A gradual and general method to understand why Functional Programming is important[1]

Federico A. Sawady O'Connor, Pablo E. Martínez López, Magdalena Garzón

February 2026

Abstract. In the current context, programming continues to be taught predominantly from an imperative point of view, in which programs are conceived as series of instructions executable by computers. However, Functional Programming (FP) proposes a broader vision, which includes the previous one but expands it: programs are computable descriptions. Incorporating this vision into programmer education requires a pedagogical approach that does not impose abstraction prematurely, but builds it progressively. This article presents the pedagogical trajectory of a public Argentine university, a method that is both gradual—introducing concepts of purity progressively—and general—independent of a particular programming language—, designed to close the gap between academic training and the demands of the industry regarding the production of quality software, grounding each stage of the method in the relevant literature on programming education, functional programming, and computer science didactics. Additionally, we analyze why denotational thinking becomes especially relevant in the era of Generative AI, where the programmer's role will likely shift from writing to specifying and verifying code.

Keywords: functional programming, denotational thinking, programming education, notional machine, Gobstones.

Un método gradual y general para entender por qué la Programación Funcional es importante[2]

Federico A. Sawady O'Connor, Pablo E. Martínez López, Magdalena Garzón

Febrero 2026

Resumen. En el contexto actual, la programación sigue enseñándose predominantemente bajo un punto de vista imperativo, en el que los programas son concebidos como series de instrucciones ejecutables por computadoras. Sin embargo, la Programación Funcional (PF) propone una visión más amplia, que incluye a la anterior, pero la expande: los programas son *descripciones computables*. Incorporar esta visión en la formación de programadores requiere un enfoque pedagógico que no imponga la abstracción prematuramente, sino que la construya de forma progresiva. Este artículo presenta el trayecto pedagógico de una universidad pública argentina, un método gradual —que introduce conceptos de pureza de forma progresiva— y general —independiente de un lenguaje de programación particular—, diseñado para cerrar la brecha entre la formación académica y las demandas de la industria en torno a la producción de software de calidad, fundamentando cada etapa del método en la bibliografía relevante sobre enseñanza de la programación, programación funcional y didáctica de las ciencias de la computación. Complementariamente, se analiza por qué el pensamiento denotacional se vuelve

[1] The title references the article *Why Functional Programming Matters*, by John Hughes (Hughes, 1989).

[2] El título referencia al artículo *Why Functional Programming Matters*, de John Hughes (Hughes, 1989).

especialmente relevante en la era de la IA Generativa, donde el rol del programador probablemente se desplace de la escritura a la especificación y verificación del código.

Palabras Clave: programación funcional, pensamiento denotacional, enseñanza de la programación, máquina notacional, Gobstones.

1. Introducción

""Las formas de combinación pueden utilizar programas de nivel superior para crear otros de nivel aún mayor en un estilo que no es posible en los lenguajes convencionales"."*
— Backus, J. (1978), Communications of the ACM, Vol. 21.

Históricamente, la enseñanza de la programación ha estado dominada por una visión operacional: el programa como una secuencia de instrucciones que manipulan el estado de una máquina. Sin embargo, en el escenario contemporáneo —caracterizado por la complejidad sistémica y la irrupción de la Inteligencia Artificial Generativa (GenAI)— esta visión resulta insuficiente. Como se ha sostenido en trabajos previos sobre la base conceptual de nuestro trayecto pedagógico (Martínez López et al., 2012; Martínez López et al., 2017; Martínez López, 2021), resulta importante que el estudiante comprenda que los programas son, ante todo, *descripciones computables*: entidades cuyo significado puede analizarse independientemente de su ejecución. La programación funcional ofrece las herramientas conceptuales más directas para sostener esta visión —inmutabilidad, transparencia referencial, sistemas de tipos expresivos—, pero su nivel de abstracción la hace inadecuada como punto de partida para estudiantes principiantes.

Este artículo presenta la totalidad del método gradual y general desarrollado en una universidad pública argentina, extendiendo significativamente el trabajo fundacional de Martínez López et al. (*2012*), que se centraba exclusivamente en las características de la primera mitad de la materia introductoria. Es gradual porque introduce los conceptos que sustentan la visión denotacional de forma progresiva, partiendo de lo concreto hacia niveles crecientes de abstracción y pureza; y es general porque sus principios trascienden lenguajes específicos, enfocándose en la naturaleza misma de la computación. Nuestro objetivo es cerrar la brecha entre la formación académica y las demandas de una industria que ya no solo requiere escritores de código, sino analistas capaces de garantizar la calidad y seguridad del software producido. Como argumentó Brooks (1987), la dificultad esencial del software no reside en la escritura de código sino en la complejidad conceptual del problema que se modela. Ninguna herramienta —ni los lenguajes de alto nivel de su época, ni la IA generativa de la nuestra— elimina esa dificultad; lo que sí puede hacer la formación es equipar al programador con las herramientas mentales para abordarla.

1.1. Dos paradojas fundacionales

El punto de partida de nuestra concepción es el reconocimiento de dos paradojas que guían la enseñanza de la programación (Martínez López et al., 2012).

La primera paradoja concierne al lenguaje de programación: *"El lenguaje que utilizamos no es importante, pero es extremadamente importante."* No es importante porque debemos enfocarnos en las ideas antes que en las características únicas de un lenguaje; pero es extremadamente importante porque es el único medio para describir, conceptualizar y comunicar esas ideas, y por lo tanto debemos aprender a dominarlo adecuadamente.

Esta paradoja fundamenta la generalidad de nuestro método: los principios son independientes de cualquier lenguaje, pero su enseñanza requiere necesariamente un lenguaje cuidadosamente elegido como vehículo.

La segunda paradoja concierne a la naturaleza misma de los programas: *"Debemos entender a los programas olvidando que son entidades operacionales, pero sin olvidar que son entidades operacionales."* Los programas son entidades finalmente operacionales —deben ejecutarse de manera eficiente para obtener soluciones—, pero a su vez describen transformaciones de información y elementos abstractos como tipos de datos, que son los aspectos en los que debe concentrarse el programador al momento de concebir el programa. Podemos identificar así dos aspectos que definen a los programas: el aspecto *denotacional*, que comprende qué describe o denota el programa; y el aspecto *operacional*, que comprende los pasos que realiza el programa para cumplir una tarea. Esta paradoja fundamenta nuestra definición de los programas como *descripciones ejecutables de soluciones a problemas*: son meras descripciones (aspecto denotacional), pero deben poder ejecutarse (aspecto operacional). La resolución que proponemos es priorizar estratégicamente lo denotacional sin negar lo operacional.[3]

Dicho esto, concebir a los programas como instrucciones dirigidas a la computadora termina siendo limitante, ya que aleja nuestra atención de los elementos denotacionales que componen la descripción que queremos desarrollar. Sin embargo, como todos los programas poseen características operacionales y denotacionales, creemos que un programador debe ser capaz de entender la correspondencia entre ambos aspectos y poder tomar decisiones que privilegien a uno u otro en cada momento, sin descuidar ninguno.

El punto de partida es un cambio de paradigma ontológico (Backus, 1978). No entendemos la programación como el acto de "decirle a la computadora qué hacer" —lo cual representa una visión enteramente operacional—, sino como el acto de modelar soluciones a problemas mediante el uso de comandos (que denotan efectos) y expresiones puras (que denotan valores). Como señaló Dijkstra (1989): no es el propósito de los programas dar instrucciones a nuestras máquinas, sino que es el propósito de las máquinas ejecutar nuestros programas. El resultado es una visión de índole denotacional.

Con este enfoque declarativo, el programador se centra en *qué* debe hacerse mediante transformaciones de datos, en lugar de *cómo* hacerlo mediante secuencias de comandos. Estas dos aproximaciones requieren marcos conceptuales distintos, ya que las técnicas y soluciones estándar de ambos mundos suelen ser conceptualmente incompatibles (Joosten et al., 1993).

El artículo se organiza de la siguiente manera: la Sección 2 describe el problema de la enseñanza imperativa y sus límites; la Sección 3 analiza el impacto de la IA Generativa y por qué creemos que refuerza la necesidad del pensamiento denotacional; la Sección 4 presenta el trayecto pedagógico completo, dividido en tres cursos en secuencia; la Sección 5 discute la generalidad del método, lo contrasta con otros enfoques y aborda su relación con la industria; la Sección 6 presenta el trabajo empírico en curso; y la

[3] El término "denotacional" es más preciso que "funcional" para describir esta visión. "Funcional", en sentido estricto, alude a la utilización de funciones de orden superior, mientras que "denotacional" captura la idea más amplia de que las expresiones del programa denotan valores. No obstante, dado que "funcional" está mucho más difundido, lo usamos en el título por razones de accesibilidad, aun reconociendo su imprecisión.

Sección 7 cierra con conclusiones y líneas futuras.

2. El problema: enseñanza imperativa y sus límites

La investigación en didáctica de las Ciencias de la Computación ha identificado que una de las dificultades centrales de los estudiantes principiantes es la construcción de un modelo mental adecuado de la ejecución de programas. Du Boulay acuñó en los años 80 el concepto de *máquina notacional* (Du Boulay, 1986) para referirse al modelo mental de la computadora que el estudiante debe internalizar para comprender qué hace su código. Cuando este modelo es incompleto o erróneo —por ejemplo, cuando el estudiante cree que las instrucciones se ejecutan simultáneamente, o no distingue entre un nombre y el valor que denota—, los errores de programación se multiplican y resultan difíciles de diagnosticar. Esta dificultad se agrava en el contexto actual: si el estudiante recurre a herramientas de IA generativa antes de haber construido una máquina notacional coherente, los resultados "mágicos" que obtiene refuerzan modelos mentales incompletos, dado que el código funciona sin que el estudiante comprenda por qué (Sentance et al., 2018).

Este fenómeno se relaciona con la *tercerización cognitiva* (*cognitive offloading*, Risko y Gilbert, 2016): un estudiante que delega sistemáticamente el razonamiento sobre el código a una IA generativa no solo no corrige sus modelos mentales, sino que pierde la oportunidad de construirlos. En la práctica, estos límites se manifiestan en dificultades recurrentes: estudiantes que no distinguen entre una variable y el valor que contiene, que dependen del debugging línea por línea en lugar de razonar sobre propiedades, y que conciben la programación exclusivamente como escritura de secuencias de instrucciones.

Este vacío conceptual se entrelaza con un problema curricular. La máquina notacional que construyen los cursos imperativos tradicionales —basada en estado mutable, asignación destructiva y secuencia de instrucciones— no solo dificulta el aprendizaje inicial, sino que además deja al estudiante mal preparado para incorporar conceptos funcionales en etapas posteriores de su formación (Joosten et al., 1993; Felleisen et al., 2004). Por ejemplo, un estudiante que ha internalizado que una variable es una "caja" cuyo contenido cambia a lo largo del programa tiene dificultades para comprender la inmutabilidad: si una variable *siempre* puede cambiar, ¿qué sentido tiene una expresión que *nunca* cambia? Del mismo modo, la costumbre de razonar paso a paso sobre el estado del programa —"después de esta línea, la variable vale 3; después de la siguiente, vale 5"— dificulta el salto al razonamiento ecuacional, donde lo relevante no es *cuándo* se evalúa una expresión sino *qué* denota.

A su vez, si bien es un hecho que la programación funcional, o conceptos relativos a este paradigma, se encuentran ya presentes en la mayoría de los lenguajes de programación y tecnologías modernas (Hu et al., 2015) —desde TypeScript y Rust hasta Kotlin y Scala—, la demanda de la enseñanza de este paradigma en cursos universitarios no va en ascenso (Chapman, 2025). La concepción de los programas en muchos cursos introductorios, tanto académicos como no-académicos, sigue siendo imperativa, lo cual va en detrimento de conceptos que la programación funcional ofrece y que resultan cada vez más relevantes en la práctica profesional: inmutabilidad para evitar errores de estado, transparencia referencial para habilitar el razonamiento sobre el comportamiento del

código, y sistemas de tipos expresivos para capturar propiedades del dominio antes de la ejecución.

Una forma de resolver esta cuestión sería introducir el paradigma funcional en las primeras materias de programación. Sin embargo, esto podría ocasionar conflictos con el resto de los paradigmas que los docentes quieran presentar en cursos posteriores. De hecho, en el primer dictado de nuestro curso exploramos esta opción —utilizar directamente el paradigma funcional— y la abandonamos rápidamente pues al ponerlo en práctica comprobamos un problema que no habíamos previsto: los lenguajes funcionales, y consecuentemente las ideas que se transmiten con ellos, son fundamentalmente abstractos; además, al tratarse en general de lenguajes puros, recurren a herramientas complejas para reemplazar la noción de efecto sobre estado. Los estudiantes que ingresan a un primer curso no poseen modelos mentales abstractos formados —deben construirlos—, y para construirlos necesitan asideros concretos desde los cuales anclar las nuevas ideas. Comenzar directamente con conceptos de alta abstracción, sin un dominio concreto que ofrezca esas anclas, priva al estudiante de los puntos de apoyo necesarios para que la abstracción tenga sentido (Martínez López et al., 2012; Heckler et al., 2006). Desde la investigación sobre aprendizaje, se sabe que para lograr comprensión genuina es necesario que los estudiantes transiten de lo concreto a lo abstracto, enfrentándose a múltiples situaciones donde el objeto de aprendizaje es aplicable, de modo que puedan establecer relaciones significativas con sus esquemas de conocimiento previos (Ruiz Martín, 2020). La experiencia de nuestro primer dictado nos confirmó esto: los conceptos que decidimos impartir son necesarios *antes* de aprender un lenguaje funcional, y por lo tanto decidimos postergar ese paradigma para cursos posteriores.

Asimismo, otra opción es presentar varios paradigmas en un mismo curso. Sin embargo, la investigación educativa muestra que la profundidad supera a la amplitud en términos de resultados (Schwartz et al., 2009): un curso que intenta cubrir varios paradigmas en paralelo difícilmente alcanza la profundidad necesaria para que el estudiante comprenda cada uno de forma acabada.

Proponemos entonces una tercera vía: introducir una visión denotacional de la computación desde el inicio, sin adscribir explícitamente a ningún paradigma particular, pero utilizando herramientas conceptuales que permitan a los estudiantes desarrollar de forma gradual la capacidad de abstracción necesaria para abordar cualquier paradigma posterior (Dijkstra, 1989; Heckler et al., 2006). Concretamente, como veremos en la Sección 4, nuestro enfoque propone una máquina notacional específica dentro de un primer lenguaje de programación que posee un modelo claro de ejecución y una separación estricta entre comandos que producen efectos y expresiones que denotan valores (Reynolds, 1998),[4] diseñada para que el estudiante construya un modelo mental que le permita luego enfrentarse a lenguajes de propósito general. Esta es la estrategia que nuestra universidad ha implementado durante la última década.[5]

La fragilidad del modelo imperativo en la enseñanza inicial, donde el estudiante suele

[4] Esta distinción entre comandos y funciones que introducimos en el primer curso se alinea con el principio de separación de frases expuesto en Reynolds (1998), según el cual la claridad semántica de un lenguaje depende de que las expresiones sean entidades matemáticas puras y los comandos se encarguen de la mutación del estado.

[5] En la Sección 5 se contrasta este enfoque con las alternativas predominantes.

"adivinar" el estado de la memoria, se vuelve crítica ante la irrupción de la inteligencia artificial generativa. Si el alumno carece de una estructura formal para razonar sobre el código, la IA deja de ser un asistente para convertirse en una fuente de "código caja negra" que el estudiante es incapaz de verificar, validar o refutar.

3. La IA Generativa como catalizador

En un mundo donde la IA Generativa puede producir miles de líneas de código en segundos, el rol del programador se desplaza de la escritura a la especificación y verificación. Este desplazamiento refuerza la relevancia del pensamiento denotacional: si el programador no escribe el código sino que lo audita, necesita herramientas conceptuales para razonar sobre qué describe ese código, no solo sobre qué pasos ejecuta. Incluso bajo la tesis más radical sobre el impacto de la GenAI —la que sostiene que la programación convencional será eventualmente reemplazada por la especificación en lenguaje natural a modelos de IA (Welsh, 2023)—, la capacidad de razonar sobre *qué* debe hacer el código se vuelve la habilidad central del profesional informático, que es precisamente la capacidad que el pensamiento denotacional desarrolla. Sin embargo, a pesar de su fluidez sintáctica, los modelos de lenguaje fallan frecuentemente en el razonamiento de flujo de datos y en la comprensión de la semántica estructural del código (Song et al., 2023), introduciendo errores lógicos debido a la ausencia de una base de razonamiento formal (Wang et al., 2025).

Estudios recientes fundamentan esta preocupación: el dataset FormAI-v2 (Tihanyi et al., 2025) reveló que el 62 % de los programas en C generados por IA contienen vulnerabilidades críticas, y el reporte de CodeRabbit de 2025 (CodeRabbit, 2025) muestra que el código de IA presenta 1.7 veces más problemas de lógica y corrección en lenguajes imperativos. Es importante ser precisos: las vulnerabilidades de memoria en C son en parte un problema del lenguaje mismo, que lenguajes como Rust resuelven sin pureza funcional. Lo que estos datos demuestran es que la generación automática amplifica las consecuencias de modelos mentales incompletos.

Un ejemplo ilustra la diferencia. Consideremos una función que procesa una lista de débitos sobre un saldo con atomicidad: si un débito excede el saldo, el proceso debe detenerse sin que los débitos anteriores hayan modificado el saldo original. En un enfoque imperativo típico —y el que una IA suele producir— el código itera mutando el saldo; si el tercer débito falla, el saldo ya fue modificado, dejando el sistema inconsistente. En un enfoque funcional, la operación es una transformación pura: el saldo original nunca cambia, y la atomicidad es consecuencia directa de la inmutabilidad. El punto no es que el enfoque imperativo sea incapaz de resolver el problema, sino que requiere disciplina adicional, mientras que en el enfoque funcional la solución incorrecta es directamente inexpresable. Cuando quien escribe el código es una IA que tiende a producir el "camino feliz" sin salvaguardas, la visión denotacional entrena al programador para detectar estas ausencias por la naturaleza semántica del código.

Más aún, un programador con la base denotacional sólidamente internalizada puede aprovechar la IA generativa para delegar la escritura mecánica de código —la lectura de patrones, la generación de boilerplate, la implementación de soluciones conocidas— y concentrar su atención en lo que la herramienta no puede garantizar: que el código

cumpla con las propiedades requeridas, que los invariantes del dominio se respeten, y que los casos borde estén contemplados. La tercerización es productiva precisamente cuando quien delega tiene el criterio para evaluar y corregir el resultado; sin la base conceptual, la delegación se convierte en dependencia.

Más generalmente, la programación funcional ofrece *transparencia referencial* —si la función **f** denota un valor entero diferente de *bottom*, entonces **f(x) + f(x)** es siempre **2 * f(x)**—, que habilita el razonamiento ecuacional como herramienta de verificación. En un entorno imperativo con efectos ocultos, esta equivalencia no es garantizable en cualquier contexto, y la IA generativa frecuentemente introduce optimizaciones que asumen propiedades que el código imperativo no puede garantizar (Du et al., 2024; Cummins et al., 2021). Mediante tipos algebraicos (Bragilevsky, 2021), mónadas (Wadler, 1992) o efectos algebraicos (Plotkin y Pretnar, 2013), podemos modelar formalmente qué puede y qué no puede hacer el código incluso antes de ejecutarlo, transformando al programador en un especificador de propiedades capaz de usar sistemas de tipos (Pierce, 2002) para auditar código generado por IA. Estos conceptos ya están presentes en lenguajes como TypeScript, Rust, Scala y Kotlin; la formación denotacional que proponemos busca que los graduados puedan aprovecharlos efectivamente.

4. El trayecto pedagógico: tres cursos hacia un modelo mental denotacional

Nuestro método no es una propuesta radical y purista para un primer curso, sino una transición deliberada que comienza en lo concreto y termina en lo abstracto. Consiste en tres materias que en secuencia construyen progresivamente la capacidad de abstracción del estudiante. La metodología se centra en un proceso de transferencia desde lo concreto hacia lo abstracto: partimos desde elementos concretos y cotidianos porque permiten al estudiante activar conexiones significativas con sus conocimientos previos —un tablero, unas bolitas, la idea de que una configuración del tablero puede representar cualquier cosa que se precise, como por ejemplo un colectivo con los asientos libres u ocupados, o letras, o números—, y desde ese anclaje concreto migramos gradualmente a situaciones en donde lo que se manipula es inevitablemente reconocido como intangible y abstracto (Heckler et al., 2006; Martínez López et al., 2012).

4.1. Introducción a la Programación

El diseño de este primer curso se nutre de varias tradiciones: los esquemas algorítmicos de Scholl y Peyrin (1988), la introducción a la programación funcional de Bird y Wadler (1988), la claridad expositiva sobre programación imperativa de Kernighan y Ritchie (1988), el énfasis en el estilo y la legibilidad del código de Kernighan y Plauger (1978), y el principio de separación entre expresiones y comandos de Reynolds (1998). De estas influencias, el curso extrae la convicción de que la buena programación es, ante todo, una cuestión de claridad conceptual y no de dominio sintáctico.

A través del lenguaje Gobstones, el alumno se enfrenta a la primera gran distinción: los comandos (que producen efectos) y las funciones (que describen valores). Aquí, las variables no son “cajas” que cambian, sino nombres que denotan expresiones, y las estructuras de datos elementales que se introducen —listas y registros— son inmutables:

no se modifican *in place*, sino que se construyen nuevas a partir de las existentes. Estas decisiones siembran la base del pensamiento denotacional en un entorno que el estudiante primeramente percibe como imperativo.

Gobstones fue diseñado con intención didáctica y respondiendo a las dos paradojas fundacionales. Por un lado, es un lenguaje cuidadosamente construido para orientar el aprendizaje (resolviendo la primera paradoja: el lenguaje importa como vehículo de ideas). Por otro lado, presenta los conceptos pretendiendo generar un pensamiento denotacional a pesar de poseer sintaxis imperativa (resolviendo la segunda paradoja: priorizamos lo denotacional sin negar lo operacional).

Las decisiones de diseño del lenguaje son deliberadas y pedagógicamente motivadas. Gobstones no posee operaciones de entrada/salida, porque esta noción es fundamentalmente operacional y conlleva la modificación *in place* de estructuras. Presenta listas y no arreglos: los arreglos requieren índices (dificultando el recorrido estructural), tienen tamaño fijo, y la asignación sobre ellos tiende a interpretarse destructivamente, en conflicto con la inmutabilidad buscada. Solo posee pasaje de parámetros por valor. Y si bien la herramienta incorporó un modulador de velocidad que permite observar pasos intermedios, la concepción original prioriza la observación del efecto final —el tablero resultante—, desalentando el pensamiento operacional.

Su dominio específico concreto —un tablero con bolitas de colores, un cabezal, direcciones y operaciones primitivas— permite al estudiante desarrollar intuiciones sobre la programación sin la sobrecarga cognitiva de un lenguaje de propósito general. A diferencia de otros entornos con dominios concretos, como la tortuga de Logo (Papert, 1980; Arias y Bélanger, 1988) o los personajes de Scratch (Resnick et al., 2009), Gobstones fue diseñado para no inducir un pensamiento narrativo-secuencial ni limitar la capacidad de abstracción: mientras que la tortuga o el gato invitan al estudiante a pensar "primero hacé esto, después hacé aquello" y son semánticamente opacos —una tortuga es una tortuga, no puede representar otra cosa—, el tablero de Gobstones ofrece un mecanismo abstracto de posicionamiento (el cabezal) y bolitas que representan información. Mediante la noción de *representación*, el estudiante aprende que una configuración del tablero puede modelar conceptos del dominio del problema —por ejemplo, usar bolitas para representar asientos de un colectivo o partes de un barco o letras o cartas o cualquier otro elemento propio del dominio modelado—. Lo que importa no son las bolitas en sí, sino lo que el programador decide que denota cada configuración: una decisión de modelado intrínsecamente abstracta y denotacional, aunque al estudiante se le presenta inicialmente como concreta.

Crucialmente, la estructura del lenguaje —la separación estricta entre procedimientos y funciones, la ausencia de asignación destructiva en las expresiones— introduce las bases del pensamiento funcional de manera implícita. La división de subtareas y su expresión en la forma de procedimientos y funciones, con ausencia explícita de variables globales —si bien los comandos producen efectos sobre un estado global, ningún otro elemento del lenguaje comparte esa característica— y las construcciones programáticas que Gobstones propone, constituyen pilares de este primer curso.

Una herramienta abstracta central que se introduce en esta materia es la noción de *recorrido*: un esquema general para la preparación correcta de una repetición sobre una secuencia de elementos, que comprende inicialización, condición de corte, procesamien-

to del elemento actual, pasaje al siguiente, y finalización. Esta idea se basa en invariantes de ciclo (Mannila, 2010), *folds* (Hutton, 1999) y esquemas básicos de algoritmos (Scholl y Peyrin, 1988). En este primer curso se presenta en su forma iterativa (*while* y *foreach*), aunque en materias posteriores aparece en versiones recursivas y abstractas (recursión estructural explícita, métodos de colecciones, *folds*). La potencia del esquema se maximiza al combinarse con la división en subtareas mediante procedimientos y funciones de nombre específico, logrando recorridos simples pero poderosos donde cada línea tiene un propósito denotacional.

En una segunda parte del curso comienza un proceso de transferencia más profundo: nos desprendemos del tablero y comenzamos a resolver problemas utilizando listas y registros. Mientras los primeros recorridos son realizados sobre las celdas del tablero y pueden ser concebidos de forma concreta, los segundos se realizan enteramente sobre listas, lo cual obliga al estudiante a reconocer la transformación de información que realizan los programas sobre los datos que manipulan. Nuestra experiencia docente sugiere que este es el punto donde los estudiantes comienzan a reconocer que están trabajando con abstracciones puras —una percepción que buscaremos validar mediante los instrumentos descriptos en la Sección 6.

4.2. Estructuras de Datos

El diseño de esta materia se inspira en el enfoque de Núñez et al. (1995), que demostró la viabilidad de enseñar estructuras de datos priorizando la especificación algebraica y el razonamiento sobre propiedades antes de abordar la implementación concreta. La bibliografía clásica de estructuras de datos y algoritmos (Aho et al., 1983; Cormen et al., 2009) se utiliza como referencia para los contenidos, pero se modifica el enfoque: en lugar de partir de la implementación, se parte de la especificación de tipos abstractos de datos mediante especificaciones algebraicas (Ehrig y Mahr, 1985) y álgebra universal (Grätzer, 2008) y su implementación funcional en Haskell, apoyándonos en Bird y Wadler (1988) para los fundamentos del lenguaje y en Okasaki (1998) para el diseño de estructuras de datos puramente funcionales.[6] Partiendo de esa base, la materia actúa como un laboratorio de contraste. La primera mitad se dedica a la implementación en Haskell, priorizando el encapsulamiento, la transparencia referencial y el razonamiento sobre propiedades de las estructuras. La segunda mitad traslada estos conceptos a C/C++,[7] donde el manejo manual de memoria y punteros revela la naturaleza operacional de lo que antes era una abstracción pura.

El contraste es pedagógicamente productivo: el estudiante experimenta en primera persona qué se pierde y qué se gana al pasar de un modelo denotacional a uno operacional. Los esquemas de recorrido que en Haskell se expresan naturalmente con recursión estructural explícita deben reimplementarse manualmente en C/C++, haciendo visible la

[6] La fundamentación teórica del curso incluye elementos de teoría de punto fijo para la semántica de funciones recursivas y para los modelos de los tipos abstractos de datos (Gunter, 1992), así como nociones básicas de teoría de categorías (Pierce, 1991; Barr y Wells, 1995). Estos conceptos se utilizan como base teórica del diseño del curso, pero no se enseñan como contenido explícito.

[7] Técnicamente no aprendemos C ni C++, sino que utilizamos C++ como si fuera C, es decir, no utilizamos features avanzados de C++, como pueden ser clases, templates, etc., sino que lo utilizamos para esquivar algunas cuestiones en las que C no nos resulta práctico, como la ausencia de strings y otros elementos básicos. Esto está en consonancia con la primera paradoja ya mencionada.

diferencia entre describir *qué* se computa y especificar *cómo* se computa. Además, la noción de precondiciones e invariantes de representación introducida desde Gobstones se profundiza aquí para guiar el modelado de las estructuras de datos, garantizando propiedades que permiten obtener mayor eficiencia, cuyo impacto se verifica a través del análisis de costos de peor caso y asintótico.

4.3. Programación Funcional

Este curso se apoya en la tradición pedagógica de Bird y Wadler (1988), en los fundamentos de sistemas de tipos de Pierce (2002), y en las técnicas de programación funcional avanzada recopiladas en Jeuring y Meijer (1995). Como argumentó Hughes (1989), lo que hace valiosa a la programación funcional no son sus restricciones sino las herramientas expresivas que habilita.

Consecuentemente, en este curso se llega al paradigma puro como modelo de cómputo. Se profundiza en el estudio de sistemas de tipos fuertes, funciones puras de orden superior, la reducción de expresiones, las demostraciones por inducción y una introducción al lambda cálculo a modo de cierre.[8] El objetivo es que el alumno entienda que, en este nivel, programar y demostrar propiedades matemáticas son, en esencia, la misma actividad (Wadler, 2015).[9]

Un programador que entiende un esquema como el de *fold*, o cuestiones como la inmutabilidad, escribe código que cumple con ciertas propiedades verificables, y lo hace incluso en lenguajes que hoy son *mainstream* y multiparadigma, como TypeScript, PHP o Java. Junto a estos lenguajes encontramos problemas complejos de escalabilidad y concurrencia que no se resuelven solo con secuencias de instrucciones, sino con modelos mentales claros y herramientas abstractas que permiten manipular la información de manera más robusta.

5. Sobre la generalidad del método

Un reclamo legítimo a nuestro enfoque es el siguiente: si el método es general e independiente de lenguaje, ¿por qué depende de Gobstones, Haskell y C/C++? La respuesta se encuentra en la primera paradoja: el lenguaje no es importante —porque los principios trascienden cualquier lenguaje particular—, pero es extremadamente importante —porque es el vehículo necesario para describir y comunicar ideas. Los principios de nuestro método —la distinción comando/expresión, la visión denotacional, la construcción progresiva de la abstracción, la noción de recorrido— son independientes de cualquier lenguaje. Gobstones, Haskell y C/C++ son vehículos elegidos por sus propiedades didácticas, pero el método podría instanciarse con otros lenguajes que respeten las mismas distinciones conceptuales. Como señalamos en el artículo fundacional de nuestra propuesta: "*cualquier curso que elija como base nuestra metodología puede optar por tomar el lenguaje que construimos, o desarrollar uno propio que atienda igualmente*

[8] La fundamentación teórica del curso incluye además el tratamiento de recursión estructural de Féjer y Simovici (1991) y diversos artículos fundacionales sobre polimorfismo paramétrico y funciones de orden superior.

[9] Esta correspondencia está formalizada por el isomorfismo de Curry-Howard. Wadler (2015) ofrece una exposición accesible y argumenta que los lenguajes que emergen de esta correspondencia son *descubiertos* a partir de la lógica, no inventados por convención.

aquellos aspectos sobre la enseñanza de la programación que estimamos importantes" (Martínez López et al., 2012).

Es útil contrastar nuestra propuesta con los enfoques predominantes. La tradición imperativa clásica —C, Pascal, Java, Python— facilita la productividad inicial pero dificulta la incorporación posterior de conceptos funcionales (Joosten et al., 1993; Felleisen et al., 2004). El enfoque *objects first* genera sobrecarga cognitiva considerable al introducir simultáneamente estado, identidad, herencia y polimorfismo (Lister et al., 2004). El enfoque funcional puro desde el inicio —como *How to Design Programs* (Felleisen et al., 2018)— resuelve el problema de los modelos mentales pero enfrenta la abstracción prematura descrita en la Sección 2. Nuestra propuesta se diferencia en que no elige un paradigma como punto de partida, sino que construye una base denotacional previa a cualquier paradigma y transferible a todos ellos.

Un segundo reclamo legítimo proviene de la industria de desarrollo de producto, donde el valor se mide en funcionalidades entregadas y la demanda es de profesionales que sean productivos rápidamente con herramientas y frameworks concretos. Reconocemos que existe un costo de transición: un graduado con formación denotacional no sale de la carrera sabiendo necesariamente frameworks de frontend y backend, y necesita un período de adaptación a las herramientas específicas del entorno laboral.

No pretendemos que la profundidad conceptual y la productividad inmediata nunca estén en tensión. Sin embargo, este costo se amortiza a mediano plazo: el código producido sin razonar sobre invariantes y propiedades acumula deuda técnica más rápido (Cunningham, 1992; Avgeriou et al., 2016), y la IA generativa ha comoditizado precisamente la escritura rápida de código. Lo que no se ha comoditizado es la capacidad de detectar bugs sutiles de concurrencia, invariantes de dominio no capturadas, o casos borde no manejados —habilidades denotacionales–, justamente el tipo de problemas que son fuente principal de la mencionada deuda técnica.

Los conceptos que enseña nuestro trayecto no son ortogonales a la productividad industrial, sino previos a ella. Entender funciones puras, efectos, recorridos o precondiciones se aplica directamente al trabajar con TypeScript, React, pipelines de datos o programación reactiva. En términos educativos, se trata de *transferencia*: la capacidad de aplicar lo aprendido en situaciones nuevas (Perkins y Salomon, 1992), que es lo que acelera la adaptación a herramientas concretas.

Reconocemos, sin embargo, que la evidencia sobre la transferencia de estas habilidades al trabajo profesional de los graduados es aún limitada. Un estudio longitudinal que mida el impacto del trayecto en la práctica profesional de sus egresados constituye una línea de trabajo futura necesaria para validar empíricamente lo que hasta ahora sostenemos por argumentación teórica. La evidencia anecdótica es alentadora —muchos graduados que trabajan en la industria reportan que el enfoque recibido les resulta valioso cotidianamente, aun cuando no lo percibían así al inicio—, pero no constituye validación rigurosa, razón por la cual hemos iniciado el relevamiento descripto en la Sección 6.

6. Trabajo empírico en curso

La argumentación presentada en este artículo es de naturaleza teórica y se basa en más de una década de experiencia docente. Reconocemos que la validación empírica es necesaria

para sostener las afirmaciones sobre la transferencia del pensamiento denotacional a la práctica profesional.

Con este objetivo, hemos diseñado dos instrumentos de recolección de datos que se encuentran actualmente en proceso de aplicación. El primero es un cuestionario dirigido a estudiantes que completaron las tres materias del trayecto, orientado a capturar su percepción sobre el proceso de construcción del pensamiento abstracto, la continuidad conceptual entre los cursos, y el impacto de la formación en su capacidad de evaluar críticamente código generado por IA. El segundo es un cuestionario dirigido a graduados que se desempeñan profesionalmente en la industria, orientado a relevar la transferencia de habilidades denotacionales al entorno laboral, las diferencias percibidas respecto a colegas con otras formaciones, y el período de adaptación a herramientas y frameworks concretos.

Complementariamente, nos encontramos relevando datos cuantitativos del trayecto: tasas de aprobación por cohorte en las tres materias, y un análisis cualitativo de los errores más frecuentes en evaluaciones a lo largo de los años.

Los resultados de este trabajo empírico serán presentados en un artículo posterior, con el objetivo de contrastar la argumentación teórica aquí expuesta con evidencia proveniente de los propios actores del proceso educativo. Sin embargo, creemos que explicitar los argumentos y la bibliografía en la que se basa nuestro recorrido es un paso inicial importante para que se pueda discutir y validar desde más de un equipo de trabajo, y eso hace que nos parezca que este artículo tiene valor por sí mismo.

7. Conclusión

El método gradual y general presentado no busca que todos los graduados programen en Gobstones, Haskell o C/C++, sino que tengan la capacidad analítica de ver un programa como un objeto descriptivo sujeto a análisis riguroso. En un futuro donde la producción de código está cada vez más automatizada, esta profundidad conceptual define la verdadera calidad del software.

La importancia de la Programación Funcional radica en que permite tratar al código generado por IA como un objeto descriptivo verificable. La separación entre comandos y funciones, la noción de recorrido, el razonamiento ecuacional y la captura de propiedades mediante sistemas de tipos convierten al programador de un escritor de código en un especificador y verificador de propiedades.

Para cerrar la brecha entre academia e industria, no debemos formar operarios de sintaxis, sino profesionales que entiendan la naturaleza de la computación. La visión denotacional no es un lujo académico: es una vía concreta para recuperar el control sobre sistemas cuya complejidad ya ha superado nuestra capacidad de escritura manual, pero no nuestra capacidad de especificación formal. Como línea futura, creemos que los mismos principios pueden extenderse a materias avanzadas —concurrencia, algoritmos sobre grafos, sistemas distribuidos— que actualmente no están conectadas con el trayecto inicial.

Bibliografía

Aho, A. V., Hopcroft, J. E. y Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley. ISBN:978-0-201-00023-8. https://dl.acm.org/doi/book/10.5555/577958

Arias, F. y Bélanger, J. (1988). *Manual de programación en Logo para la enseñanza básica*. Ed.Anaya Multimedia. ISBN: 978-84-7614-178-6. https://www.grupoquorum.com/libro/ver/51696-manual-de-programacion-en-logo-para-la-ensenanza-basica.html

Avgeriou, P., Kruchten, P., Ozkaya, I. y Seaman, C. (2016). Managing technical debt in software engineering. *Dagstuhl Reports*, *6*(4), 110–138. https://doi.org/10.4230/DagRep.6.4.110

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, *21*(8), 613–641. https://doi.org/10.1145/359576.359579

Barr, M., Wells, C. (1995). *Category theory for Computing Science.* Prentice Hall. 432 pages. ISBN:978-0-13-120486-7. https://dl.acm.org/doi/10.5555/92134

Bird, R. y Wadler, P. (1988). *Introduction to functional programming*. Prentice-Hall. ISBN:978-0-13-484189-2. https://dl.acm.org/doi/10.5555/113903

Bragilevsky, V. (2021). *Haskell in depth.* Manning Publications. 664 pages. ISBN 9781617295409. https://www.manning.com/books/haskell-in-depth

Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, *20*(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

Chapman, P. (2025). Experiences of early assessment to teach functional programming. *Journal of Functional Programming*, *35*(4). Cambridge University Press. ISSN: 0956-7968. https://doi.org/10.1017/S0956796824000182

CodeRabbit. (2025). *State of AI vs human code generation report*.
https://www.coderabbit.ai/blog/state-of-ai-vs-human-code-generation-report

Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. (2009). *Introduction to algorithms* (3.ª ed.). MIT Press. 1312 pages. ISBN: 9780262533058. https://mitpress.mit.edu/9780262533058/introduction-to-algorithms/

Cummins, C., Fisches, Z.V., Ben-Nun, T., Hoefler, T., O'Boyle, M. y Leather, H. (2021). PROGRAML: A graph-based program representation for data flow analysis and compiler optimizations. *Proceedings of the 38 th International Conference on Machine Learning*, *139*:2244-2253. Marina Meila, Tong Zhang (eds.). ISSN: 1938-7228. https://proceedings.mlr.press/v139/cummins21a.html

Cunningham, W. (1992). The WyCash portfolio management system. En *Addendum to the Proceedings of OOPSLA '92* (pp. 29–30). ACM. https://doi.org/10.1145/157709.157715

Dijkstra, E. W. (1989). On the cruelty of really teaching computing science. Scholarly article EDW-1036. https://www.cs.utexas.edu/~EWD/ewd10xx/EWD1036.PDF, https://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, Ch., Peng, X. y Lou, Y. (2024). Evaluating Large Language Models in Class-Level Code Generation. *ICSE '24: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, article n.81, 1–13. IEEE. ISBN: 979-8-4007-0217-4. https://doi.org/10.1145/3597503.3639219

Ehrig, H. y Mahr, B. (1985). *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag. ISBN: 9780387137186. https://api.semanticscholar.org/CorpusID:59866408

Féjer, P. A. y Simovici, D. A. (1991). *Mathematical foundations of computer science, Volume I: Sets, relations, and induction*. Book series Monographs in Computer Science. Springer-Verlag. 425 pages. ISBN 9781461277927. https://link.springer.com/book/10.1007/978-1-4612-3086-1

Felleisen, M., Findler, R. B., Flatt, M. y Krishnamurthi, S. (2004). The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, *14*(1), 55–77. https://doi.org/10.1076/csed.14.1.55.23499

Felleisen, M., Findler, R. B., Flatt, M. y Krishnamurthi, S. (2018). *How to design programs: An introduction to programming and computing* (2.ª ed.). MIT Press. 792 pages. ISBN: 9780262534802. https://mitpress.mit.edu/9780262534802/how-to-design-programs/

Grätzer, G. (2008). *Universal algebra* (2.ª ed.). Springer-Verlag. 583 pages. ISBN: 978-0-387-77486-2. https://link.springer.com/book/10.1007/978-0-387-77487-9

Gunter, C. A. (1992). *Semantics of Programming Languages: Structures and Techniques*. MIT Press. 441 pages. ISBN: 9780262570954. https://mitpress.mit.edu/9780262570954/semantics-of-programming-languages/

Heckler, A. F., Kaminski, J. A. y Sloutsky, V. M. (2006). Do children need concrete instantiations to learn an abstract concept? En R. Sun y N. Miyake (Eds.), *Proceedings of the 28th Annual Conference of the Cognitive Science Society*, *(28)*:411–416. Psychology Press. ISBN: 978-0805862966. https://escholarship.org/uc/item/5dr0z639

Hu, Z., Hughes, J. y Wang, M. (2015). How functional programming mattered. *National Science Review*, *2*(3), 349–370. https://doi.org/10.1093/nsr/nwv042

Hughes, J. (1989). Why functional programming matters. *The Computer Journal*, *32*(2), 98–107. https://doi.org/10.1093/comjnl/32.2.98

Hutton, G. (1999). A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, *9*(4), 355–372. https://doi.org/10.1017/S0956796899003500

Jeuring, J. y Meijer, E. (Eds.). (1995). *Advanced functional programming: First international spring school on advanced functional programming techniques* (LNCS, Vol. 925). Springer. https://doi.org/10.1007/3-540-59451-5

Joosten, S., Van Den Berg, K. y Van Der Hoeven, G. (1993). Teaching functional programming to first-year students. *Journal of Functional Programming*, *3*(1), 49–65. https://doi.org/10.1017/S0956796800000599

Kernighan, B. W. y Plauger, P. J. (1978). *The elements of programming style* (2.ª ed.). McGraw-Hill. 181 pages. ISBN: 0-07-034207-5. https://granatensis.ugr.es/discovery/fulldisplay?vid=34CBUA_UGR:VU1&docid=alma991007107019704990

Kernighan, B. W. y Ritchie, D. M. (1988). *The C programming language* (2.ª ed.). Prentice-Hall. 274 pages. ISBN 0-13-

110362-8. https://books.google.com.ar/books/about/The_C_Programming_Language.html?id=FGkPBQAAQBAJ

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. y Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119–150. https://doi.org/10.1145/1041624.1041673

Mannila, L. (2010). Invariant based programming in education: An analysis of student difficulties. *Informatics in Education*, *9*(1), 115–132. https://doi.org/10.15388/infedu.2010.07

Martínez López, P. E., Bonelli, E. A. y Sawady O'Connor, F. A. (2012). El nombre verdadero de la programación: una concepción de enseñanza de la programación para la sociedad de la información. En *Actas del 10° Simposio sobre la Sociedad de la Información (SSI), 41 JAIIO*, pages 1–23. SADIO.

Martínez López, P. E., Ciolek D., Arévalo G. y Pari D. (2017). The Gobstones method for teaching computer programming. *XXV Simposio de Educación Superior en Computación (SIESC'17), dentro de la XLIII Conferencia Latinoamericana de Informática (CLEI'17)*, Córdoba. IEEE, ISBN 978-1-5386-3057-0

Martínez López, P. E. (2021). Programar es mucho más que secuenciar comandos. *Actas de las 1eras Jornadas Argentinas de Didáctica de Ciencias de la Computación (JADiCC'21)*, pages 191-204. Boari et al. (eds.), Argentina. Libro digital, PDF. ISBN 978-987-27416-9-3. https://jadicc.program.ar/wp-content/uploads/2022/03/actas-jadicc-2021.pdf

Núñez, M., Palao, P. y Peña, R. (1995). A second year course on data structures based on functional programming. En P. H. Hartel y R. Plasmeijer (Eds.), *Functional Programming Languages in Education. FPLE 1995* (LNCS, Vol. 1022). Springer. https://doi.org/10.1007/3-540-60675-0_39

Okasaki, C. (1998). *Purely functional data structures*. Cambridge University Press. ISBN: 9780511530104. DOI: https://doi.org/10.1017/CBO9780511530104

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* Basic Books. 244 pages. ISBN:978-0-465-04627-0. https://dl.acm.org/doi/10.5555/1095592

Perkins, D. N. y Salomon, G. (1992). Transfer of learning. En T. Husén y T. N. Postlethwaite (Eds.), *International encyclopedia of education* (2.ª ed.). Pergamon Press. https://www.researchgate.net/publication/2402396_Transfer_Of_Learning

Pierce, B. C. (1991). *Basic Category Theory for Computer Scientists*. MIT Press. 116 pages. ISBN: 9780262660716. https://mitpress.mit.edu/9780262660716/basic-category-theory-for-computer-scientists/

Pierce, B. C. (2002). *Types and programming languages*. MIT Press. 648 pages. ISBN: 9780262162098. https://mitpress.mit.edu/9780262162098/types-and-programming-languages/

Plotkin, G. D. y Pretnar, M. (2013). Handling algebraic effects. *Logical Methods in Computer Science*, *9*(4), 1–36. https://doi.org/10.2168/LMCS-9(4:23)2013

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J.,Silverman, B. and Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM*, 52(11):60–67. ACM. ISSN:0001-0782. https://doi.org/10.1145/1592761.1592779

Reynolds, J. C. (1998). *Theories of programming languages*. Cambridge University Press. 500 pages. ISBN: 978-0521594141. https://dl.acm.org/doi/10.5555/307999

Risko, E. F. y Gilbert, S. J. (2016). Cognitive offloading. *Trends in Cognitive Sciences*, *20*(9), 676–688. https://doi.org/10.1016/j.tics.2016.07.002

Ruiz Martín, H. (2020). *¿Cómo aprendemos? Una aproximación científica al aprendizaje y la enseñanza*. (2da ed.) Editorial Graó. Colección Educación basada en evidencias, 1. ISBN: 978-84-18058-05-9. https://dialnet.unirioja.es/servlet/libro?codigo=833023

Scholl, P. C. y Peyrin, J. P. (1988). *Schémas algorithmiques fondamentaux: séquences et itération*. Masson. ISBN: 978-2-225-81656-7.
https://etoilesvagabondes.fr/livre/1420679-schemas-algorithmiques-fondamentaux-sequences-et-iteration-pierre-claude-scholl-jean-pierre-peyrin-masson

Schwartz, M. S., Sadler, P. M., Sonnert, G. y Tai, R. H. (2009). Depth versus breadth: How content coverage in high school science courses relates to later success in college science coursework. *Science Education*, *93*(5), 798–826. https://doi.org/10.1002/sce.20328

Sentance, S., Barendsen, E. y Schulte, C. (Eds.). (2018). *Computer science education: Perspectives on teaching and learning in school*. Bloomsbury Academic. 247 pages. ISBN: 978 1 350 057104.
https://www.bloomsbury.com/us/computer-science-education-9781350296909/

Song, D., Zhou, Z., Wang, Z., Huang, Y., Chen, S., Kou, B., Ma, L. y Zhang, T. (2023). An empirical study of code generation errors made by large language models. *7th Annual Symposium on Machine Programming*. https://mapsworkshop.github.io/assets/LLM_Code_Error_Analysis_MAPS2023_camera-ready.pdf, https://mapsworkshop.github.io/, https://2023.esec-fse.org/track/fse-2023-workshops

Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R. y Cordeiro, L. C. (2025). How secure is AI-generated code: A large-scale comparison of large language models. *Empirical Software Engineering*, *30*(2), Artículo 47. https://doi.org/10.1007/s10664-024-10590-1

Wadler, P. (1992). The essence of functional programming. En *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)* (pp. 1–14). ACM. https://doi.org/10.1145/143165.143169

Wadler, P. (2015). Propositions as types. *Communications of the ACM*, *58*(12), 75–84. https://doi.org/10.1145/2699407

Wang, Z., Zhou, Z., Song, D., Huang, Y., Chen, S., Ma, L. y Zhang, T. (2025). Towards understanding the characteristics of code generation errors made by large language models. En *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*. pp. 2587–2599). IEEE. ISBN: 979-8-3315-0569-1. https://doi.org/10.1109/ICSE55347.2025.00180

Welsh, M. (2023). The end of programming. *Communications of the ACM*, *66*(1), 34–35. https://doi.org/10.1145/3570220