verificar el tipo de M', obtendré nuevamente T.

Si este ciclo se rompiera, tendríamos un sistema de tipos inseguro o incorrecto. En lenguajes reales como C, esto sucede a menudo (por ejemplo, con los cast de punteros), lo que lleva a errores en tiempo de ejecución que son imposibles de detectar estáticamente. Como señala Robert Harper en su texto Practical Foundations for Programming Languages, un sistema de tipos sólido es esencial porque garantiza que un programa bien tipificado nunca puede quedarse atascado en un estado indefinido [8]. Al implementar un sistema basado en el Cálculo λ Tipificado, construimos un entorno seguro donde, por diseño, los errores de tipo en tiempo de ejecución serían matemáticamente imposibles.

VI. Verificación de la Preservación Lógica

Se realizó un programa como un ejemplo práctico de todo este tema, ya que el tema así por sí solo podría ser difícil de pasar de la abstracción de los conceptos lógicos a la práctica. El objetivo es visualizar el momento exacto en que la lógica se preserva durante la ejecución.

Para esto se implemento un intérprete en Haskell del Cálculo λ Tificado, extendiendo el `MiniLisp` que entregué en mi proyecto 1. La elección de utilizar el algoritmo de sustitución de la Nota 08 [9] y la semántica de paso pequeño de la Nota 06 [10] fue con intención: es importante poder observar cada paso intermedio de la reducción.

El programa funciona como un auditor o revisor lógico: 1. Toma un término y verifica que sea una "demostración válida" (Semántica Estática). 2. Ejecuta un paso de reducción (Semántica Dinámica). 3. Vuelve a verificar que el nuevo término siga siendo válido.

Al ejecutar el programa con términos complejos (como condicionales booleanos), podemos confirmar de primera mano que el tipo nunca cambia. Esto ilustra la respuesta a la pregunta: podemos confiar en la corrección del programa final porque el formalismo del cálculo protege la integridad lógica en cada micro-paso de la ejecución.

La implementación está organizada en tres partes, siguiendo las estructuras que utilicé en mi versión de Minilisp:

VI-A. Sintaxis Abstracta (ASA)

Siguiendo la Nota 05 [11], se definió el árbol de sintaxis (AST) en el archivo `SistemaTipos.hs`. Aquí se hizo un cambio clave respecto al MiniLisp original: modifiqué el constructor de funciones para que pida explícitamente el tipo del parámetro (algo como `Fun String Type ASA`). Sin este cambio, sería imposible rastrear la lógica dentro del código.

VI-B. Semántica Estática

La parte más importante del programa está en el módulo `InterpTipos.hs`. En donde se tiene la función `typeof`, que básicamente aplica las reglas del sistema de tipos $(T, \Gamma, \vdash)$ que vimos en la Nota 07 [5].

Este módulo funciona como un "filtro": antes de ejecutar nada, revisa que la lógica tenga sentido. Por ejemplo, si se intenta aplicar una función a un argumento que no corresponde, el programa avisa del error lógico inmediatamente. Si pasa este filtro, sabemos que el programa es una demostración válida.

VI-C. Captura de Variables

Uno de los obstáculos más interesantes a los que nos podemos enfrentar durante la implementación de la semántica dinámica es el problema de la captura de variables. Como se explica en la Nota 08 [9], la sustitución ($e[x := v]$) puede ser peligrosa si no se tiene cuidado con el alcance.

Si se sustituye una expresión que contiene una variable libre dentro de una función que tiene una variable ligada con el mismo nombre, ocurre una colisión que cambia el significado lógico del programa. Por ejemplo, si en la función $\lambda x.(\lambda y.x)$ intento sustituir x por y, obtendría $\lambda y.y$, que es la función identidad. Esto invalidaría totalmente la demostración lógica.

Para solucionar esto en el módulo `InterpTipos.hs`, se tuvo que implementar una lógica de sustitución que respete los alcances, verificando que los nombres de los parámetros no se sobrepongan. Aunque existen técnicas más eficientes como los Índices de de Bruijn, decidí mantener los nombres explícitos para que la traza de ejecución fuera legible a simple vista, priorizando la claridad.

VI-D. Semántica Dinámica

Para la parte de la ejecución (en `InterpTipos.hs`), tomé una decisión en el diseño que es importante. Aunque en la Nota 11 del curso vimos que usar máquinas abstractas es más eficiente que la sustitución directa [12], para este proyecto usamos la sustitución ($e[x := v]$).

Porque la sustitución se parece mucho más a cómo simplificamos ecuaciones o pruebas lógicas en papel (la reducción β). Al usar la semántica de paso pequeño (Small-Step) de la

Nota 06 [10], podremos ver exactamente cómo el término se va transformando paso a paso, lo cual era el objetivo de probar la teoría.

VI-E. Prueba del Concepto

Finalmente, en el script `DemoProyecto.hs` se junta todo: se toma un término, se verifica su tipo y se ejecuta paso a paso. En cada paso, se comprobará que el tipo no haya cambiado. Si podemos ver en la consola que el tipo se mantiene igual de principio a fin se confirmaría que la propiedad de *Subject Reduction* realmente funciona.

VII. Resultados y Evidencia

A continuación, se presenta la evidencia de ejecución del programa de ejemplo. En la captura de pantalla se puede observar la traza del intérprete para dos casos de prueba, una operación aritmética y una estructura condicional.

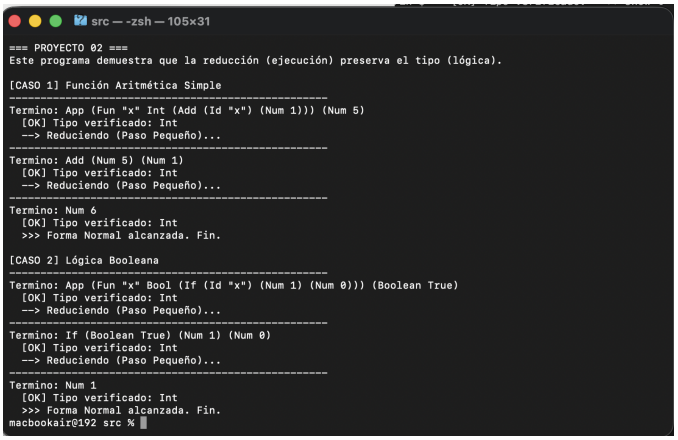

Figura 1. Ejecución del verificador de tipos y reducción paso a paso. Se observa cómo el tipo (Int) se preserva en cada iteración hasta llegar a la Forma Normal.

El hecho de que el verificador confirme el tipo `Int` en cada paso intermedio, y no solo al principio y al final, es una validación práctica de la propiedad de *Subject Reduction* dentro de nuestro modelo de Cálculo λ.

VIII. Conclusión

Este proyecto cambió mi forma de ver los lenguajes. La analogía de Wadler sobre los alienígenas me pareció que tiene mucho sentido, si enviáramos código en Java al espacio, nadie lo entendería porque es arbitrario; pero el Cálculo Lambda lo entenderían porque es lógica pura [1].

Al implementar formalmente la sintaxis y las semánticas siguiendo las notas del curso [5], [10], [11], no solo construí un intérprete, sino que pude verificar una verdad profunda: las computadoras son, en esencia, máquinas de lógica. La propiedad de reducción de sujeto me confirmó que, en un lenguaje bien diseñado, la computación es simplemente el proceso de deducir verdades automáticamente.

Referencias

[1] P. Wadler, "Propositions as types," *Communications of the ACM*, vol. 58, no. 12, pp. 75–84, 2015.

[2] A. Church, "A formulation of the simple theory of types," *The Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940.

[3] M. Soto Romero, "Nota 01: Historia de los lenguajes de programación," Facultad de Ciencias, UNAM, 2026, curso de Lenguajes de Programación.

[4] ——, "Nota 09: El cálculo λ como núcleo para funciones," Facultad de Ciencias, UNAM, 2026, curso de Lenguajes de Programación.

[5] ——, "Nota 07: Semántica estática," Facultad de Ciencias, UNAM, 2026, curso de Lenguajes de Programación.

[6] ——, "Nota 10: Expresiones lambda," Facultad de Ciencias, UNAM, 2026, curso de Lenguajes de Programación.

[7] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.

[8] R. Harper, *Practical Foundations for Programming Languages*, 2nd ed. Cambridge University Press, 2016.

[9] M. Soto Romero, "Nota 08: Expresiones let," Facultad de Ciencias, UNAM, 2026, curso de Lenguajes de Programación.

[10] ——, "Nota 06: Semántica dinámica," Facultad de Ciencias, UNAM, 2026, curso de Lenguajes de Programación.

[11] ——, "Nota 05: Sintaxis abstracta," Facultad de Ciencias, UNAM, 2026, curso de Lenguajes de Programación.

[12] ——, "Nota 11: Máquinas abstractas," Facultad de Ciencias, UNAM, 2026, curso de Lenguajes de Programación.