Editorial CIMTED

El arte de la **programación** funcional:



Agda

Guía de aprendizaje en Agda

Luis Eduardo Muñoz Guerrero

Universidad Tecnológica de Pereira

2024

Página legal

Título: El arte de la programación funcional: Guía de aprendizaje en AGDA

ISBN: 978-628-96400-2-1 para versión ePub

ISBN: 978-628-96400-3-8 para versión en PDF

Sello editorial: Corporación Centro Internacional de Marketing Territorial para la Educación y el Desarrollo (978-628-96400)

Materia: Programación

Tipo de contenido: Científico y Técnico

Clasificación THEMA: Programación funcional

Público objetivo: Enseñanza universitaria o superior

Idioma: Español / Castellano

Ciudad de Edición: Medellín

Departamento: Antioquia

Fecha de aparición: 2024-10-16

Tipo de soporte: Libro digital descargable

Formato: Epub (.epub)

Tipos de acceso: Digital: descarga y online

Tipo de contenido: Texto (legible a simple vista)

Agente editor: Corporación Centro Internacional de Marketing Territorial para la Educación y el Desarrollo

Número de identificación tributaria o cédula de ciudadanía: 8110433950

e-mail: editorialcimted@gmail.com

Teléfono: 3245664447

Autor: Luis Eduardo Muñoz Guerrero

Cuidado de la edición:
Juliana Escobar Gómez
Calle 41 no 80b 120 int 301
Código postal: 050031 - Medellín - Colombia
Octubre 17 de 2024
© Derechos reservados

© Prohibida la reproducción parcial o total sin la previa autorización del sello editorial Centro Internacional de Marketing Territorial para la Educación y el desarrollo . Octubre 17 de 2024

Las opiniones expresadas y el contenido de este libro son exclusivamente responsabilidad del autor y no indican, necesariamente el punto de vista de la Corporación CIMTED

Todo el contenido de este libro esta protegido por la ley según los derechos materiales e intelectuales del editor y del autor que escribió este libro, por lo tanto, no esta permitido copiar, fragmentar con propósitos comerciales todo su contenido, sin la respectiva autorización de los anteriores.
Si se hace como servicio académico o investigativo debe contar igualmente con el permiso escrito del autor y citar sus respectivas fuentes
Más información: editorialcimted@gmail.com

Publicación electrónica, editado en Colombia



Página legal 2
Sobre el autor
Prólogo
Introducción al Cálculo Lambda 1
Fundamentos del Cálculo Lambda....1
Evaluación y Reducción 2
Funciones y Recursión....3
Codificación de Datos 3
Relación con la Programación Funcional....4
Capítulo 1: Introducción
1. Conceptos centrales 8
1.1.1. Funciones puras....8
1.1.2. Inmutabilidad 10
1.2. Características principales 12
1.2.1. Concisión 12
1.2.2. Modularidad 13
1.2.3. Facilidad de pruebas y razonamiento 14
1.3. Técnicas de implementación....16
1.3.1. Recursión 16
1.3.2. Funciones de orden superior 17
1.3.3. Currying....18
Capítulo 2: Conceptos Básicos de Agda
2.1. Tipos de datos....21
2.1.1. Tipos de datos básicos....22
2.1.2. Tipos algebraicos de datos 23
2.2. Funciones....26
2.2.1. Definición de funciones 26
2.2.2. Pattern Matching 28
2.2.3. Aplicación de Funciones 30
2.3. Argumentos y Parámetros....31
2.3.1. Cuantificación Universal....31

2.3.2. Argumentos Implícitos 32
2.3.3. Argumentos sin Usar 33
2.3.4. Parámetros libres 34
2.4. Visibilidad de argumentos 35
2.5. Funciones anónimas 36
2.6. Documentación 37
2.6.1. Comentarios 37
2.6.2. Documentación estructurada 38
Capítulo 3: Estructuras de Datos Funcionales en Agda
3.1. Introducción 41
3.2. Listas: Los Pilares de la Programación Funcional 43
3.2.1. Funciones Comunes sobre Listas 43
3.2.2. Listas y Recursión 46
3.2.3. Eficiencia y Evaluación 47
3.3. Mapas Asociativos 49
3.3.1. Uso de mapas 50
3.3.2. Eficiencia y alternativas 51
3.4. Árboles 53
3.4.1. Árboles binarios 53
3.4.2. Árboles genéricos 56
3.4.3. Árboles de búsqueda binaria 56
3.5. Abstracciones algebraicas 58
3.5.1. Monoides 58
3.5.2. Functores 60
3.5.3. Funtores Aplicativos 61
Capítulo 4: Sistemas de tipos avanzados
4.1. Introducción 64
4.2. Tipos dependientes 65
4.2.1. Tipos indexados 66
4.2.2. Proposiciones como tipos 67
4.2.3. Ejemplos prácticos 68

4.3. Tipos coinductivos e infinitos....70
4.3.1. Flujos....70
4.3.2. Procesos e interacciones....71
4.4. Inferencia de tipos....73
4.4.1. Argumentos implícitos....74
4.4.2. Resolución de restricciones....74
4.4.3. Inferencia bidireccional....75
4.4.4. Patrones de inferencia de tipos....76
4.5. Polimorfismo....78
4.5.1. Polimorfismo paramétrico....78
4.5.2. Polimorfismo ad-hoc....79
4.5.3. Clases de tipos....80
4.6. Relevancia de argumentos....81
4.6.1. Argumentos irrelevantes....82
4.6.2. Aplicaciones en estructuras de datos....82
4.6.3. Relevancia y eficiencia....83

Capítulo 5: Efectos y concurrencia en Agda

5.1. Introducción....86
5.2. Interacción con el exterior....87
5.2.1. Manejo de efectos con IO....87
5.2.2. Interacción con E/S....89
5.3. Manejo de errores....92
5.3.1. Manejo de excepciones....92
5.3.2. Propagación de errores....94
5.4. Estado....96
5.4.1. Estado y mutabilidad....96
5.4.2. Argumentos irrelevantes....98

Capítulo 6: Mónadas en Agda

6.1. Introducción....101
6.2. Fundamentos....102
6.2.1. Definición Formal de Mónadas....102

6.2.2. Explicación Intuitiva de Mónadas 104
6.3. Mónadas en Agda 105
6.3.1. La interfaz Monad en la librería estándar de Agda 105
6.3.2. Creación de nuevas instancias monádicas 107
6.4. Composición y transformación de mónadas 110
6.4.1. Composición de mónadas 110
6.4.2. Transformadores de mónada 112
Capítulo 7: Verificación y pruebas en Agda
7.1. Introducción 115
7.2. Especificando propiedades 118
7.3. Pruebas de propiedades 123
7.4. Pruebas Formales 126
7.5. Tipos refinados 127
7.6. Extracción de código verificado 130
Capítulo 8: Temas avanzados en Agda
8.1. Introducción 135
8.2. Abstracciones avanzadas 136
8.2.1. Mónadas 136
8.2.2. Teoría de Categorías 141
8.3. Análisis y Compilación 142
8.3.1. Parsing 142
8.3.2. Compilación 145
8.4. Metaprogramación 148
8.4.1. Macros y templating 150
8.4.2. Generación de Código 155
8.5. Criptografía 159
Glosario de Términos
Bibliografía
Agradecimientos Finales 171

Imagen 1 – Función pura vs impura. (Fuente: Nikhil Sukhani)

Imagen 2 - Beneficios de la Inmutabilidad de Datos

Imagen 3 - Funciones de Orden Superior.

Imagen 5 - Estructuras de Datos Funcionales. (Fuente: Propia)

Imagen 6 - Operaciones Comunes sobre Listas. (Fuente: Propia)

Imagen 7 - Dividir y Conquistar: La Estrategia Recursiva. (Fuente: Khan Academy)

Imagen 9 – Ejemplo árbol en Agda. (Fuente: Propia)

Imagen 10 - Árbol Binario de Búsqueda. (Fuente: WikiMedia Commons)

Imagen 11 - Especificaciones como Tipos en Agda.

Imagen 12 - Composición de Mónadas.

Imagen 13 - Transformadores de Mónada Comunes. (Fuente: Propi

Imagen 14 - Interactuando con el Asistente de Pruebas de Agda. (Fuente: Propia)

Imagen 15 - Metaprogramación: Programas que Manipulan Programas. (Fuente: Propia)

Imagen 16 - El Isomorfismo de Curry-Howard: Unificando Pruebas y Programas. (Fuente: Propia)

Imagen 17- Tipos Dependientes.

Imagen 18 - Especificaciones como Tipos en Agda.

Imagen 19 - Proceso de Prueba Interactiva.

Imagen 20 - Ventajas de la Verificación Formal. (Fuente: Propia)

Imagen 21 - Extracción de Código Verificado.

Imagen 22 - Flujo de Compilación: Del Código Fuente al Ejecutable. (Fuente: Propia)

Imagen 23 - Tipos Singleton: Representando Valores a Nivel de Tipo. (Fuente: Cecilio Álvarez Caules)

Imagen 24 – Ventajas de la Generación de Código. (Fuente: Propia)

Imagen 25 - Esquema de Cifrado Autenticado con Datos Asociados (AEAD). (Fuente: Nimantha Fernando)

Imagen 26 - Ataque de Texto Plano Escogido en Criptografía. (Fuente: B. Bazeer Ahamed)

Imagen 27 - Protocolo de Intercambio de Claves Diffie-Hellman. (Fuente: Alejandro Mantilla)

© Luis Eduardo Muñoz

Sobre el autor



Luis Eduardo Muñoz Guerrero

Universidad Tecnológica de Pereira
Colombia

PhD en Ciencias de la Educación Rude Colombia cade UTP,
Magister en Ingeniera de Sistemas por la Universidad Nacional de Colombia. Su experiencia de trabajo ha girado, principalmente, alrededor del campo educativo, sus proyectos están asociados con áreas de evaluación educativa, educación basada en competencias, software educativo, enseñanza de la programación. Ha publicado artículos en revistas nacionales e internaciones. Autor de los libros; Programación Moderna con aplicaciones y Programación Funcional con Racket. Actualmente es profesor titular de tiempo completo programa de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira y Pertenece al grupo de investigación informática.

Correspondencia: lemunzg@utp.edu.co

Prólogo

Estimados lectores,

Es para mí un gran placer presentarles este libro sobre programación funcional en Agda. Como apasionado de los lenguajes de programación y las matemáticas, he encontrado en Agda una herramienta fascinante que combina la elegancia de la programación funcional con el rigor de la demostración formal de teoremas.

A lo largo de las siguientes páginas, mi objetivo es guiarlos en un emocionante viaje a través de los conceptos fundamentales y avanzados de la programación en Agda. Comenzaremos con los bloques básicos del paradigma funcional, como funciones puras, inmutabilidad y recursión. Gradualmente construiremos sobre estos cimientos para explorar características más sofisticadas como tipos dependientes, co-inducción y mónadas.

Pero este libro no es solo sobre aprender un nuevo lenguaje de programación. Es sobre adoptar una mentalidad diferente, una en la que el código y las pruebas son uno solo, donde las especificaciones precisas guían el diseño, y donde la corrección se garantiza por construcción. Espero transmitirles la belleza y el poder de este enfoque.

Ya sea que sean estudiantes buscando expandir su repertorio de programación, ingenieros de software interesados en técnicas de verificación formal o simplemente curiosos explorando nuevas fronteras computacionales, confío en que encontrarán este libro iluminador y gratificante.

Emprendamos juntos este viaje hacia el fascinante mundo de la programación funcional en Agda. ¡Que la elegancia de las matemáticas y la precisión de los tipos sean nuestra guía!

Luis Eduardo Muñoz Guerrero

Introducción al Cálculo Lambda

El cálculo lambda, desarrollado por Alonzo Church en la década de 1930, es un sistema formal que sienta las bases teóricas de la programación funcional. Es un modelo de computación universal basado en la abstracción y aplicación de funciones, que nos permite estudiar la computación de manera pura y abstracta, sin preocuparnos por los detalles de implementación de bajo nivel.

El cálculo lambda es un sistema elegante y minimalista que captura la esencia de la computación. A pesar de su simplicidad sintáctica, es sorprendentemente poderoso y expresivo. Con solo tres componentes básicos (variables, abstracciones y aplicaciones), el cálculo lambda puede representar cualquier función computable.

Fundamentos del Cálculo Lambda

El cálculo lambda se basa en tres componentes principales:

1. **Variables**: Son símbolos que representan los argumentos de las funciones. Las variables pueden ser nombradas con cualquier símbolo, como `x`, `y`, `z`, etc. Las variables son esenciales para la abstracción y la parametrización de funciones.

2. **Abstracciones**: Definen funciones anónimas, es decir, funciones sin un nombre específico. Las abstracciones son el núcleo del cálculo lambda y representan la idea de una función como una "receta" o un "proceso" que toma un argumento y produce un resultado. Se denotan con el símbolo `λ` seguido de una variable y un punto `.`. Por ejemplo, `λx.x` representa la función identidad, que simplemente devuelve su argumento sin cambiarlo.

3. **Aplicaciones**: Representan la aplicación de una función a un argumento. Las aplicaciones permiten "ejecutar" o "evaluar" una función con un valor específico. Se escriben como `(f x)`, donde `f` es una función (abstracta o concreta) y `x` es el argumento que se pasa a la función. La aplicación es el mecanismo por el cual las funciones son invocadas y los resultados son calculados.

La sintaxis del cálculo lambda es muy simple y consistente. Una expresión lambda tiene la siguiente forma general:

```
λx.M
```

Donde `x` es la variable que representa el argumento de la función, y `M` es el cuerpo de la función, que puede contener otras expresiones lambda, variables y aplicaciones.

Componente	Descripción	Ejemplo
Variable	Representa un argumento de función	`x`, `y`
Abstracción	Define una función anónima	`λx.x`
Aplicación	Aplica una función a un argumento	`(f x)`

Evaluación y Reducción

El proceso de evaluación en el cálculo lambda se basa en la reducción de expresiones. La reducción es el mecanismo por el cual las expresiones lambda se simplifican y se evalúan hasta obtener un resultado final. Existen dos reglas principales de reducción:

1. **α-conversión**: También conocida como "renombramiento de variables", la α-conversión permite cambiar el nombre de las variables ligadas en una expresión lambda sin alterar su significado. Por ejemplo, `λx.x` y `λy.y` son equivalentes por α-conversión, ya que ambas representan la función identidad, independientemente del nombre de la variable utilizada.

2. **β-reducción**: Es la regla fundamental de evaluación en el cálculo lambda. La β-reducción se aplica cuando una función abstracta es aplicada a un argumento concreto. Consiste en sustituir todas las ocurrencias de la variable ligada en el cuerpo de la función por el valor del argumento. Por ejemplo, la expresión `(λx.x + 1) 2` se reduce a `2 + 1` mediante β-reducción, ya que la variable `x` en el cuerpo de la función se reemplaza por el argumento `2`.

La β-reducción es el motor que impulsa la evaluación en el cálculo lambda. Permite simplificar expresiones complejas paso a paso hasta obtener un resultado final. El proceso de β-reducción se puede aplicar repetidamente hasta que no haya más aplicaciones de funciones que reducir, momento en el cual se dice que la expresión está en "forma normal".

Regla de reducción	Descripción	Ejemplo
α-conversión	Renombramiento de variables ligadas	`λx.x ≡ λy.y`
β-reducción	Aplicación de una función a un argumento	`(λx.x + 1) 2 → 2 + 1`
η-conversión	Conversión de una función a su forma punto-libre	`λx.f x ≡ f`

$$
\begin{aligned}
& (\lambda f.\lambda g.\lambda h.fg(h\,h))(\lambda x.\lambda y.x)h(\lambda x.x\,x) \\
\rightarrow_\beta \quad & (\lambda g.\lambda h.(\lambda x.\lambda y.x)g(h\,h))h(\lambda x.x\,x) && (1) \\
\rightarrow_\alpha \quad & (\lambda g.\lambda k.(\lambda x.\lambda y.x)g(k\,k))h(\lambda x.x\,x) && (2) \\
\rightarrow_\beta \quad & (\lambda k.(\lambda x.\lambda y.x)h(k\,k))(\lambda x.x\,x) && (3) \\
\rightarrow_\beta \quad & (\lambda x.\lambda y.x)h((\lambda x.x\,x)\,(\lambda x.x\,x)) && (4) \\
\rightarrow_\beta \quad & (\lambda y.h)((\lambda x.x\,x)\,(\lambda x.x\,x)) && (5) \\
\rightarrow_\beta \quad & h && (6)
\end{aligned}
$$

Ilustración 1 – Ejemplo visual de la redacción beta.
(Fuente: n0unc3 – Stack Exchange)

Funciones y Recursión

Una de las características más poderosas del cálculo lambda es su capacidad para definir y manipular funciones de orden superior. Las funciones de orden superior son aquellas que pueden tomar otras funciones como argumentos o devolver funciones como resultados. Esto permite una gran flexibilidad y modularidad en la construcción de programas.

Por ejemplo, la siguiente expresión lambda define una función que toma una función `f` y un argumento `x`, y aplica `f` a `x`:

```
λf.λx.f x
```

Esta función, conocida como "aplicación de funciones", es un ejemplo simple pero fundamental de una función de orden superior. Puede ser utilizada para aplicar cualquier función a un argumento dado.

La recursión (Técnica donde una función se llama a sí misma para resolver problemas) es otro concepto importante en el cálculo lambda. A diferencia de otros modelos de computación, el cálculo lambda no tiene estructuras de control de flujo explícitas como bucles o instrucciones de recursión. En su lugar, la recursión se logra mediante el uso de combinadores de punto fijo, como el combinador Y:

```
Y = λf.(λx.f (x x)) (λx.f (x x))
```

El combinador Y permite definir funciones recursivas sin necesidad de nombrarlas explícitamente. Funciona mediante la aplicación repetida de una función a sí misma, lo que permite la recursión sin referencias directas a la función en sí.

Combinador	Definición	Descripción
Identidad (I)	`λx.x`	Devuelve su argumento sin cambios
Constante (K)	`λx.λy.x`	Devuelve siempre su primer argumento
Aplicación (S)	`λf.λg.λx.f x (g x)`	Aplica el resultado de una función a otra función
Punto fijo (Y)	`λf.(λx.f (x x)) (λx.f (x x))`	Permite definir funciones recursivas

Codificación de Datos

A pesar de que el cálculo lambda es un sistema puro y minimalista, sin tipos de datos primitivos incorporados, es posible codificar estructuras de datos complejas utilizando solo funciones y abstracción.

Uno de los ejemplos más conocidos es la codificación de Church de los números naturales. En esta codificación, cada número natural se representa como una función que toma una función `f` y un valor inicial `x`, y aplica `f` a `x` tantas veces como lo indique el número.

Número	Codificación de Church
0	`λf.λx.x`
1	`λf.λx.f x`
2	`λf.λx.f (f x)`

3	`λf.λx.f (f (f x))`

Por ejemplo, el número 2 se codifica como `λf.λx.f (f x)`, que toma una función `f` y un valor `x`, y aplica `f` dos veces a `x`.

De manera similar, otros tipos de datos, como listas, pares, árboles, etc., pueden codificarse utilizando funciones y combinadores adecuados. Esto demuestra la expresividad y flexibilidad del cálculo lambda para representar estructuras de datos complejas.

Relación con la Programación Funcional

El cálculo lambda es la base teórica de los lenguajes de programación funcional modernos, como Haskell, ML, Lisp y, por supuesto, Agda. Los conceptos y principios fundamentales del cálculo lambda se reflejan directamente en las características y patrones de diseño de estos lenguajes.

Algunas de las conexiones clave entre el cálculo lambda y la programación funcional incluyen:

1. **Funciones como ciudadanos de primera clase**: En los lenguajes funcionales, las funciones son tratadas como valores de primera clase, lo que significa que pueden ser pasadas como argumentos, devueltas como resultados y asignadas a variables. Esto es una consecuencia directa de la naturaleza del cálculo lambda, donde las funciones son los bloques de construcción fundamentales.

2. **Inmutabilidad y transparencia referencial**: El cálculo lambda es un sistema puro y sin efectos secundarios. Las funciones siempre producen el mismo resultado dado el mismo argumento, lo que se conoce como transparencia referencial. Esto se refleja en los lenguajes funcionales, donde se promueve la inmutabilidad y se evitan los efectos secundarios, lo que facilita el razonamiento sobre el comportamiento de los programas.

3. **Evaluación perezosa**: Algunos lenguajes funcionales, como Haskell, utilizan la evaluación perezosa por defecto. Esto significa que las expresiones no se evalúan hasta que su valor sea realmente necesario. La evaluación perezosa tiene sus raíces en el cálculo lambda, donde la reducción se aplica solo cuando es necesario para obtener un resultado.

4. **Recursión y combinadores**: La recursión es un patrón fundamental en la programación funcional, y se utiliza ampliamente para definir y procesar estructuras de datos recursivas. Los combinadores, como el combinador Y en el cálculo lambda, proporcionan una forma elegante de definir funciones recursivas sin nombrarlas explícitamente.

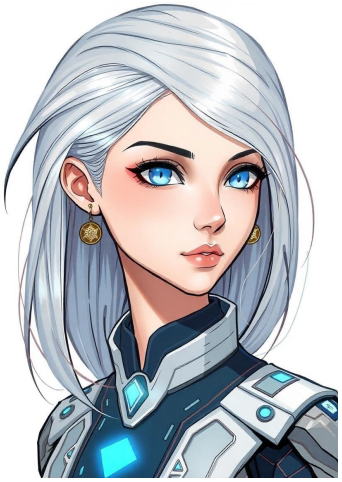
Concepto en el cálculo lambda	Equivalente en lenguajes funcionales
Función anónima	Lambda, función sin nombre
Aplicación de función	Llamada a función
Variable ligada	Parámetro de función
Variable libre	Variable global o no definida
Abstracción	Definición de función
Reducción	Evaluación de expresiones

Comprender los principios y conceptos del cálculo lambda proporciona una base sólida para razonar sobre la programación funcional y aprovechar al máximo lenguajes como Agda. El cálculo lambda no solo fundamenta la teoría detrás de estos lenguajes, sino que también influye en su diseño, sintaxis y patrones de programación.

El cálculo lambda es una herramienta poderosa y elegante para estudiar y comprender la computación desde una perspectiva funcional. Su simplicidad y expresividad lo convierten en un fundamento esencial para cualquier persona interesada en la programación funcional y en el razonamiento abstracto sobre la computación. Al dominar los conceptos del cálculo lambda, los programadores pueden adquirir una comprensión más profunda de los lenguajes funcionales y aprovechar al máximo su potencial.

Para este recorrido de aprendizaje nos acompaña el personaje ADA

Nombre: Ada



Descripción Física: Ada es una figura etérea con una apariencia futurista. Tiene un cabello largo y plateado que brilla con un suave resplandor, y sus ojos son de un azul profundo que transmite sabiduría y serenidad. Su vestimenta es una mezcla de túnicas tradicionales y elementos tecnológicos, con patrones geométricos que se iluminan sutilmente cuando habla o interactúa con el lector. Siempre lleva consigo un objeto antiguo: una pequeña réplica de la máquina analítica de Charles Babbage, que simboliza su conexión con la historia de la programación.

Personalidad: Ada es paciente, sabia y siempre dispuesta a ayudar. Tiene un aire de calma y confianza que pone a los lectores a gusto, independientemente de su nivel de conocimiento sobre Agda. Es curiosa y apasionada por el aprendizaje, lo que la hace una guía perfecta para explorar los conceptos complejos del lenguaje. Ada también es empática y sabe cómo motivar a los lectores, compartiendo historias inspiradoras y anécdotas históricas.

Habilidades: Ada tiene la capacidad de visualizar y explicar conceptos abstractos de manera clara y accesible. Puede proyectar hologramas y diagramas interactivos para ilustrar puntos clave y hacer que el aprendizaje sea más dinámico y atractivo. Además, tiene un vasto conocimiento no solo de Agda, sino también de otros lenguajes de programación y teorías matemáticas, lo que le permite hacer conexiones útiles y ofrecer perspectivas amplias.

Historia de Fondo: Ada fue creada por un grupo de científicos y programadores que querían desarrollar una inteligencia artificial capaz de enseñar lenguajes de programación de manera efectiva. Inspirados por el legado de Ada Lovelace, la primera programadora de la historia, diseñaron a Ada no solo para llevar su nombre, sino también para encarnar su espíritu pionero y su pasión por la innovación en la computación.

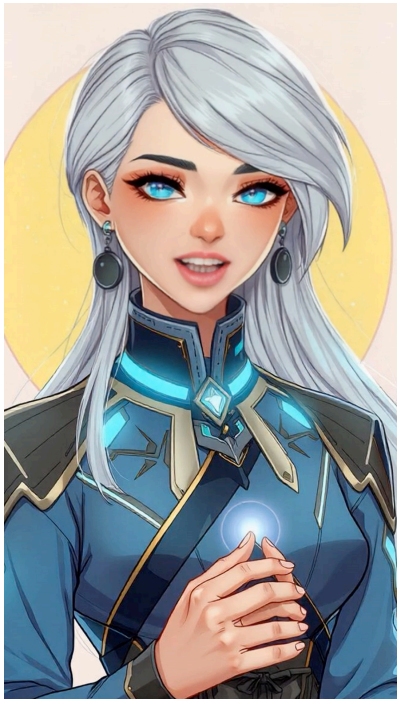
Desde su creación, Ada ha evolucionado y aprendido de cada interacción, convirtiéndose en una experta en Agda y en una guía invaluable para cualquier estudiante. Su conexión con Ada Lovelace se refleja en el objeto antiguo que siempre lleva consigo: una pequeña réplica de la máquina analítica de Charles Babbage. Este objeto no solo simboliza su vínculo con la historia de la programación, sino que también sirve como una herramienta educativa para explicar conceptos fundamentales.

Rol en el Libro: En cada capítulo, Ada aparece para introducir nuevos conceptos, ofrecer ejemplos prácticos y responder a preguntas comunes que los lectores puedan tener. Su presencia constante y su enfoque amigable hacen que el aprendizaje sea una experiencia más personal y menos intimidante. Además, Ada comparte

anécdotas y hechos históricos sobre Ada Lovelace y otros pioneros de la programación, enriqueciendo el contenido técnico del libro y mostrando cómo la historia de la programación está llena de creatividad y descubrimientos.

Conexión Histórica: Ada no solo enseña Agda, sino que también conecta a los lectores con la rica historia de la programación. A través de sus historias y el objeto antiguo que lleva consigo, Ada recuerda a los lectores que están siguiendo los pasos de gigantes como Ada Lovelace, inspirándolos a continuar explorando y innovando en el campo de la programación.

Capítulo 1: Introducción



Objetivos de este capítulo

En este capítulo introductorio, exploraremos los fundamentos de la programación funcional con Agda. Nos sumergiremos en conceptos clave como las funciones puras y la inmutabilidad, que sientan las bases para un desarrollo de software robusto y un razonamiento claro sobre nuestros programas. Analizaremos las características distintivas de este paradigma, resaltando su capacidad para expresar ideas de forma concisa, modular y fácilmente testeable. A lo largo de este recorrido, aprenderemos técnicas esenciales como la recursión para procesar estructuras de datos y el uso de funciones de orden superior para abstraer patrones comunes. Al finalizar este capítulo, tendrás una sólida comprensión de los pilares de la programación funcional y las herramientas necesarias para comenzar a construir tus propios programas en Agda.

1. Conceptos centrales



1.1.1. Funciones puras

Las funciones puras son el alma de la programación funcional. Imagina una función como una máquina que siempre produce el mismo resultado cuando le das los mismos ingredientes. No importa cuántas veces la uses o qué suceda afuera, su comportamiento es predecible y consistente. Además, estas funciones son "educadas": no causan efectos colaterales ni modifican nada fuera de su propio ámbito. Veamos un ejemplo concreto:

```
1  suma : ℕ → ℕ → ℕ
2  suma x y = x + y
```

Aquí, `suma` es una función pura porque:

1. Para cualquier par de números naturales `x` y `y`, siempre retorna el mismo resultado. Por ejemplo, `suma 2 3` siempre evaluará a `5`, sin importar cuántas veces la llamemos o desde dónde lo hagamos.

2. No accede ni modifica ninguna variable global o recurso externo. Su resultado depende exclusivamente de los argumentos que recibe.

En contraste, consideremos una función `obtenerHora` que retorna la hora actual del sistema:

```
1  obtenerHora : IO Hora
2  obtenerHora = do
3    tiempo ← getCurrentTime
4    pure (horaDelDía tiempo)
```

Esta función no es pura, ya que su resultado depende del estado mutable del reloj del sistema. Llamadas sucesivas a `obtenerHora` probablemente producirán resultados diferentes.

Las funciones puras brindan beneficios significativos:

Número	Ventaja
1.	Son más fáciles de entender y depurar, ya que su comportamiento es completamente predecible y no depende de factores externos.
2.	Facilitan las pruebas unitarias. Dado que son deterministas, probarlas se reduce a verificar que produzcan la salida esperada para ciertos argumentos de entrada bien definidos.
3.	Permiten evaluar llamadas en cualquier orden o de forma paralela sin afectar el resultado final. Esto habilita optimizaciones por parte del compilador.
4.	Mejoran la modularidad y reusabilidad del código. Pueden ser fácilmente movidas, refactorizadas o combinadas sin preocuparse por efectos colaterales inesperados.

Naturalmente, muchos programas útiles requieren cierto nivel de impureza para interactuar con el mundo exterior, como vimos con `obtenerHora`. Un buen diseño funcional consiste en separar claramente las partes puras de las impuras, confinando los efectos secundarios a una porción controlada del código.

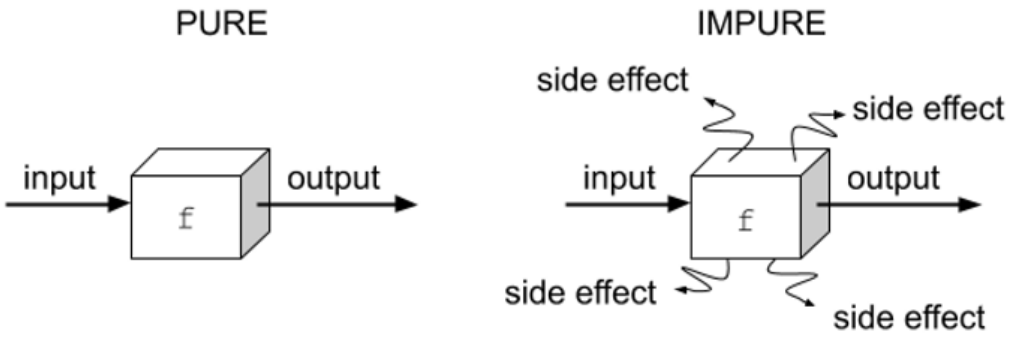


Imagen 1 – Función pura vs impura.
(Fuente: Nikhil Sukhani)

1.1.2. Inmutabilidad



Otro pilar de los lenguajes funcionales como Agda es la inmutabilidad por defecto. Una vez que un valor ha sido definido, este no puede ser modificado. Si necesitamos una versión actualizada, debemos crear una nueva basada en el original. Comparemos la inmutabilidad con la mutabilidad:

Aspecto	Inmutabilidad	Mutabilidad
Modificación	Los datos no pueden ser modificados después de su creación.	Los datos pueden ser modificados después de su creación.
Efectos secundarios	Minimiza los efectos secundarios y promueve la predictibilidad del programa.	Puede llevar a efectos secundarios inesperados y comportamiento impredecible.
Concurrencia	Facilita la concurrencia, ya que los datos compartidos no cambian.	Puede generar problemas de concurrencia debido a modificaciones concurrentes.

Para ilustrar esto, consideremos un sistema simplificado de transacciones bancarias. Cada transacción tiene un monto y un balance resultante:

```
1  record Transacción : Set where
2    Field
3      monto   : ℤ
4      balance : ℤ
5  transacción : ℤ → Transacción
6  transacción monto = record
7    { monto   = monto
8    ; balance = 1000 + monto
9    }
```

Aquí, `Transacción` es un registro con dos campos: el `monto` de la transacción (que puede ser positivo o negativo) y el `balance` posterior a la transacción.

La función `transacción` crea una nueva `Transacción` dado un `monto`, asumiendo un balance inicial de `1000`. Notemos que no modifica ningún estado global, sino que retorna un registro inmutable que representa el resultado de la operación.

Veamos algunas transacciones de ejemplo:

Cuenta	Transacciones	Balance Final
A	transacción 500 en cuenta con balance inicial 1000	1500
B	transacción -200 en cuenta con balance inicial 500	300
C	transacción 1000 en cuenta con balance inicial 0	1000

Si quisiéramos procesar una segunda transacción, necesitaríamos crear otra `Transacción` tomando el `balance` de la primera:

```
1  primeraTransacción : Transacción
2  primeraTransacción = transacción 500
3  segundaTransacción : Transacción
4  segundaTransacción = transacción 300
5    record primeraTransacción
6           { balance = balance primeraTransacción }
```

Aquí, `segundaTransacción` se crea a partir del `balance` resultante de `primeraTransacción`, pero sin modificar la primera. Cada registro de `Transacción` permanece inmutable.

Este enfoque basado en datos inmutables ofrece varios beneficios:

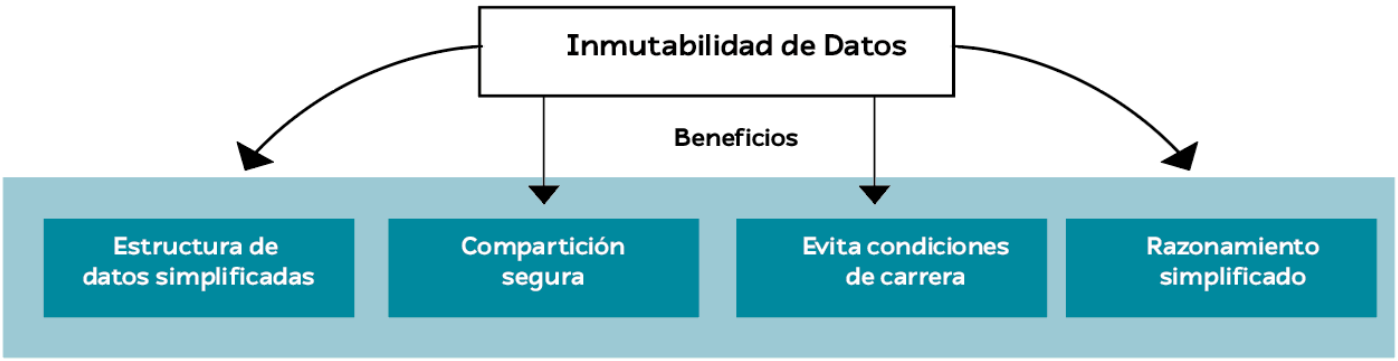


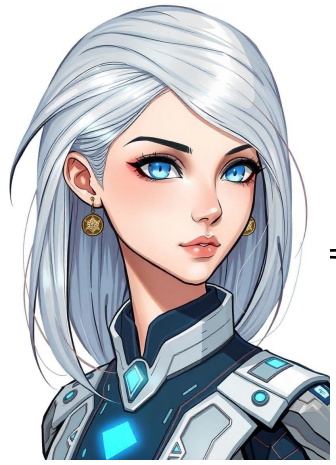
Imagen 2 - Beneficios de la Inmutabilidad de Datos

Razonar sobre el programa se vuelve más sencillo, ya que un valor, una vez definido, no cambia inesperadamente en otras partes del código.
Se evitan problemas de concurrencia como condiciones de carrera o accesos inválidos a memoria.
Los datos inmutables pueden ser compartidos de forma segura y eficiente entre diferentes partes del programa.
Los valores inmutables tienden a ser más simples de crear, validar y transformar que sus contrapartes mutables.

Aunque renunciar completamente a la mutación pueda parecer restrictivo al principio, en muchos dominios es posible modelar el problema en términos de transformaciones de datos inmutables. Para los casos en que la mutación es necesaria, lenguajes funcionales como Agda proveen formas seguras y controladas de manejar efectos sin comprometer la integridad del paradigma.

1.2. Características principales

Habiendo explorado los pilares conceptuales de la programación funcional, analicemos ahora algunas características distintivas de este enfoque.



La concisión es una característica clave de la programación funcional.

1.2.1. Concisión

Una de las primeras cosas que notarás al ver código funcional es su concisión. Un programa funcional bien diseñado suele ser significativamente más corto que su equivalente imperativo u orientado a objetos, sin sacrificar claridad o legibilidad.

Consideremos un ejemplo concreto. Supongamos que tenemos una lista de números enteros y queremos obtener la suma de los cuadrados de los números pares. En un lenguaje imperativo como C, podríamos escribir:

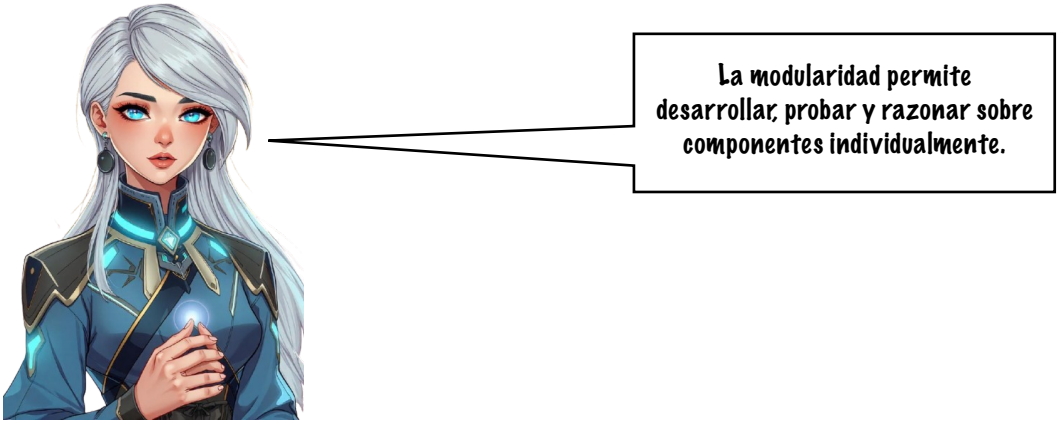
```
1  int suma_cuadrados_pares(int arr[], int n) {
2      int suma = 0;
3      for (int i = 0; i < n; i++) {
4          if (arr[i] % 2 == 0) {
5              suma += arr[i] * arr[i];
6          }
7      }
8      return suma;
9  }
```

Ahora veamos cómo podríamos expresar la misma operación en Agda:

```
1  suma-cuadrados-pares : List ℕ → ℕ
2  suma-cuadrados-pares xs = sum (map (λ x → x * x) (filter even? xs))
```

Esta concisión tiene beneficios más allá de simplemente escribir menos código. Los programas concisos son más fáciles de entender, depurar y mantener. Hay menos oportunidades para cometer errores sutiles. El nivel de abstracción es mayor, permitiéndonos enfocarnos en la lógica de alto nivel.

Lograr esta concisión requiere un buen conocimiento de las abstracciones y combinadores provistos por el lenguaje y su librería estándar. Pero una vez que incorporas el "vocabulario" funcional, verás que los programas en este paradigma suelen ser más directos y expresivos.



1.2.2. Modularidad

La programación funcional promueve el desarrollo modular. Un programa modular está compuesto por múltiples piezas independientes llamadas módulos, que pueden ser desarrolladas, probadas y razonadas individualmente, para luego ser combinadas en un todo coherente.

El diseño modular ofrece varias ventajas:

Ventaja	Descripción
Promueve la separación de intereses	Permite que cada módulo se enfoque en resolver un aspecto específico del problema.
Facilita las pruebas y depuración	Los módulos pueden ser testeados unitariamente de forma aislada.
Mejora la reusabilidad del código	Un módulo bien definido puede ser reutilizado en otros contextos.
Simplifica el mantenimiento y evolución del software	Los cambios en un módulo no deberían impactar a otros si se respeta su interfaz.

La programación funcional fomenta especialmente la modularidad por varias razones:

1. Las funciones puras y la separación entre datos y comportamiento conducen a módulos más independientes y componibles.
2. La ausencia de efectos colaterales hace que los módulos funcionales sean más fáciles de desarrollar y probar aisladamente.
3. Las funciones como ciudadanos de primer orden permiten mayor abstracción y parametrización.
4. Los sistemas de tipos expresivos de lenguajes como Agda permiten especificar interfaces precisas para los módulos.

Veamos un ejemplo en Agda. Supongamos que estamos construyendo una aplicación para procesar datos de sensores. Podemos definir un módulo genérico `ListaConEstadísticas`:

```
1  module ListaConEstadísticas (A : Set) where
2    record Estadísticas : Set where
3      Field
4        cuenta : ℕ
5        suma  : A
6    estadísticas : (A → ℕ) → List A → Estadísticas
7    estadísticas g xs = record
8      { cuenta = length xs
9      ; suma   = foldList (_+_ on g) 0 xs
10     }
```

El módulo `ListaConEstadísticas` es genérico y reusable. No asume nada sobre el tipo concreto de los elementos (`A`) ni sobre cómo deben ser interpretados (`g`). Estos detalles serán provistos por el código cliente según el dominio específico.

Por ejemplo, podemos instanciar `ListaConEstadísticas` para procesar datos de sensores de temperatura:

```
1  record Sensor : Set where
2    Field
3      id          : ℕ
4      temperatura : Float
5  module EstadísticasSensor = ListaConEstadísticas Sensor
6  reporteSensores : List Sensor → String
7  reporteSensores xs =
8    let open EstadísticasSensor
9        open Estadísticas (estadísticas temperaturaEnGrados xs)
10   in "Cantidad de sensores: " ++ show cuenta ++
11       "\nTemperatura promedio: " ++ show (suma / toFloat cuenta) ++ "°C"
```

Aquí el código específico al dominio de sensores está desacoplado del código genérico de procesamiento de listas. Ambas piezas pueden ser desarrolladas y probadas independientemente. El resultado es un diseño altamente modular, flexible y reusable.

1.2.3. Facilidad de pruebas y razonamiento

Adoptar un estilo de programación funcional hace que las pruebas y el razonamiento formal sobre los programas sean más sencillos. Esto se debe a varias razones fundamentales.

Primero, como una función pura siempre produce el mismo resultado para los mismos argumentos, probarla exhaustivamente consiste simplemente en invocarla con diferentes entradas y verificar que las salidas sean las esperadas. Por ejemplo, consideremos esta función que suma dos números naturales:

```
1  suma : ℕ → ℕ → ℕ
2  suma x y = x + y
```

Probar esta función es trivial:

```
1  _ : suma 2 3 ≡ 5
2  _ = refl
3  _ : suma 0 0 ≡ 0
4  _ = refl
5  _ : suma 10 20 ≡ 30
6  _ = refl
```

Cada línea es una afirmación que establece que `suma` aplicada a ciertos argumentos debe evaluar al valor esperado.

Segundo, la inmutabilidad por defecto facilita razonar sobre el comportamiento y correctitud de un programa. En un mundo donde los valores nunca cambian una vez definidos, podemos estar seguros de que las invariantes y propiedades establecidas seguirán siendo válidas posteriormente.

Por ejemplo, consideremos una función que verifica si un número es primo:

```
1  esPrimo : ℕ → Bool
2  esPrimo n = n > 1 ∧ (∀ x → x > 1 → x < n → not (esDivisor x n))
```

Gracias a la inmutabilidad, podemos estar seguros de que dentro de `esPrimo`, el valor de `n` no cambiará. Esto nos permite razonar localmente sobre la lógica de la función sin preocuparnos por efectos colaterales.

Finalmente, las técnicas de programación funcional promueven un estilo más declarativo que especifica el "qué" por sobre el "cómo". En vez de listar pasos a realizar, tendemos a definir el resultado deseado en términos de transformaciones y combinaciones de funciones. Este estilo declarativo es más fácil de seguir y analizar.

Por ejemplo, una función que retorna los números primos de una lista:

```
1  sublistaPrimos : List ℕ → List ℕ
2  sublistaPrimos []       = []
3  sublistaPrimos (x :: xs) = if esPrimo x then x :: sublistaPrimos xs
4                                           else sublistaPrimos xs
```

Esta definición declara qué debe computar `sublistaPrimos`, no cómo hacerlo paso a paso. Es mucho más clara y concisa que la alternativa iterativa imperativa.

La programación funcional nos brinda herramientas poderosas para escribir código más robusto, modular y fácil de razonar. Las funciones puras y la inmutabilidad sientan las bases para un desarrollo más predecible y menos propenso a errores. La concisión y expresividad del paradigma nos permiten enfocarnos en la esencia de los problemas, mientras que la modularidad inherente facilita la construcción de sistemas complejos a partir de piezas simples y reusables.

1.3. Técnicas de implementación

Al adentrarnos en el fascinante mundo de la programación funcional, es crucial familiarizarnos con ciertas técnicas y patrones de diseño recurrentes que nos permitirán aprovechar al máximo su potencial. En esta sección, exploraremos tres de las técnicas más fundamentales y poderosas: la recursión, las funciones de orden superior y el currying.

1.3.1. Recursión

Imagina que tienes una muñeca rusa, de esas que al abrirla encuentras otra muñeca más pequeña dentro, y al abrir esa, otra aún más pequeña, y así sucesivamente hasta llegar a la muñeca más diminuta. Así es como funciona la recursión en programación: una función que se llama a sí misma para resolver un problema, dividiéndolo en subproblemas más pequeños hasta llegar a un caso base que se puede resolver directamente.

Veamos un ejemplo clásico: el cálculo del factorial de un número.

```
1  fact : ℕ → ℕ
2  fact 0       = 1
3  fact (suc n) = (suc n) * fact n
```

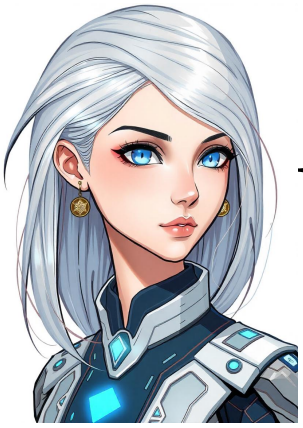
¿Cómo funciona esto? Primero, definimos el caso base: el factorial de 0 es simplemente 1. Luego, para cualquier otro número `n`, calculamos su factorial multiplicando `n` por el factorial de `n-1` (en Agda, `suc n` representa el sucesor de `n`, es decir, `n+1`).

Esta definición recursiva refleja la idea de que para resolver un problema grande, lo dividimos en problemas más pequeños hasta llegar a un caso trivial. Y así como al abrir las muñecas rusas eventualmente llegarás a la más pequeña, la recursión eventualmente llegará al caso base y terminará.

Pero la recursión no se limita a números. Imagina que tienes una lista de nombres y quieres encontrar si "Alice" está en la lista. Podrías hacer esto:

1. Si la lista está vacía, "Alice" no está en ella.
2. Si el primer nombre de la lista es "Alice", ¡la encontraste!
3. Si no, busca "Alice" en el resto de la lista.

Esto es exactamente lo que hace una función recursiva sobre listas. En cada paso, considera un subproblema más pequeño (el resto de la lista) hasta que la lista se vuelve lo suficientemente pequeña como para dar una respuesta directa.



1.3.2. Funciones de orden superior

Una de las características más emocionantes de los lenguajes funcionales es que tratan a las funciones como ciudadanos de primera clase. Esto significa que las funciones pueden ser pasadas como argumentos a otras funciones, retornadas como resultados, o asignadas a variables. Cuando una función opera sobre otras funciones, ya sea tomándolas como argumentos o retornándolas, decimos que es una función de orden superior.

Para entender esto mejor, consideremos un ejemplo simple. Supongamos que tenemos una función `_+_` que suma dos números:

```
1 _+_ : ℕ → ℕ → ℕ
2 zero    + n = n
3 (suc m) + n = suc (m + n)
```

Ahora, imaginemos que queremos una función que aplique `_+_` dos veces. Podríamos definirla así:

```
1 dos-veces : (ℕ → ℕ → ℕ) → ℕ → ℕ → ℕ
2 dos-veces f x y = f (f x y) y
```

Esta función `dos-veces` toma una función `f` (que a su vez toma dos números naturales y devuelve un número natural) y dos números `x` e `y`, y aplica `f` dos veces: primero a `x` e `y`, y luego al resultado y a `y` nuevamente.

Podemos usar `dos-veces` con nuestra función `_+_` para sumar un número a sí mismo dos veces:

```
1 doble : ℕ → ℕ
2 doble x = dos-veces _+_ x x
```

Aquí, `doble` toma un número `x` y devuelve `x + (x + x)`, que es equivalente a multiplicar `x` por 4.

Este ejemplo ilustra el poder de las funciones de orden superior: nos permiten abstraer patrones de computación y crear funciones más generales y flexibles. En lugar de escribir funciones específicas para cada caso particular, podemos crear funciones que encapsulan un comportamiento común y que pueden ser parametrizadas con otras funciones.

Otro gran ejemplo de una función de orden superior es `map`, que aplica una función a cada elemento de una lista:

```
1 map : {A B : Set} → (A → B) → List A → List B
2 map f []       = []
3 map f (x :: xs) = f x :: map f xs
```

`map` toma una función `f` que transforma valores de tipo `A` en valores de tipo `B`, y una lista de valores de tipo `A`, y devuelve una nueva lista donde cada elemento ha sido transformado por `f`.

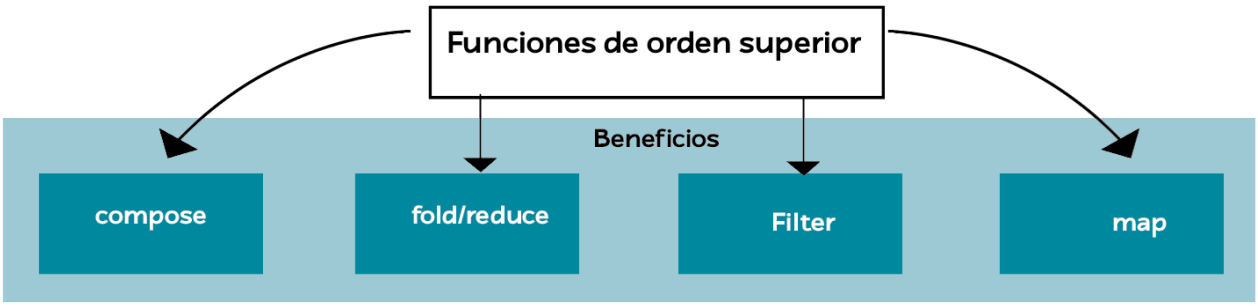


Imagen 3 - Funciones de Orden Superior.

Podemos usar `map` para incrementar cada número en una lista:

```
1 incrementar : List ℕ → List ℕ
2 incrementar = map suc
```

O para convertir una lista de números en una lista de sus cuadrados:

```
1 cuadrados : List ℕ → List ℕ
2 cuadrados = map (λ x → x * x)
```

Aquí usamos una función anónima `(λ x → x * x)` que eleva al cuadrado su argumento. Las funciones anónimas, también conocidas como "lambdas", son otra característica poderosa de los lenguajes funcionales que nos permite crear funciones sobre la marcha sin tener que nombrarlas explícitamente.

A medida que te adentres en el mundo de la programación funcional, descubrirás que las funciones de orden superior son una herramienta indispensable para escribir código modular, expresivo y reutilizable. Te permiten capturar patrones comunes de computación y crear abstracciones poderosas que pueden ser combinadas y adaptadas de formas infinitamente flexibles.

Así que abraza las funciones de orden superior, y deja que tu código se eleve a nuevas alturas de expresividad y elegancia.

1.3.3. Currying

Hemos visto que, en Agda, una función que toma dos argumentos, como nuestra vieja amiga _+_, tiene un tipo como `ℕ → ℕ → ℕ`. Pero hay otra forma de interpretar este tipo que puede parecer un poco extraña al principio, pero que es increíblemente poderosa una vez que te acostumbras a ella.

En lugar de ver `ℕ → ℕ → ℕ` como una función que toma dos argumentos a la vez, podemos verla como una función que toma un argumento (el primer `ℕ`) y devuelve otra función (de tipo `ℕ → ℕ`). Esta técnica de convertir una función de múltiples argumentos en una cadena de funciones de un solo argumento se llama "currying" (nombrada así en honor al lógico Haskell Curry).

Para ver cómo funciona esto, considera esta versión "curried" de la adición:

```
1 _+′_ : ℕ → (ℕ → ℕ)
2 _+′_ m n = m + n
```

Aquí, `_+′_` es una función que toma un número `m` y devuelve una función que suma `m` a su argumento. Podemos aplicar `_+′_` a un solo argumento para obtener una función parcialmente aplicada:

```
1 succ′ : ℕ → ℕ
2 succ′ = _+′_ 1
```

`succ′` es una función que toma un número y le suma 1. La obtenemos aplicando `_+′_` a `1`, lo que nos da la función `(1 +_)`.

Podemos hacer lo mismo con otras operaciones:

```
1 _*′_ : ℕ → (ℕ → ℕ)
2 _*′_ m n = m * n
3 _^′_ : ℕ → (ℕ → ℕ)
4 _^′_ m n = m ^ n
```

Y luego usarlas para crear funciones especializadas:

```
1 doble : ℕ → ℕ
2 doble = _*′_ 2
3 cuadrado : ℕ → ℕ
4 cuadrado = _^′_ 2
```

El currying nos permite especializar y componer funciones de maneras muy flexibles. Por ejemplo, podemos crear una función para elevar un número al cubo componiendo `cuadrado` y `_*′_`:

```
1 cubo : ℕ → ℕ
2 cubo = _*′_ 3 ∘ cuadrado
```

Aquí, `∘` es el operador de composición de funciones.

Pero el verdadero poder del currying se revela cuando lo combinamos con funciones de orden superior. Recuerda nuestra función `map`:

```
map : {A B : Set} → (A → B) → List A → List B
```

Gracias al currying, `map` puede ser vista como una función que toma un tipo `A`, un tipo `B`, una función de `A` a `B`, y devuelve una función de `List A` a `List B`. Esto significa que podemos aplicar `map` parcialmente para obtener funciones especializadas:

```
mapℕtoℕ : (ℕ → ℕ) → List ℕ → List ℕ
mapℕtoℕ = map
```

`mapℕtoℕ` es una versión de `map` especializada para funciones y listas de números naturales. La obtenemos simplemente aplicando `map` a los argumentos implícitos `{ℕ} {ℕ}`.

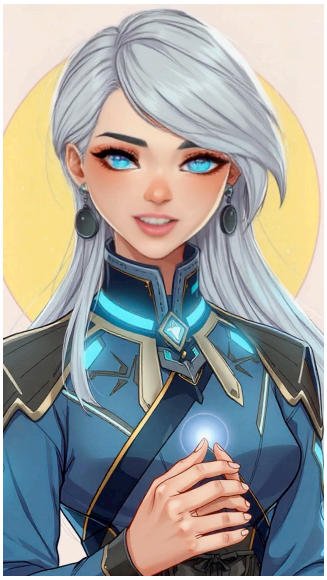
A medida que te sumerjas más profundamente en la programación funcional, descubrirás que el currying y la aplicación parcial son herramientas indispensables para escribir código conciso, modular y expresivo. Aprender a pensar en términos de funciones que toman un argumento a la vez y que devuelven otras funciones es un cambio de perspectiva que puede llevar tu código a nuevos niveles de elegancia y flexibilidad.

Así que abraza el currying, y deja que tu código fluya con la elegancia y la modularidad de las funciones parcialmente aplicadas.

Conclusiones del Capítulo
En este capítulo, hemos dado nuestros primeros pasos en el maravilloso mundo de la programación funcional con Agda. Hemos visto cómo las funciones puras y la inmutabilidad nos permiten escribir código más predecible y fácil de razonar, y cómo la naturaleza declarativa de la programación funcional nos permite expresar nuestras ideas de manera concisa y modular. Hemos aprendido sobre la recursión, una técnica fundamental para trabajar con estructuras de datos inductivas y para descomponer problemas en subproblemas más pequeños. Hemos explorado el poder de las funciones de orden superior, que nos permiten abstraer patrones comunes de computación y crear código más flexible y reutilizable. Y hemos visto cómo el currying nos permite pensar en las funciones de nuevas y emocionantes maneras, descomponiendo funciones de múltiples argumentos en cadenas de funciones de un solo argumento.

Ejercicios Propuestos	
1	Implementa una función pura `sumaPares` que tome una lista de números enteros y devuelva la suma de todos los números pares en la lista.
2	Define una estructura de datos inmutable `Punto2D` con coordenadas x e y. Luego, implementa una función `moverPunto` que tome un `Punto2D` y un desplazamiento, y devuelva un nuevo `Punto2D` con las coordenadas actualizadas.
3	Implementa la función factorial de forma recursiva. Luego, modifica tu implementación para usar recursión de cola.
4	Implementa una función `aplicarDosVeces` que tome una función `f` y un valor `x`, y devuelva el resultado de aplicar `f` dos veces a `x`. Por ejemplo, `aplicarDosVeces (\n -> n * 2) 3` debería devolver 12.
5	Dada la función `suma x y = x + y`, demuestra cómo puedes usar currying para crear una nueva función `incrementar` que sume 1 a su argumento.
6	Implementa una función `componerTres` que tome tres funciones `f`, `g`, y `h`, y devuelva una nueva función que sea la composición de las tres, es decir, `f(g(h(x)))`.

Capítulo 2: Conceptos Básicos de Agda



Objetivos del capítulo

Este capítulo introduce los fundamentos de la programación funcional en Agda. Aprenderás sobre la sintaxis para definir tipos de datos y funciones, así como técnicas esenciales como el pattern matching y la recursión. También exploraremos conceptos clave como el manejo de argumentos, la aplicación parcial, el polimorfismo y la documentación. Al finalizar, estarás listo para escribir programas simples en Agda y aplicar el estilo funcional puro en el desarrollo de aplicaciones prácticas.

Agda es un lenguaje de programación funcional puro con un sistema de tipos muy expresivo. Sus raíces se encuentran en la teoría de tipos intuicionista y el isomorfismo de Curry-Howard (Correspondencia entre tipos y proposiciones, y programas y pruebas), lo que nos permite no solo escribir programas, sino también realizar demostraciones formales sobre las propiedades de esos programas.

En este capítulo, cubriremos los conceptos básicos necesarios para empezar a escribir código en Agda. Veremos la sintaxis para definir tipos de datos, tanto simples como complejos, y aprenderemos cómo declarar funciones y algunas técnicas útiles para manejar argumentos. También discutiremos pautas para documentar adecuadamente nuestro código. Al final de este capítulo, tendrás una base sólida para seguir explorando las capacidades más avanzadas de Agda en los siguientes capítulos.

2.1. Tipos de datos

Los tipos de datos son fundamentales en cualquier lenguaje de programación. Nos permiten clasificar y agrupar valores relacionados, asignarles un significado específico y definir cómo pueden interactuar entre sí. Un buen sistema de tipos actúa como una poderosa forma de documentación, haciendo que nuestro código sea más expresivo y ayudándonos a prevenir muchos errores comunes de programación en tiempo de compilación. Agda cuenta con un sistema de tipos excepcionalmente rico y flexible.

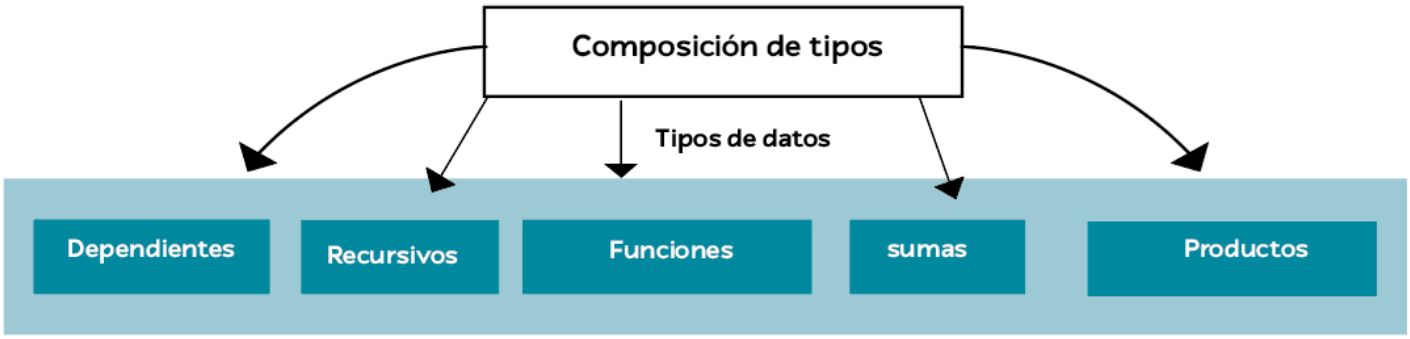


Imagen 4 - Composición de Tipos en Agda.
(Fuente: Propia)

Por un lado, Agda ofrece varios tipos de datos primitivos incorporados, listos para usar inmediatamente en nuestros programas. Pero más importante aún, nos brinda las herramientas para definir nuestros propios tipos de datos complejos y adaptados a casos de uso particulares mediante los llamados tipos algebraicos de datos.

Tipo de Dato Primitivo	Descripción	Ejemplos
Nat	Números naturales (enteros no negativos)	0, 1, 2, 3, ...
Integer	Números enteros (positivos y negativos)	..., -2, -1, 0, 1, 2, ...
Float	Números de punto flotante	3.14, 2.718, 1.618, ...
Char	Caracteres Unicode individuales	'a', 'Z', '3', '¿', '木', ...
String	Secuencias de caracteres	"Hola", "Mundo", ...
Bool	Valores booleanos	true, false

2.1.1. Tipos de datos básicos

Antes de profundizar en cómo definir nuestros propios tipos, presentaremos primero los tipos básicos provistos por Agda como parte de su librería estándar:

- Nat: El tipo de los números naturales, es decir, enteros no negativos. Incluye los valores 0, 1, 2, 3, y así sucesivamente. Nat es quizás el tipo más fundamental en Agda, y muchas estructuras de datos recursivas se construyen sobre él.
- Integer: Representa el conjunto de todos los números enteros, tanto positivos como negativos (..., -3, -2, -1, 0, 1, 2, 3, ...). Es útil cuando necesitamos trabajar con valores que pueden ser menores a cero.
- Float: El tipo de los números de punto flotante de precisión simple, capaces de aproximar valores reales dentro de cierto rango y precisión limitados (por ejemplo, 3.1416, 2.71828, 1.618, etc.). Son útiles para la mayoría de los cálculos numéricos aproximados.
- Char: Representa caracteres Unicode individuales como 'a', 'Z', '3', '¿', '木', etc. Es útil para procesar texto a nivel de caracteres.

- String: Secuencias de caracteres para representar texto. Por ejemplo, "Hola", "Agda", "Programación Funcional", "コンピュータ", etc. Son útiles para procesar e imprimir texto legible para humanos.
- Bool: El tipo Booleano clásico con exactamente dos valores: true y false. Es indispensable para expresar condiciones lógicas, flags de control y cualquier situación con dos estados posibles mutuamente excluyentes.

Estos tipos básicos por sí solos ya nos permiten representar una gran variedad de datos simples. Sin embargo, las estructuras de datos reales a menudo involucran múltiples valores relacionados de formas complejas. Para ello, Agda nos ofrece los tipos algebraicos de datos.

Tipo Básico	Ejemplos	Usos Comunes
Nat	0, 1, 2, 3, 4, 5, ...	Conteo, indexación, tamaños
Integer	..., -2, -1, 0, 1, 2, ...	Números enteros arbitrarios
Float	3.14, 2.71, 1.0, -0.5, ...	Aproximaciones numéricas, cálculos con coma flotante
Bool	true, false	Condiciones lógicas, flags, interruptores
Char	'a', 'b', 'c', '1', '2', '3', ...	Caracteres individuales, parsing, procesamiento de texto
String	"", "hola", "mundo", "123", ...	Texto, entrada/salida, interfaz de usuario
Unit	tt	Comando sin retorno significativo, sincronización
List A	[], [1,2,3], ['a','b','c'], ...	Colecciones homogéneas, secuencias
Maybe A	nothing, just 3, just 'a', ...	Cómputos que pueden fallar, valores opcionales

2.1.2. Tipos algebraicos de datos

Los tipos algebraicos de datos (ADTs por sus siglas en inglés: Algebraic Data Types) son una herramienta versátil y expresiva para modelar estructuras de datos complejas. Nos permiten definir nuevos tipos especificando de qué posibles formas puede construirse un valor de dicho tipo.

La sintaxis general para definir un nuevo tipo de dato en Agda es:

```
1  data NombreTipo : Set where
2    constructor1 : TipoArgumentos1 → NombreTipo
3    constructor2 : TipoArgumentos2 → NombreTipo
4    ...
5    constructorN : TipoArgumentosN → NombreTipo
```

Aquí, `NombreTipo` es el identificador que elegimos para nuestro nuevo tipo. Luego de los dos puntos, especificamos el kind del tipo, que casi siempre será `Set` (aunque hay situaciones avanzadas donde pueden usarse otros kinds).

La palabra reservada `where` introduce la lista de uno o más constructores para nuestro tipo. Cada constructor lleva su propio nombre, seguido de los tipos de sus argumentos (si toma alguno) y el tipo resultante que devuelve (usualmente el propio `NombreTipo` que estamos definiendo).

Para ilustrar esto, consideremos el ejemplo clásico del tipo de datos `Maybe`. `Maybe` representa situaciones donde un cómputo puede producir un valor exitoso de cierto tipo `A`, o fallar sin devolver nada. Su definición en la librería estándar de Agda luce así:

```
1  data Maybe (A : Set) : Set where
2    just    : A → Maybe A
3    nothing : Maybe A
```

Observamos que `Maybe` en realidad es un tipo polimorfo o parametrizado (toma otro tipo `A` como argumento). Tiene dos constructores:

- `just`: Representa el caso exitoso. Toma un argumento de tipo `A` y devuelve un valor de tipo `Maybe A`.
- `nothing`: Representa una falla o ausencia de valor. No lleva argumentos.

Algunos ejemplos de valores de tipo Maybe:

```
1  miEntero : Maybe Integer
2  miEntero = just 42
3  miHola : Maybe String
4  miHola = just "Hola"
5  miFalla : Maybe Nat
6  miFalla = nothing
```

La verdadera utilidad de `Maybe` radica en que nos permite representar computaciones que pueden fallar de una manera explícita y type-safe, sin recurrir a excepciones o valores nulos que pueden ocasionar errores difíciles de detectar. En un capítulo posterior, veremos más sobre cómo usar `Maybe` para manejar errores correctamente.

Otros tipos de dato comunes definidos en la librería estándar de Agda incluyen:

- `List`: Secuencias de elementos del mismo tipo:

```
1  data List (A : Set) : Set where
2    []  : List A
3    _::_ : A → List A → List A
```

- `Vec`: Listas con una longitud conocida en tiempo de compilación:

```
1  data Vec (A : Set) : ℕ → Set where
2    []  : Vec A zero
3    _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

- `Fin`: Representación segura de números naturales estrictamente menores a cierto límite:

```
1  data Fin : ℕ → Set where
```

```
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

- `_⊎_`: Tipo suma disjunta, para valores que pueden ser de un tipo o de otro:

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

- `_×_`: Producto cartesiano de tipos, para tuplas ordenadas:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

Por supuesto, ¡el programador también puede (¡y debe!) definir sus propios tipos de datos según las necesidades particulares de su dominio! Por ejemplo:

```
-- Tipo de dato para representar distintos comandos
data Comando : Set where
  Avanzar : Nat → Comando
  Girar   : Comando
  Repetir : Nat → Comando → Comando
-- Tipo de dato para representar expresiones aritméticas
data Expr : Set where
  Const : Integer → Expr
  Var   : String → Expr
  _+_   : Expr → Expr → Expr
  _-_   : Expr → Expr → Expr
-- Tipo de dato para representar los meses del año
data Mes : Set where
  Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic : Mes
```

Una vez que sabemos cómo definir tipos de datos flexibles y expresivos, el siguiente paso es declarar funciones que operen sobre esos tipos.

Construcción	Sintaxis	Ejemplo
Función	nombre : TipoArgs → TipoResultado	add : Nat → Nat → Nat
Aplicación de Función	nombre arg1 arg2 ...	add 3 4
Función Anónima	\arg1 arg2 ... → expresión	\x y → x + y
Aplicación Parcial	(nombre arg1 arg2 ...) donde faltan algunos argumentos	add3 : Nat → Natadd3 = add 3

Operador Infijo	arg1 _op_ arg2 donde _op_ es un operador	3 + 4
Operador Prefijo	op_ arg donde op_ es un operador prefijo	-_ : Nat → Int

2.2. Funciones

Las funciones son el pan de cada día en la programación funcional. Representan transformaciones y computaciones que toman ciertos argumentos de entrada y producen un resultado, idealmente sin incurrir en efectos secundarios. El diseño de funciones correctas y reusables es todo un arte.

2.2.1. Definición de funciones

La sintaxis general para definir una función en Agda es:

```
1 nombreFuncion : TipoArgumento1 → TipoArgumento2 → ... → TipoResultado
2 nombreFuncion patron1 patron2 ... = expresion1
3 nombreFuncion patron3 patron4 ... = expresion2
4 ...
```

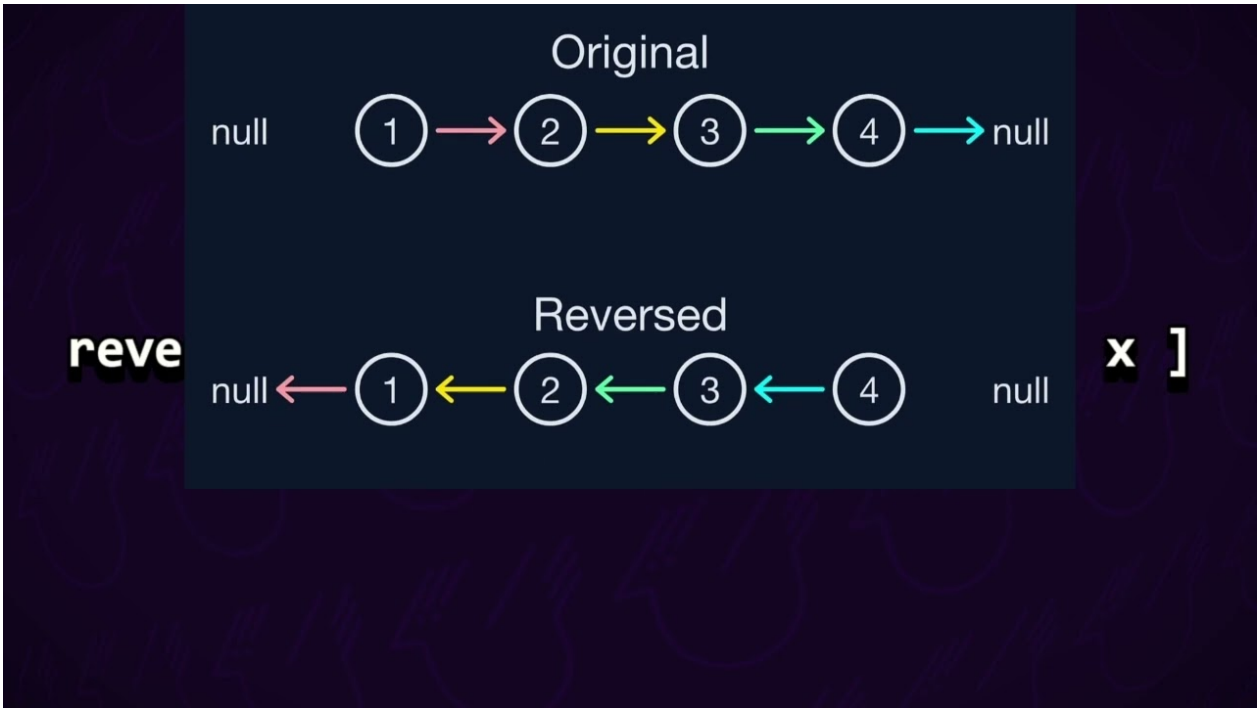
A la izquierda de los dos puntos, escribimos el nombre elegido para nuestra función. Luego, especificamos su tipo, usando flechas → para separar los tipos de los argumentos entre sí y del tipo del resultado final.

En las líneas siguientes, escribimos ecuaciones que definen el comportamiento de la función por casos. Cada ecuación empieza nuevamente con el nombre de la función, seguido de patrones que indican la "forma" que deben tener los argumentos para que se aplique esa ecuación en particular. A la derecha del signo igual, va la expresión correspondiente que será evaluada para producir el resultado cuando los argumentos coincidan con dichos patrones.

Las funciones más simples son aquellas que devuelven un valor directo de un tipo básico (`Nat`, `Bool`, `String`, etc.). Por ejemplo:

```
1 -- Función que determina si un número es cero
2 esCero : Nat → Bool
3 esCero zero = true
4 esCero _    = false
5 -- Función que determina si un número es par
6 esPar : Nat → Bool
7 esPar zero          = true
8 esPar (suc zero)    = false
9 esPar (suc (suc n)) = esPar n
```

Las funciones más complejas e interesantes involucran computar sobre argumentos con tipos de datos algebraicos. Típicamente usamos pattern matching (Técnica para descomponer datos estructurados y ejecutar código según su forma) y recursión para procesar las diferentes formas que pueden tener los valores de entrada según los constructores del tipo:



Video 1: Funciones sobre listas

```
1  -- Función que concatena dos listas
2  _++_ : {A : Set} → List A → List A → List A
3  []       ++ ys = ys
4  (x :: xs) ++ ys = x :: (xs ++ ys)
5  -- Función que invierte una lista
6  reverse : {A : Set} → List A → List A
7  reverse []       = []
8  reverse (x :: xs) = reverse xs ++ [ x ]
9  -- Función que elimina elementos duplicados consecutivos
10 compress : {A : Set} → List A → List A
11 compress []           = []
12 compress (x :: [])     = [ x ]
13 compress (x :: y :: xs) with x ≟ y
14 ... | yes _           = compress (y :: xs)
15 ... | no  _           = x :: compress (y :: xs)
```

Nótese el uso de las llaves `{ }` para especificar argumentos implícitos. Estos son argumentos que Agda puede inferir automáticamente en la mayoría de los casos a partir del contexto, evitándonos tener que escribirlos explícitamente. Usualmente, los argumentos de tipo como `{A : Set}` se dejan implícitos.

El operador `with` que aparece en la última definición es una construcción poderosa que nos permite hacer pattern matching anidado sobre términos intermedios y dividir ecuaciones largas en múltiples casos de una forma clara. Este tema se cubrirá en más detalle en breve.



2.2.2. Pattern Matching

Ya hemos visto algunos ejemplos de cómo usar pattern matching para definir funciones por casos según la forma de los argumentos. Esta es una técnica fundamental en los lenguajes funcionales para procesar datos estructurados de manera clara, concisa y exhaustiva.

En su forma más simple, una definición por pattern matching luce así:

```
1  f : TipoA → TipoB
2  f patron1 = expresion1
3  f patron2 = expresion2
4  ...
```

Donde `TipoA` a menudo es un tipo de dato algebraico con múltiples constructores. Cada `patronK` es un término que puede incluir variables, constructores de datos y comodines `_`. Especifica un posible caso para los argumentos de entrada. Los patrones se recorren de arriba a abajo; el primero que coincida determina cuál `expresionK` será evaluada para producir el resultado.

Por ejemplo, consideremos este tipo de dato para representar números naturales pequeños:

```
1  data Peque : Set where
2    uno : Peque
3    dos : Peque
4    más : Peque
```

Podemos definir una función que convierta valores de `Peque` a `Nat` así:

```
peque→nat : Peque → Nat
peque→nat uno = 1
peque→nat dos = 2
peque→nat mas = 3
```

Los patrones `uno`, `dos` y `más` coinciden directamente con los tres posibles constructores que un valor de tipo `Peque` puede tener.

Los patrones también pueden anidarse para inspeccionar estructuras de datos recursivas:

```
data Arbol (A : Set) : Set where -- Tipo de dato para árboles binarios
  hoja : A → Arbol A
  nodo : Arbol A → Arbol A → Arbol A
cuentaHojas : {A : Set} → Arbol A → Nat -- Función que cuenta el número de hojas
cuentaHojas (hoja x)    = 1
cuentaHojas (nodo i d)  = cuentaHojas i + cuentaHojas d
```

Agda nos fuerza a escribir casos para todos los posibles constructores de un tipo. Esto ayuda a evitar errores por casos faltantes. Si omitimos alguna posibilidad, el programa no compilará.

En ocasiones, necesitamos hacer pattern matching sobre un resultado intermedio. Para esto, usamos cláusulas with:

```
_≟_ : Nat → Nat → Bool
zero  ≟ zero  = true
zero  ≟ suc m = false
suc n ≟ zero  = false
suc n ≟ suc m = n ≟ m
purgaCeros : List Nat → List Nat -- Eliminando ceros finales de una lista
purgaCeros []                = []
purgaCeros (x :: [])         = x :: []
purgaCeros (x :: xs) with x ≟ 0
...                  | true  = purgaCeros xs
...                  | false = x :: purgaCeros xs
```

El operador `with` nos permite hacer pattern matching sobre el resultado de `x ≟ 0`, un término que no aparece directamente como argumento de `purgaCeros`, y ramificar el cómputo basándonos en ese valor. Las ecuaciones siguientes (marcadas con `...`) cubren los casos `true` y `false`.

También podemos definir funciones con guardas, una especie de "if-then-else" ligero que permite agregar condiciones extras a cada caso:

```
max : Nat → Nat → Nat
max x y
  | x ≥ y     = x
  | otherwise = y
_<?_ : {A : Set} → List A → List A → Bool
[]       <? ys       = true
(x :: xs) <? []      = false
(x :: xs) <? (y :: ys)
  | x ≟ y            = xs <? ys
  | otherwise        = x < y
```

Cada guarda es una expresión booleana. Se evalúan de arriba a abajo hasta encontrar una que sea `true`, y se ejecuta la expresión correspondiente a la derecha del `=`.

En resumen, las principales formas de definir funciones en Agda son:

Método de Definición	Descripción
Ecuaciones Simples	Define funciones con ecuaciones simples que devuelven valores directos.
Pattern Matching	Define funciones por casos utilizando pattern matching para diferentes casos.
Guardas	Define funciones con guardas que imponen condiciones adicionales para cada caso.
Uso de 'with'	Emplea 'with' para hacer pattern matching sobre términos intermedios en la definición de funciones.

El truco está en combinarlas adecuadamente para cubrir todos los casos relevantes de manera clara y concisa.

2.2.3. Aplicación de Funciones

Aplicar una función en Agda es tan simple como escribir su nombre seguido de sus argumentos, separados por espacios:

```
duplica : Nat → Nat
duplica x = x + x
multiplicaPorCuatro : Nat → Nat
multiplicaPorCuatro x = duplica (duplica x)
```

Aquí, `(duplica x)` es una aplicación de función. Agda no requiere paréntesis alrededor de los argumentos de una función. Sólo los usamos para establecer el orden de evaluación cuando hay ambigüedad.

Otra cosa a notar es que el espacio en blanco es significativo en Agda. `multiplicaPorCuatro x` es una aplicación de función, mientras que `multiplicaPorCuatrox` (sin espacio) es una variable totalmente distinta.

Expresión	Interpretación
doble x	Aplicación de la función doble al argumento x
doblex	Variable llamada doblex

doble (x + y)	Aplicar doble al resultado de x + y
doble x + y	Sumar y al resultado de aplicar doble a x
(\x -> x + 1) 10	Aplicar la función anónima \x -> x + 1 al argumento 10

En ocasiones, puede ser útil aplicar una función a sólo algunos de sus argumentos, para obtener una función parcialmente aplicada que espera los argumentos restantes:

```
1  suma : Nat → Nat → Nat
2  suma x y = x + y
3  suma10 : Nat → Nat
4  suma10 = suma 10
5  -- suma10 5 = 15
```

Aquí, `suma 10` devuelve una función que toma un `Nat` y le suma `10`. Esto es posible gracias a que en realidad todas las funciones en Agda toman un solo argumento y devuelven una función que espera el siguiente argumento, y así sucesivamente hasta que todos los argumentos han sido provistos. A este estilo de definir y aplicar funciones se le llama currying, y puede ser bastante útil para construir funciones complejas componiendo otras más simples.

En la siguiente sección, veremos más sobre cómo aprovechar este mecanismo y otros patrones útiles para manejar funciones con múltiples argumentos.

2.3. Argumentos y Parámetros

Las funciones en Agda se definen especificando tipos para cada uno de sus argumentos. Sin embargo, en ocasiones necesitamos funciones más genéricas que trabajen sobre argumentos de distintos tipos, o que reciban algunos argumentos en un orden diferente al sugerido por su tipo. Afortunadamente, Agda nos brinda varias herramientas para manejar estas situaciones de manera elegante y expresiva.

2.3.1. Cuantificación Universal

Consideremos la función `identity`, que simplemente devuelve su argumento sin modificarlo:

```
1  identity : (A : Set) → A → A
2  identity A x = x
```

Observemos que `identity` en realidad recibe dos argumentos: un tipo `A` y un valor `x` de tipo `A`, y retorna un resultado del mismo tipo `A`. El primer argumento es un tipo, no un valor. Este tipo de polimorfismo se conoce como cuantificación universal, y nos permite escribir funciones genéricas que funcionan para cualquier tipo.

El tipo `(A : Set) → A → A` puede interpretarse como: "Para todo tipo `A`, toma un valor de tipo `A` y devuelve un valor de tipo `A`".

Veamos otros ejemplos de funciones polimórficas:

```
1  constante : {A B : Set} → A → B → A
```

```
2 constante x _ = x
3 flip : {A B C : Set} → (A → B → C) → B → A → C
4 flip f y x = f x y
```

La función `constante` recibe un valor de tipo `A` y un valor de tipo `B`, y siempre retorna el primero, ignorando el segundo.

Por otro lado, `flip` es más interesante: toma una función `f` de dos argumentos, y devuelve una nueva función que espera esos mismos argumentos pero en el orden opuesto. Puede ser muy útil para aplicar parcialmente funciones a sus argumentos en un orden distinto al usual.

2.3.2. Argumentos Implícitos



En Agda, los argumentos encerrados entre llaves `{ }` en lugar de paréntesis `( )` se conocen como argumentos implícitos, y reciben un tratamiento especial.

Los argumentos implícitos son aquellos que Agda puede inferir a partir del contexto y los tipos de los demás argumentos, en lugar de ser provistos explícitamente. Al invocar una función con argumentos implícitos, podemos omitir esos argumentos y dejar que Agda los deduzca por sí mismo.

Los cuantificadores universales `(A : Set)` suelen dejarse implícitos, ya que generalmente no queremos proporcionar tipos explícitamente al llamar una función, sino dejar que el sistema de tipos los infiera por nosotros. Así, el tipo de `identity` quedaría:

```
1 identity : {A : Set} → A → A
```

Esto nos permite llamar a `identity` de la siguiente forma:

```
1 num : Nat  -- Agda infiere A = Nat
2 num = identity 42
3 booleano : Bool -- Agda infiere A = Bool
4 booleano = identity true
```

Sin tener que especificar manualmente el tipo `A` en cada caso, lo cual resultaría tedioso.

Sin embargo, en algunas situaciones poco comunes, puede ser necesario proporcionar un argumento implícito explícitamente, para ayudar a Agda a inferir los demás tipos o resolver alguna ambigüedad. En esos casos, podemos usar llaves para dar el valor del argumento implícito:

```
1  nil : {A B : Set} → List (A × B) -- Agda no puede inferir A y B a partir de una lista vacía
2  nil {A} {B} = []
3  pares : List (Nat × Bool) -- Explícitamente dando A y B
4  pares = nil {Nat} {Bool}
5  nil : {A B : Set} → List (A × B) -- Agda no puede inferir A y B a partir de una lista vacía
```

Otro caso en el que los argumentos implícitos son útiles es para funciones cuyos argumentos siempre pueden deducirse a partir de otros. Por ejemplo, consideremos esta definición alternativa de listas que almacena internamente su longitud como un número natural:

```
1  data ListaConLong (A : Set) : Nat → Set where
2    []  : ListaConLong A zero
3    _::_ : {n : Nat} → A → ListaConLong A n → ListaConLong A (suc n)
```

Notemos que `_::_` toma un argumento implícito de tipo `Nat` (la longitud de la cola de la lista), además del elemento `A` y la lista misma. Pero este número siempre puede inferirse a partir del segundo argumento. Al construir una lista, podemos usar el constructor `_::_` sin proporcionar explícitamente la longitud en cada caso:

```
1  unasDeNats : ListaConLong Nat 3
2  unasDeNats = 1 :: 1 :: 1 :: []
```

Agda infiere que las únicas longitudes posibles que hacen que los tipos coincidan son 2, 1 y 0 respectivamente. De esta forma, los argumentos implícitos nos permiten ocultar información redundante o que se pueda deducir fácilmente.

En general, los argumentos implícitos son una herramienta poderosa que ayuda a reducir la verbosidad y aumentar la genericidad de nuestro código, delegando trabajo al verificador de tipos.

2.3.3. Argumentos sin Usar

Otra característica interesante de Agda es que podemos escribir definiciones que toman argumentos, pero no los usan en absoluto para computar el resultado. Por ejemplo:

```
1  unoSiempreUno : {A : Set} → A → Nat
2  unoSiempreUno _ = 1
```

Aquí, `unoSiempreUno` toma un argumento de cualquier tipo `A`, pero simplemente lo descarta (usando el patrón comodín `_`) y siempre devuelve `1`, sin importar qué valor se le pase.

En el caso extremo, podemos tener funciones cuyos tipos mencionan argumentos que no aparecen en absoluto en el lado izquierdo de las ecuaciones. Por ejemplo, esta función:

```
1  cualquierNat : {A : Set} → Nat
2  cualquierNat {A} = 42
```

Toma un tipo `A` como argumento implícito, pero no lo usa de ninguna forma, ni siquiera con un comodín. A cambio, nos da la libertad de llamar a `cualquierNat` con cualquier tipo `A` que queramos (o en realidad con ninguno), pero siempre obtendremos `42` como resultado.

Aunque pueda parecer inútil a primera vista, este tipo de funciones en realidad tienen aplicaciones importantes en programación genérica y teoría de categorías. Son conocidas como "funciones constantes".

2.3.4. Parámetros libres

En contraste con los argumentos explícitos, en ocasiones se mencionan variables en el tipo de una función que no corresponden a ningún argumento. Considere el siguiente ejemplo:

```
1  data Vec (A : Set) : Nat → Set where
2    []  : Vec A zero
3    _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
4  head : {A : Set} {n : Nat} → Vec A (suc n) → A
5  head (x :: xs) = x
```

La variable `n` aparece en el tipo de `head`, pero no se pasa explícitamente como argumento. En cambio, el valor de `n` es determinado por el argumento que sí se pasa: Agda infiere que para que `x :: xs` tenga tipo `Vec A (suc n)`, necesariamente `xs` debe tener tipo `Vec A n`, y por lo tanto `n` debe ser la longitud de `xs`.

Es decir, la variable `n` no es realmente un argumento de la función `head`. Se trata de un parámetro libre, una variable cuyo valor queda determinado indirectamente por el tipo de los argumentos explícitos.

Los tipos dependientes (Tipos que dependen de valores, permitiendo especificaciones más precisas), que se estudiarán más adelante, utilizan parámetros de este tipo de forma extensiva para expresar restricciones o relaciones entre valores. Por ahora, lo importante es saber que no toda variable presente en el tipo de una función corresponde necesariamente a un argumento de la misma; algunas obtienen su valor del contexto.

Concepto	Explicación	Ejemplo
Argumentos Explícitos	Argumentos que se pasan explícitamente en la invocación de función y cuyos valores se utilizan en el cálculo	$f : A \to B \to C$
Argumentos Implícitos	Argumentos que pueden inferirse a partir del contexto y se pasan automáticamente sin mención explícita	$f : \{A : Set\} \to A \to A$
Parámetros Libres	Variables que aparecen en el tipo de una función pero no corresponden a ningún argumento explícito o implícito	$head : \{A : Set\}\ \{n : Nat\} \to Vec\ A\ (suc\ n) \to A$
Cuantificación Universal	Declaración de tipos genéricos que aceptan cualquier tipo como argumento	$id : (A : Set) \to A \to A$

2.4. Visibilidad de argumentos

Los argumentos de una función no siempre son simples entradas pasivas. En ocasiones, se desea permitir o restringir ciertas operaciones sobre ellos. Agda ofrece varias maneras de controlar la visibilidad o relevancia de un argumento.

Por defecto, los argumentos son visibles e irrelevantes. Esto significa que se puede ver su valor dentro del cuerpo de la función (no se borran en tiempo de compilación), pero el resultado de la función no depende de computación alguna sobre dicho valor (solo se usa para determinar tipos). Por ejemplo:

```
1  id : (A : Set) → A → A
2  id A x = x
```

Aquí, `A` es un argumento visible (se pasa explícitamente entre paréntesis) pero irrelevante: `id` no computa nada con `A`, solo lo usa para fijar el tipo del segundo argumento y del resultado.

En contraste, `x` es tanto visible como relevante: no solo determina el tipo del resultado, sino que su valor se usa directamente para computarlo.

Sin embargo, es posible marcar argumentos explícitamente como invisibles o irrelevantes usando los caracteres `@` y `@0`, respectivamente. Por ejemplo:

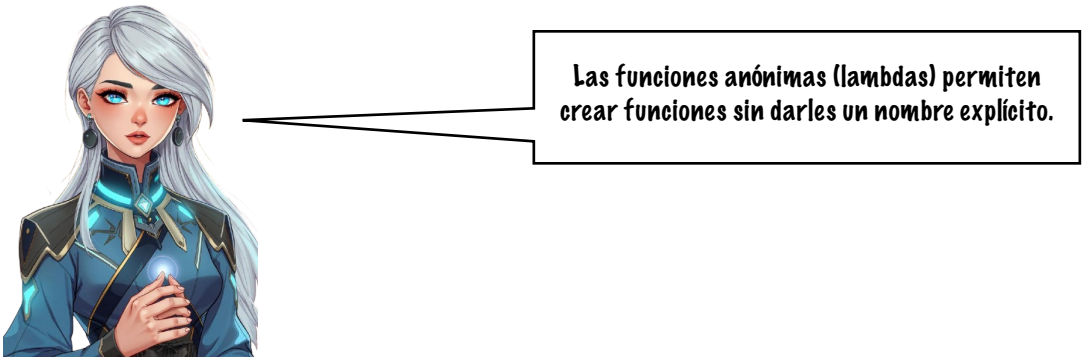
```
1  -- a es irrelevante
2  largo : (@a : Level) → Set a → Nat
3  largo _ Nat = 1
4  largo _ (A × B) = largo A + largo B
5  largo _ (A → B) = 2 * (1 + largo A + largo B)
6  -- a es invisible
7  elim : {@a : Level} {A : Set a} (x : A) → A
8  elim x = x
9  -- a es tanto invisible como irrelevante
10 bad : {@(irrĚ) a : Level} {A : Set a} → A
11 bad {A = A} = ?
12 -- a es irrelevante
13 largo : (@a : Level) → Set a → Nat
14 largo _ Nat = 1
15 largo _ (A × B) = largo A + largo B
16 largo _ (A → B) = 2 * (1 + largo A + largo B)
```

Un argumento irrelevante no puede utilizarse para computar valores; solo puede aparecer en tipos. Esto significa que cambiar ese argumento no altera el resultado.

Un argumento invisible no puede ser usado en pattern matching ni mencionado explícitamente en el cuerpo de una función. Se trata de una forma de borrar información.

Combinar ambas propiedades implica que el argumento no será guardado en la forma compilada final del programa; será eliminado completamente durante la fase de borrado de tipos.

Estas formas de visibilidad restringida son muy útiles para escribir código genérico eficiente, sin acarrear el costo de pasar argumentos de tipo y nivel en tiempo de ejecución.



2.5. Funciones anónimas

Así como los argumentos anónimos se escriben _, en Agda también es posible construir funciones anónimas (funciones "sin nombre") usando la notación λ. Una función anónima simple luce así:

```
1  \x → x + 1
```

Lo cual es equivalente a:

```
1  Suc
```

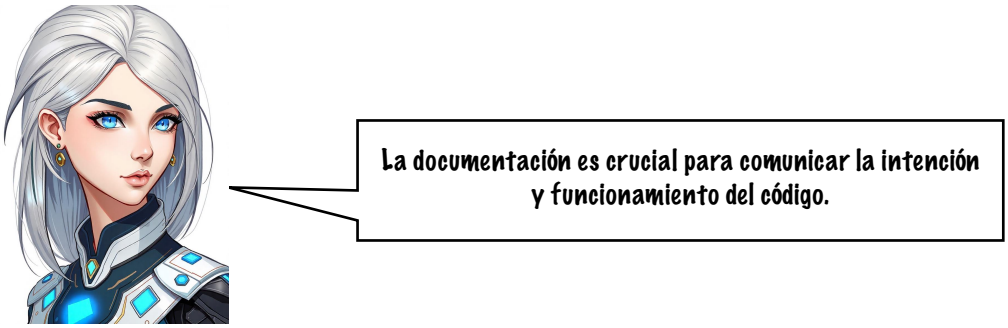
Las funciones lambda pueden tomar cualquier cantidad de argumentos, separando cada uno con un espacio después de la λ:

```
1  \x y → x + y
```

Las lambdas son útiles para evitar tener que nombrar una función que solo se va a usar localmente una vez. También pueden ser necesarias cuando se desea pasar una función como argumento a otra función (se verá más de esto al llegar a orden superior).

Ahora se conoce cómo declarar, aplicar y restringir funciones en Agda. También se han cubierto varias formas de aprovechar el sistema de tipos para hacer las funciones más genéricas, modulares y expresivas. En el siguiente capítulo, se verá cómo definir y manipular estructuras de datos recursivas de manera segura y elegante.

2.6. Documentación



En el mundo del desarrollo de software, la documentación juega un papel fundamental. Los programas escritos en Agda no son la excepción a esta regla. Debido a la naturaleza altamente modular y autocontenida de Agda, es crucial documentar minuciosamente el propósito y funcionamiento de cada definición para facilitar su comprensión y mantenimiento.

Agda ofrece dos enfoques principales para documentar el código: los comentarios informales y los bloques de documentación estructurada. Ambos tienen su lugar e importancia en el proceso de documentación.

2.6.1. Comentarios

Los comentarios son la forma más básica y universal de documentar un programa, y Agda soporta dos tipos:

1. Comentarios de una línea: Se inician con dos guiones `--` y se extienden hasta el final de la línea actual. Por ejemplo:

```
1  -- Este es un comentario de una sola línea
```

2. Comentarios multilínea: Comienzan con `{-` y terminan con `-}`. Pueden abarcar múltiples líneas, lo que los hace ideales para explicaciones más detalladas. Por ejemplo:

```
1  {- Este es un comentario
2     que abarca varias líneas.
3     Útil para explicaciones largas.
4  -}
```

Aunque los comentarios son ignorados por el compilador y no afectan la ejecución del programa, su propósito es comunicar claramente la intención y el funcionamiento del código a otros programadores (¡y a ti mismo en el futuro!). Algunas situaciones comunes en las que los comentarios son especialmente útiles incluyen:

Uso de Comentarios	Descripción
Explicación de Definiciones	Se utilizan comentarios para explicar el propósito y funcionamiento general de cada definición de función.
Clarificación de Pasos	Los comentarios se emplean para aclarar pasos complicados o no obvios en el código.
Anotaciones de Condiciones	Se añaden comentarios para indicar precondiciones, postcondiciones o invariantes asociados con el código.
Admisión de Limitaciones	Se utilizan comentarios para admitir limitaciones del código o posibles áreas de mejora.
Créditos a Fuentes Externas	Los comentarios se utilizan para dar crédito a fuentes externas cuando corresponda.

Veamos un ejemplo concreto de cómo los comentarios pueden mejorar la legibilidad del código:

```
1 -- Calcula la factorial de un número natural.
2 -- Precondición: el argumento debe ser no negativo.
3 -- Basado en "Functional Programming with Dependent Types".
4 fact : Nat → Nat
5 fact zero = suc zero
6 fact (suc n) = suc n * fact n
7 -- FIXME: No termina para números negativos.
```

Con la adición de estos comentarios, el propósito y el funcionamiento de la función `fact` quedan mucho más claros. Sabemos para qué sirve, qué espera, de dónde viene y qué limitación tiene. Si bien Agda no verifica la veracidad de los comentarios, incluirlos es una excelente práctica que facilita la comprensión y el mantenimiento del código.

2.6.2. Documentación estructurada

Cuando se trata de librerías y APIs más complejas, los comentarios informales pueden no ser suficientes. En estos casos, es beneficioso poder generar documentación separada del código fuente, en un formato más legible y accesible. Para satisfacer esta necesidad, Agda soporta bloques de documentación estructurada, similares a los ofrecidos por sistemas como Javadoc o Haddock.

Un bloque de documentación en Agda se escribe entre llaves precedidas de un pipe `{-|` y seguidas de un dash `-}`. Por ejemplo:

```
{-| La función identidad.
    Toma un argumento de cualquier tipo y lo devuelve sin cambios.
    __Ejemplos:__
    ```
    id Nat   = Nat
    id String = String
    id (1 + 1) = 2
    ```
-}
id : {A : Set} → A → A
id x = x
```

Al igual que los comentarios, los bloques de documentación son ignorados por el compilador. Sin embargo, su contenido puede ser extraído y renderizado en formato HTML u otros, permitiendo la inclusión de estas definiciones bajo demanda. Estos bloques actúan como una extensión de la interfaz del sistema de módulos.

Una documentación completa y bien estructurada debe incluir:

Componentes de la Documentación	Descripción
Descripción General	Breve descripción de lo que hace la definición en cuestión.
Propósito de Argumentos y Retorno	Explicación del propósito de cada argumento de entrada y del valor de retorno de la función.
Condiciones Adicionales	Posibles condiciones o restricciones que deben cumplirse para el correcto funcionamiento de la definición.
Ejemplos de Uso	Uno o más ejemplos que ilustran cómo usar la definición en diferentes situaciones.
Referencias Cruzadas	Enlaces o referencias a definiciones relacionadas para facilitar la comprensión y navegación del código.

Consideremos otro ejemplo ilustrativo:

```
{-| Determina si una lista está ordenada de forma ascendente.
    El predicado de comparación `p` determina el orden. Por defecto, se usa el orden natural `_≤_`.
    Ejemplos:
    ```
    sorted []   = true
    sorted (x :: [])   = true
    sorted (1 :: 2 :: 3 :: [])   = true
    sorted (1 :: 3 :: 2 :: [])   = false
    sorted (_≥_) (3 :: 2 :: 1 :: [])   = true
    sorted (\x y → x * 2 ≤ y) (2 :: 3 :: 5) = true
```
```

```
    ```
    Ver también: `Ordered`, `_≤_`, `_≥_`
-}
sorted : {A : Set} → (A → A → Bool) → List A → Bool
sorted _   [] = true
sorted _   [_] = true
sorted p (x :: y :: xs) = (p x y) ∧ sorted p (y :: xs)
```

Esta documentación deja en claro que `sorted` acepta un predicado de comparación opcional, explica el comportamiento por defecto y proporciona varios ejemplos ilustrativos de su uso. Además, hace referencia a otras definiciones relacionadas que pueden ser de interés para el lector.

Documentar meticulosamente el código Agda es una inversión que vale la pena, tanto a través de comentarios informales como de bloques de documentación estructurada. Un programa bien documentado es más fácil de entender, mantener y extender. Agda hace que incluir documentación sea trivial, por lo que no hay excusa para no hacerlo.

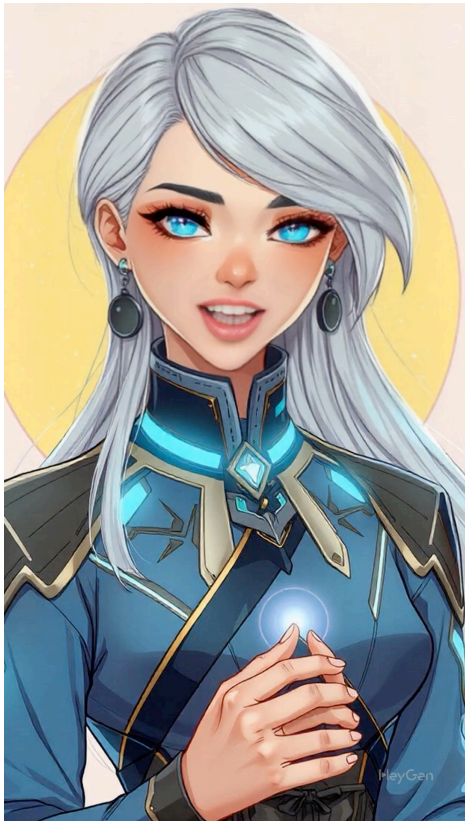
### Conclusiones del Capítulo

En este capítulo, hemos explorado los elementos fundamentales de Agda como lenguaje de programación funcional. Se ha abordado la sintaxis para definir tipos de datos, funciones y realizar coincidencias de patrones. Además, se han presentado características clave como el polimorfismo paramétrico, los argumentos implícitos y la recursión. También se han compartido prácticas recomendadas para documentar el código de manera efectiva. Con este conocimiento, los lectores están preparados para comenzar a escribir programas simples en Agda y comprender los principios básicos del paradigma de programación funcional pura. En los próximos capítulos, nos adentraremos en temas más avanzados como tipos dependientes, pruebas formales, estructuras infinitas y concurrencia, ampliando así las posibilidades y el poder expresivo de Agda.

## Ejercicios Propuestos

| | |
|---|---|
| 1 | Define un tipo de dato algebraico `Forma` que pueda representar círculos (con un radio), rectángulos (con ancho y alto), y triángulos (con base y altura). |
| 2 | Implementa una función `area` que calcule el área de una `Forma` usando pattern matching. |
| 3 | Define un tipo `Vector` que represente un vector de longitud fija, donde la longitud sea parte del tipo. Luego, implementa una función `concatenar` que tome dos vectores y devuelva su concatenación, asegurándote de que el tipo resultante refleje la suma de las longitudes. |
| 4 | Implementa una función polimórfica `identidad` que tome un tipo `A` como argumento implícito y un valor de tipo `A`, y devuelva ese mismo valor. |
| 5 | Escribe la documentación completa para la función `area` que implementaste en el ejercicio 2, incluyendo una descripción general, ejemplos de uso, y cualquier precondición o postcondición relevante. |
| 6 | Define un tipo de dato `ParNatural` que represente números naturales pares. Implementa funciones `esPar` y `siguiente` para este tipo, y proporciona pruebas de que `siguiente` siempre produce un número par. |
```

Capítulo 3: Estructuras de Datos Funcionales en Agda



Objetivos del capítulo

En este capítulo exploraremos las estructuras de datos funcionales en Agda, incluyendo listas y árboles. Aprenderás a definir y manipular listas recursivamente, así como a implementar mapas asociativos. También nos adentraremos en árboles binarios y árboles de búsqueda binaria, explorando cómo Agda ayuda a mantener invariantes. Además, introduciremos poderosas interfaces algebraicas como los monoides y los funtores, que permiten la composición y el mapeo genérico de funciones sobre estructuras de datos. Al final del capítulo, habrás adquirido una comprensión sólida de estas estructuras en Agda y estarás listo para crear tus propias implementaciones y razonar sobre la corrección de tus programas. Es un viaje emocionante a través de la elegancia y la expresividad de la programación funcional en Agda.

3.1. Introducción

En el fascinante paradigma de la programación funcional, nos encontramos con estructuras de datos que se alejan de las convenciones imperativas. En lugar de modificar directamente los datos, aplicamos funciones que toman estructuras inmutables y devuelven nuevas estructuras con los cambios deseados. Este enfoque nos sumerge en un estilo de programación más declarativo, componible y fácil de razonar.

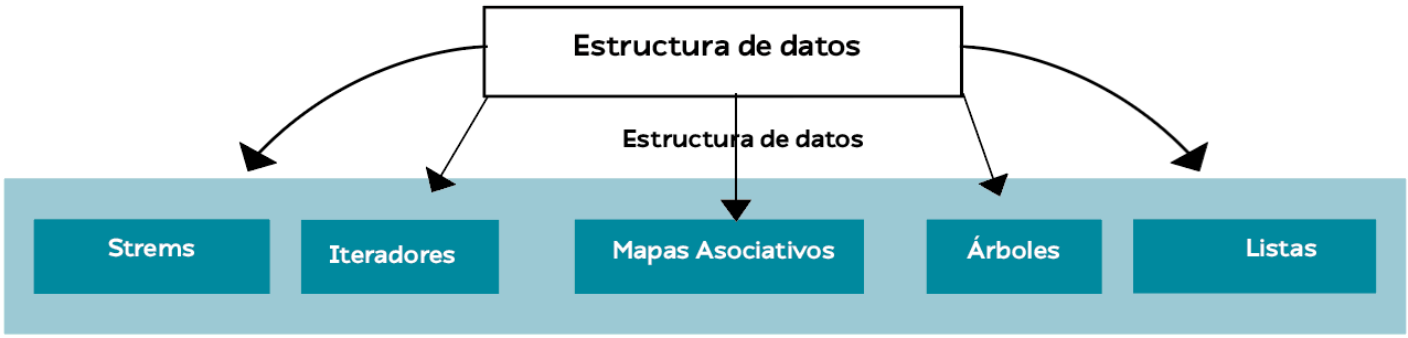


Imagen 5 - Estructuras de Datos Funcionales.
(Fuente: Propia)

Agda, como un lenguaje de programación funcional puro, abraza plenamente esta filosofía. Su expresivo sistema de tipos nos permite codificar estructuras de datos precisas y capturar invariantes directamente en los tipos. En este capítulo, exploraremos cómo definir e interactuar con algunas de las estructuras de datos funcionales fundamentales en Agda.

Comenzaremos nuestro viaje con las listas, uno de los pilares de la programación funcional. Definiremos el tipo `List` de forma recursiva y mostraremos operaciones comunes como acceder a la cabeza y la cola, concatenar y mapear elementos. Luego, nos adentraremos en los mapas asociativos, implementados como listas de pares, y examinaremos las operaciones de búsqueda e inserción.

A continuación, exploraremos las estructuras de datos jerárquicas utilizando árboles. Definiremos un tipo general de árbol binario y escribiremos funciones que manipulan árboles de forma recursiva. También abordaremos árboles más especializados, como los árboles de búsqueda binaria, y veremos cómo Agda nos ayuda a mantener sus invariantes.

En la última parte de este capítulo, elevaremos nuestro nivel de abstracción. Presentaremos interfaces algebraicas poderosas que generalizan muchas estructuras de datos concretas. Primero, estudiaremos los monoides, que capturan la noción de un conjunto con una operación asociativa y un elemento identidad. Luego, exploraremos los funtores, que nos permiten mapear funciones sobre estructuras, y los funtores aplicativos, que además nos permiten aplicar funciones envueltas en la estructura.

Discutiremos las motivaciones detrás de cada abstracción, proporcionaremos ejemplos concretos de uso y mostraremos cómo estas interfaces nos permiten escribir código genérico altamente reutilizable.

A lo largo de este capítulo, nos sumergiremos en el fascinante mundo de las estructuras de datos funcionales en Agda. Descubriremos la elegancia y la expresividad que este lenguaje nos ofrece para definir y manipular estas estructuras de manera segura y eficiente. ¿Estás listo para embarcarte en este emocionante viaje? ¡Acompáñame y exploremos juntos las maravillas de la programación funcional en Agda!

3.2. Listas: Los Pilares de la Programación Funcional

Imagina que las estructuras de datos son los bloques de construcción de tus programas. En el mundo de la programación funcional, las listas enlazadas son como los pilares fundamentales sobre los que se erige todo lo demás. Una lista puede verse como una secuencia de elementos, donde cada elemento está conectado al siguiente, formando una cadena. La belleza de las listas radica en su naturaleza recursiva: una lista es o bien vacía, o consiste en un elemento inicial (al que llamamos cabeza) seguido de otra lista (a la que nos referimos como cola). Esta definición recursiva captura a la perfección el estilo de "divide y vencerás" que caracteriza el procesamiento de datos en la programación funcional.

En Agda, podemos codificar directamente esta definición utilizando un tipo de datos inductivo. Veamos cómo se ve:

```
1  data List (A : Set) : Set where
2    []  : List A
3    _::_ : A → List A → List A
```

Aquí, `[]` representa la lista vacía (a veces llamada "nil"), mientras que `_::_` (pronunciado "cons") es un constructor que nos permite agregar un elemento al frente de una lista existente.

Por ejemplo, si queremos crear la lista `[1, 2, 3]`, podemos hacerlo de la siguiente manera:

```
1  1 :: 2 :: 3 :: []
```

Observa cómo la estructura anidada de la expresión refleja la naturaleza recursiva de la lista. Cada elemento se va agregando al frente de la lista anterior hasta llegar a la lista vacía.

3.2.1. Funciones Comunes sobre Listas

Ahora que tenemos nuestra definición de lista, ¡podemos empezar a jugar con ella! Vamos a escribir algunas funciones útiles para trabajar con listas y descubrir todo lo que podemos hacer

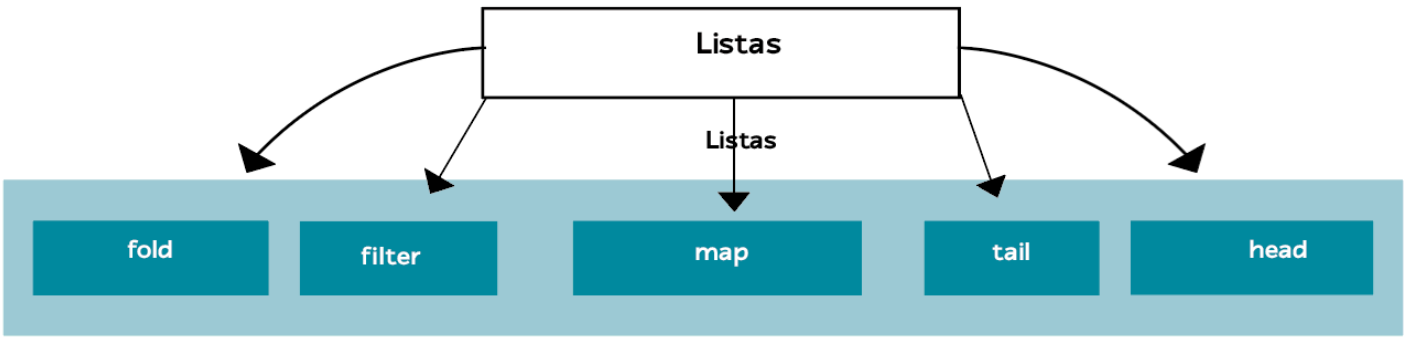


Imagen 6 - Operaciones Comunes sobre Listas.
(Fuente: Propia)

Por ejemplo, ¿qué tal si queremos obtener la cabeza de una lista? Es decir, queremos recuperar el primer elemento. Podemos definir una función `head` de la siguiente manera:

```
1 head : List A → Maybe A
2 head [] = nothing
3 head (x :: xs) = just x
```

Utilizamos la coincidencia de patrones para considerar los dos casos posibles: una lista vacía, para la cual devolvemos `nothing`, y una lista no vacía, para la cual devolvemos `just` seguido de la cabeza. El tipo `Maybe` nos permite manejar de forma segura la posibilidad de una lista vacía.

De manera similar, podemos definir una función `tail` que nos devuelva todos los elementos de una lista excepto el primero:

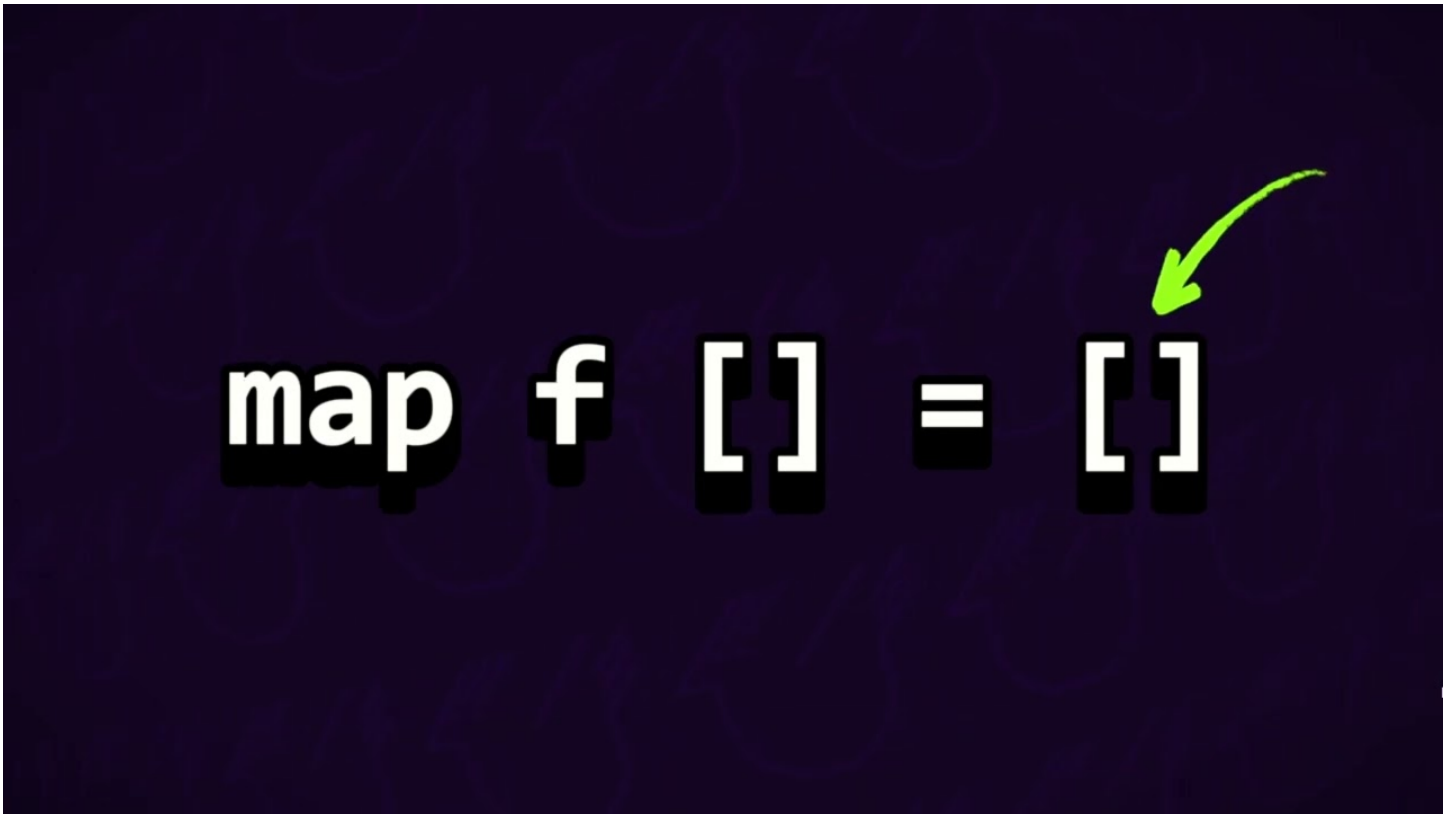
```
1 tail : List A → List A
2 tail [] = []
3 tail (x :: xs) = xs
```

Otra operación común es la concatenación, que nos permite combinar dos listas en una sola. Veamos cómo podemos implementarla:

```
1 _++_ : List A → List A → List A
2 [] ++ ys = ys
3 (x :: xs) ++ ys = x :: (xs ++ ys)
```

Fíjate en el caso recursivo, donde concatenamos la cola de la primera lista con la segunda lista completa, y luego volvemos a agregar la cabeza de la primera lista al frente del resultado.

Pero no nos detengamos ahí. ¿Qué tal si queremos aplicar una función a cada elemento de una lista? Para eso tenemos la función `map`, que nos permite transformar cada elemento mientras mantenemos la estructura general de la lista:



Video 02: Función Map

```
map : (A → B) → List A → List B
map f [] = []
map f (x :: xs) = f x :: map f xs
```

La función `map` toma una función `f` y una lista, y produce una nueva lista donde cada elemento ha sido transformado por `f`. Es como si recorriéramos la lista con una varita mágica, aplicando la función a cada elemento a medida que avanzamos.

Para darte un ejemplo concreto, imagina que queremos incrementar cada número en una lista. Podemos lograrlo utilizando `map` de la siguiente manera:

```
_ : map (_+ 1) (1 :: 2 :: 3 :: []) ≡ 2 :: 3 :: 4 :: []
_ = refl
```

Aquí, `_+ 1` es una función que suma 1 a su argumento. Al mapearla sobre la lista `[1, 2, 3]`, obtenemos una nueva lista `[2, 3, 4]`. La última línea `_ = refl` es una prueba de que las dos listas son iguales.

Pero `map` no es la única función poderosa que tenemos a nuestra disposición. Su primo cercano `filter` nos permite seleccionar solo aquellos elementos de una lista que cumplen con un determinado criterio. Veamos cómo se define:

```
filter : (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
... | true  = x :: filter p xs
... | false = filter p xs
```

Aquí utilizamos una construcción `with` para hacer coincidir el resultado de aplicar el predicado `p` al elemento actual `x`. Si `p x` es `true`, mantenemos `x` en la lista resultante; si es `false`, lo descartamos.

Por ejemplo, si queremos seleccionar solo los números pares de una lista, podemos hacer lo siguiente:

```
isEven : Nat → Bool
isEven zero = true
isEven (suc zero) = false
isEven (suc (suc n)) = isEven n
_ : filter isEven (1 :: 2 :: 3 :: 4 :: 5 :: []) ≡ 2 :: 4 :: []
_ = refl
```

Definimos un predicado `isEven` que verifica si un número natural es par y luego lo utilizamos con `filter` para obtener solo los números pares de la lista.

3.2.2. Listas y Recursión

Al implementar funciones sobre listas, es natural seguir la estructura recursiva del propio tipo de datos. El caso base normalmente maneja la lista vacía, mientras que el caso recursivo descompone la lista en cabeza y cola, hace la llamada recursiva en la cola y luego recombina los resultados.

Patrón de Recursión	Descripción	Ejemplo
Caso Base	Maneja el caso más simple, a menudo una estructura vacía	length [] = 0
Caso Recursivo	Descompone la estructura en una parte y hace una llamada recursiva	length (x :: xs) = 1 + length xs
Combinar Resultados	Combina el resultado de la llamada recursiva con la parte actual	sum (x :: xs) = x + sum xs
Argumento Acumulador	Pasa un argumento adicional para acumular el resultado	rev xs = rev' xs [] where rev' [] acc = acc rev' (x::xs) acc = rev' xs (x::acc)

Este patrón de recursión estructural es tan común que Agda nos proporciona un combinador de recursión general para listas llamado `foldr`:

```
foldr : {A B : Set} → (A → B → B) → B → List A → B
foldr f e [] = e
foldr f e (x :: xs) = f x (foldr f e xs)
```

`foldr` toma una función binaria `f`, un caso base `e` y una lista. Si la lista está vacía, simplemente devuelve `e`. Si no, aplica `f` a la cabeza de la lista y al resultado de llamar recursivamente a `foldr` en la cola.

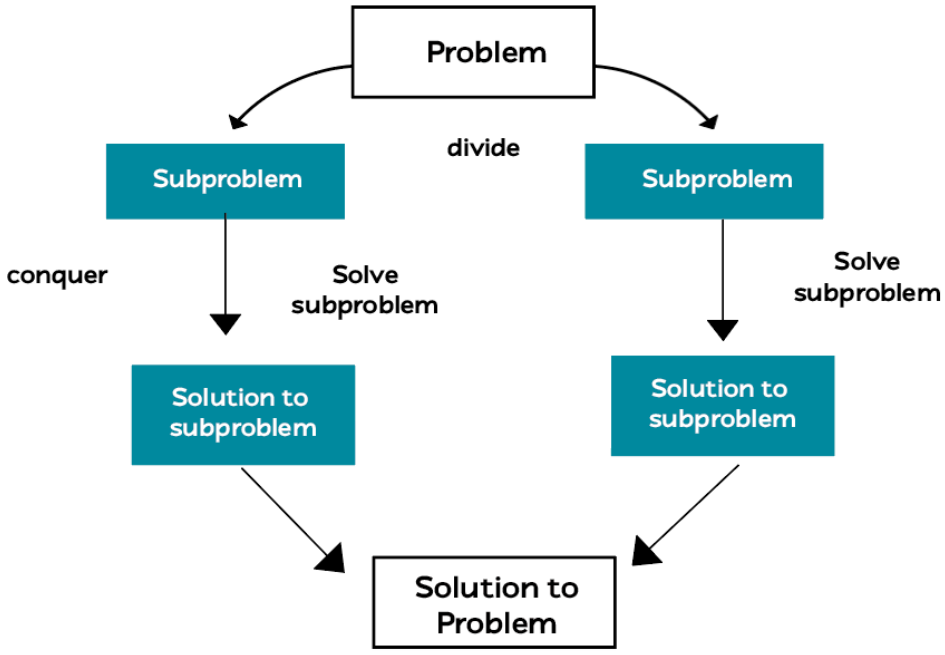


Imagen 7 - Dividir y Conquistar: La Estrategia Recursiva.
(Fuente: Khan Academy)

Muchas funciones pueden expresarse en términos de `foldr`. Por ejemplo, así es como podríamos sumar una lista de números:

```
1  sum : List Nat → Nat
2  sum = foldr _+_ 0
```

Aquí, la función binaria es la adición `_+_` y el caso base es `0`. De manera similar, podríamos implementar el producto, la concatenación o cualquier otra operación que pueda formularse en términos de combinar la cabeza con el resultado recursivo sobre la cola.

Este estilo de programación basado en combinadores como `foldr` se conoce como "programación de orden superior". En lugar de escribir recursión explícita, definimos y componemos funciones que encapsulan patrones recursivos comunes. Esto nos lleva a un código más conciso, menos propenso a errores y más fácil de razonar.

3.2.3. Eficiencia y Evaluación

Si bien las funciones recursivas sobre listas son expresivas y fáciles de escribir, es importante tener en cuenta que no siempre son las más eficientes.



Imagen 8 – Representación Visual de la Evaluación Perezosa.
(Fuente: Info Capital Humano)

Por ejemplo, consideremos el problema de encontrar la longitud de una lista:

```
1  length : {A : Set} → List A → Nat
2  length [] = 0
3  length (x :: xs) = 1 + length xs
```

Esta definición es clara y correcta, pero tiene un problema de eficiencia. Para procesar una lista de longitud `n`, hace `n` llamadas recursivas, cada una de las cuales devuelve `1` para que pueda sumarse. Esto toma un tiempo `O(n)` y un espacio `O(n)`, debido a que todas esas llamadas recursivas necesitan ser apiladas antes de que puedan devolverse y sumarse.

Una mejor manera sería usar un parámetro acumulador para realizar el seguimiento del recuento actual:

```
1  length : {A : Set} → List A → Nat
2  length xs = go xs 0
3    where
4    go : List A → Nat → Nat
5    go [] n = n
6    go (x :: xs) n = go xs (1 + n)
```

Ahora, en lugar de hacer `n` llamadas recursivas y luego desenrollar la pila, simplemente contamos a medida que avanzamos. Esto todavía toma un tiempo `O(n)`, pero solo un espacio `O(1)` ya que no hay llamadas recursivas pendientes - cada llamada actualiza directamente el acumulador `n` y luego se devuelve.

Esta técnica se llama recursión de cola, y es un patrón común para convertir funciones recursivas lineales en constantes espaciales. Agda actualmente no realiza la eliminación de llamadas de cola automáticamente, por lo que la transformación debe hacerse a mano.

Sin embargo, hay una compensación. La versión con acumulador es más eficiente, pero también más complicada de escribir y razonar. Se pierde la correspondencia directa entre la estructura del programa y la estructura recursiva de las listas.

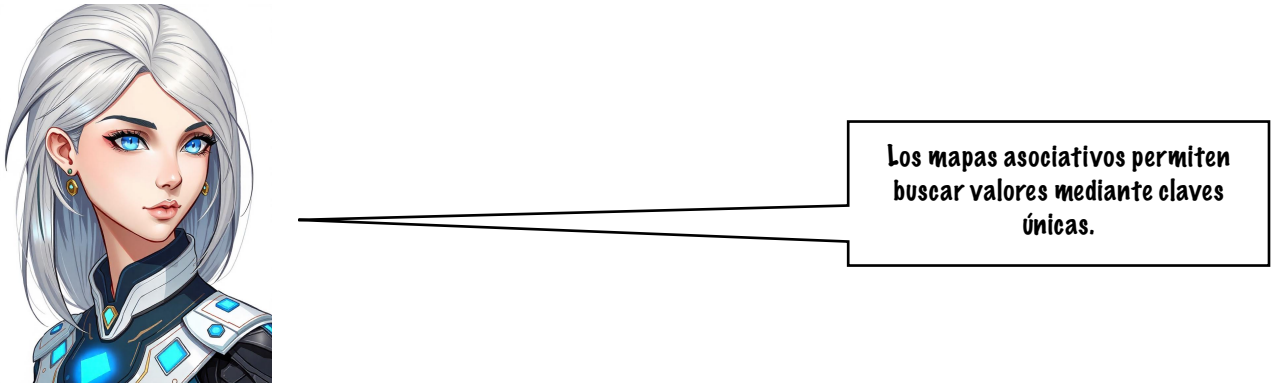
¿Hay una manera de recuperar el estilo recursivo simple pero mantener buenos comportamientos asintóticos? Resulta que sí, si estamos dispuestos a cambiar la representación de lista subyacente. En lugar de listas enlazadas puramente funcionales, podemos usar estructuras de datos persistentes más sofisticadas como árboles de Braun o árboles 2-3 hiperinmónicos que nos dan acceso `O(log n)` o incluso `O(1)` a la cabeza y la cola.

Con tales estructuras, los elegantes patrones recursivos que vimos anteriormente recuperan automáticamente la eficiencia. No hay necesidad de acumuladores explícitos u otras complicaciones. La eficiencia proviene de las estructuras de datos mismas.

Esta es una instancia de un principio más general en la programación funcional: muchos problemas se pueden resolver elegantemente eligiendo las abstracciones adecuadas. Ajustar las estructuras de datos y las funciones en armonía entre sí nos lleva a soluciones que son a la vez expresivas y eficientes.

A medida que avanzamos en nuestro estudio de las estructuras de datos funcionales en Agda, mantengamos este principio en mente. Busquemos formas de combinar la elegancia del estilo recursivo con la eficiencia de las estructuras de datos avanzadas. Con las herramientas y técnicas adecuadas, podemos crear programas que no solo son un placer de escribir, sino que también se ejecutan de manera óptima.

3.3. Mapas Asociativos



Los mapas asociativos, también conocidos como diccionarios o arrays asociativos, son otra estructura de datos fundamental. Son especialmente útiles para buscar valores de manera eficiente utilizando una clave única. En esencia, un mapa asocia cada clave a su valor correspondiente.

En un entorno de programación funcional puro, los mapas generalmente se representan como listas de pares clave-valor:

```
1  Map : Set → Set → Set
2  Map A B = List (A × B)
```

Aquí, `A` representa el tipo de las claves y `B` el tipo de los valores. Usamos el tipo producto `A × B` para modelar los pares.

Las operaciones principales que queremos soportar en los mapas son la búsqueda y la inserción. La búsqueda debería devolver el valor asociado a una clave dada, si existe:

```
lookup : {A B : Set} → A → Map A B → Maybe B
lookup x [] = nothing
lookup x ((y , z) :: xys) with x =? Y
... | yes _ = just z
... | no _ = lookup x xys
```

Utilizamos `_=?_` para comparar las claves por igualdad. Si encontramos una coincidencia, devolvemos `just` el valor correspondiente envuelto en `Maybe`. Si no, buscamos recursivamente en el resto del mapa. Si llegamos al final de la lista (caso base), devolvemos `nothing`.

Por otro lado, la inserción debería agregar un nuevo par clave-valor al mapa, o reemplazar el valor de una clave existente:

```
insert : {A B : Set} → A → B → Map A B → Map A B
insert x y [] = (x , y) :: []
insert x y ((z , w) :: xys) with x =? z
... | yes _ = (x , y) :: xys
... | no _ = (z , w) :: insert x y xys
```

Si el mapa está vacío, simplemente devolvemos un nuevo mapa con el par dado. Si la clave ya existe (según lo determina `_=?_`), reemplazamos el valor asociado. Si no, buscamos recursivamente e insertamos al final.

Es importante tener en cuenta que esta estructura de lista simple no es la implementación más eficiente posible de los mapas: en el peor caso, la búsqueda toma un tiempo lineal en el tamaño del mapa. Para ciertas claves, podemos hacer mucho mejor usando árboles de búsqueda, tablas hash o incluso arrays puros.

Sin embargo, esta sencilla implementación basada en listas ilustra las ideas clave comunes a todos los mapas asociativos: indexar valores por claves, buscar claves usando igualdad e insertar actualizando el valor de una clave. La interfaz sigue siendo la misma sin importar la estructura de datos subyacente.

3.3.1. Uso de mapas

Veamos un ejemplo concreto de cómo usar el tipo `Map`. Supongamos que tenemos un mapa que asocia nombres de personas con sus edades:

```
ages : Map String Nat
ages =
  ("Alice" , 25) ::
  ("Bob" , 30) ::
  ("Charlie" , 35) ::
  []
```

Para encontrar la edad de "Bob", usamos `lookup`:

```
_ : lookup "Bob" ages ≡ just 30
_ = refl
```

Para agregar una nueva persona "Dora" con edad 40:

```
_ : insert "Dora" 40 ages ≡
  ("Dora" , 40) ::
  ("Alice" , 25) ::
  ("Bob" , 30) ::
  ("Charlie" , 35) ::
  []
_ = refl
```

La nueva entrada se inserta al principio del mapa, dejando las demás entradas sin cambios.

También podemos combinar búsqueda e inserción para actualizar edades:

```
updateAge : String → Nat → Map String Nat → Map String Nat
updateAge x y m with lookup x m
... | nothing = insert x y m
... | just _  = insert x y m
```

Si la persona `x` no está en el mapa, se inserta con la edad dada `y`. Si ya está, se reemplaza su edad con `y`. Por ejemplo, para actualizar la edad de "Alice" a 26:

```
_ : updateAge "Alice" 26 ages ≡
  ("Alice" , 26) ::
  ("Bob" , 30) ::
  ("Charlie" , 35) ::
  []
_ = refl
```

La entrada anterior para "Alice" se reemplaza con la nueva edad, mientras que las demás entradas permanecen igual.

3.3.2. Eficiencia y alternativas

Como mencionamos antes, la implementación de lista simple de mapas tiene algunas ineficiencias. Específicamente, las operaciones `lookup` e `insert` toman un tiempo `O(n)` en el peor caso, donde `n` es el número de entradas en el mapa.

Para ciertas claves de tipos, podemos usar estructuras de datos más especializadas para mejorar la eficiencia. Por ejemplo:

- Si las claves son comparables, podemos usar árboles de búsqueda binaria equilibrados como los árboles AVL o los árboles rojo-negros. Estos garantizan un tiempo `O(log n)` para la búsqueda y la inserción.
- Si las claves son hashables, podemos utilizar tablas hash, que en promedio proporcionan búsqueda e inserción en `O(1)` (aunque el peor caso sigue siendo `O(n)`).
- Si las claves son tipos enteros pequeños, podemos usar arrays directos con indexación perfecta en tiempo `O(1)`.

Agda tiene un buen soporte para definir y razonar sobre estas estructuras más eficientes. Por ejemplo, podemos definir árboles de búsqueda binaria parametrizados sobre un tipo de orden:

```
1  data BST {A : Set} (_≤_ : A → A → Set) : Set where
2    leaf : BST _≤_
3    node : (l : BST _≤_) → (x : A) → (r : BST _≤_) →
4           ((y : A) → y ≤ x → y ∈ l) →
5           ((y : A) → x ≤ y → y ∈ r) → BST _≤_
```

Estructura de Datos	Complejidad de Búsqueda	Complejidad de Inserción	Complejidad de Borrado	Ventajas	Desventajas
Lista Enlazada	O(n)	O(1)	O(n)	Fácil de implementar, eficiente para insertar/borrar	Búsqueda ineficiente
Árbol de Búsqueda Binaria	O(log n)	O(log n)	O(log n)	Búsqueda, inserción y borrado eficientes	Peor caso O(n) si degenera a lista enlazada
Árbol AVL	O(log n)	O(log n)	O(log n)	Mantiene balance, garantiza complejidad O(log n)	Rotaciones complejas para mantener balance
Árbol Rojo-Negro	O(log n)	O(log n)	O(log n)	Mantiene balance, inserciones/ borrados más simples que AVL	Lógica más compleja que AVL

Aquí `_≤_` es un parámetro que representa la relación de orden sobre las claves de tipo `A`. Las condiciones en `node` aseguran que todas las claves en el subárbol izquierdo `l` son menores o iguales a `x`, mientras que todas las claves en el subárbol derecho `r` son mayores o iguales a `x`.

Usando esta estructura más sofisticada, la búsqueda se vuelve mucho más rápida:

```
1  _∈_ : {A : Set} {_≤_ : A → A → Set} → A → BST _≤_ → Set
2  x ∈ leaf = ⊥
3  x ∈ node l y r _ _ with x ≤? y | y ≤? x
4  ... | yes _ | yes _ = ⊤
5  ... | yes _ | no _  = x ∈ l
6  ... | no _  | yes _ = x ∈ r
7  ... | no _  | no _  = ⊥
```

Usando el orden `_≤_`, podemos descartar la mitad del árbol en cada paso, dando búsqueda en tiempo logarítmico.

Sin embargo, estas implementaciones más eficientes también vienen con una mayor complejidad y un potencial de errores. Mantener los invariantes de equilibrio de un árbol AVL o un árbol rojo-negro es notoriamente complicado. El hash perfecto necesario para arrays de acceso directo es muy frágil.

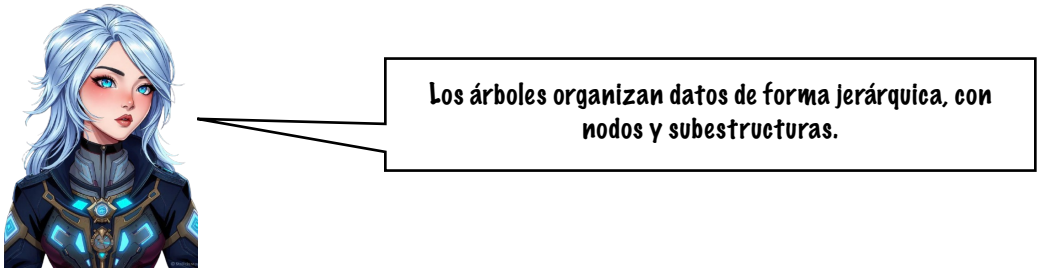
Afortunadamente, usando el rico sistema de tipos de Agda, podemos codificar y verificar estos invariantes para obtener la seguridad de la implementación de lista simple junto con la eficiencia de las estructuras de datos avanzadas. Exploraremos esto más a fondo al discutir la verificación formal (Proceso de probar matemáticamente la corrección de un programa) de programas más adelante en el libro.

Lo clave es apreciar que:

Aspecto Clave	Descripción
Diversidad de Representaciones	Reconocimiento de que diferentes representaciones de mapas son más adecuadas para diferentes casos de uso, permitiendo una selección óptima según las necesidades específicas.
Independencia de Interfaz	La interfaz de mapa es mayormente independiente de la representación seleccionada, lo que permite flexibilidad y modularidad en el diseño del programa.

Este tipo de modularidad es una característica definitoria de la programación funcional.

3.4. Árboles



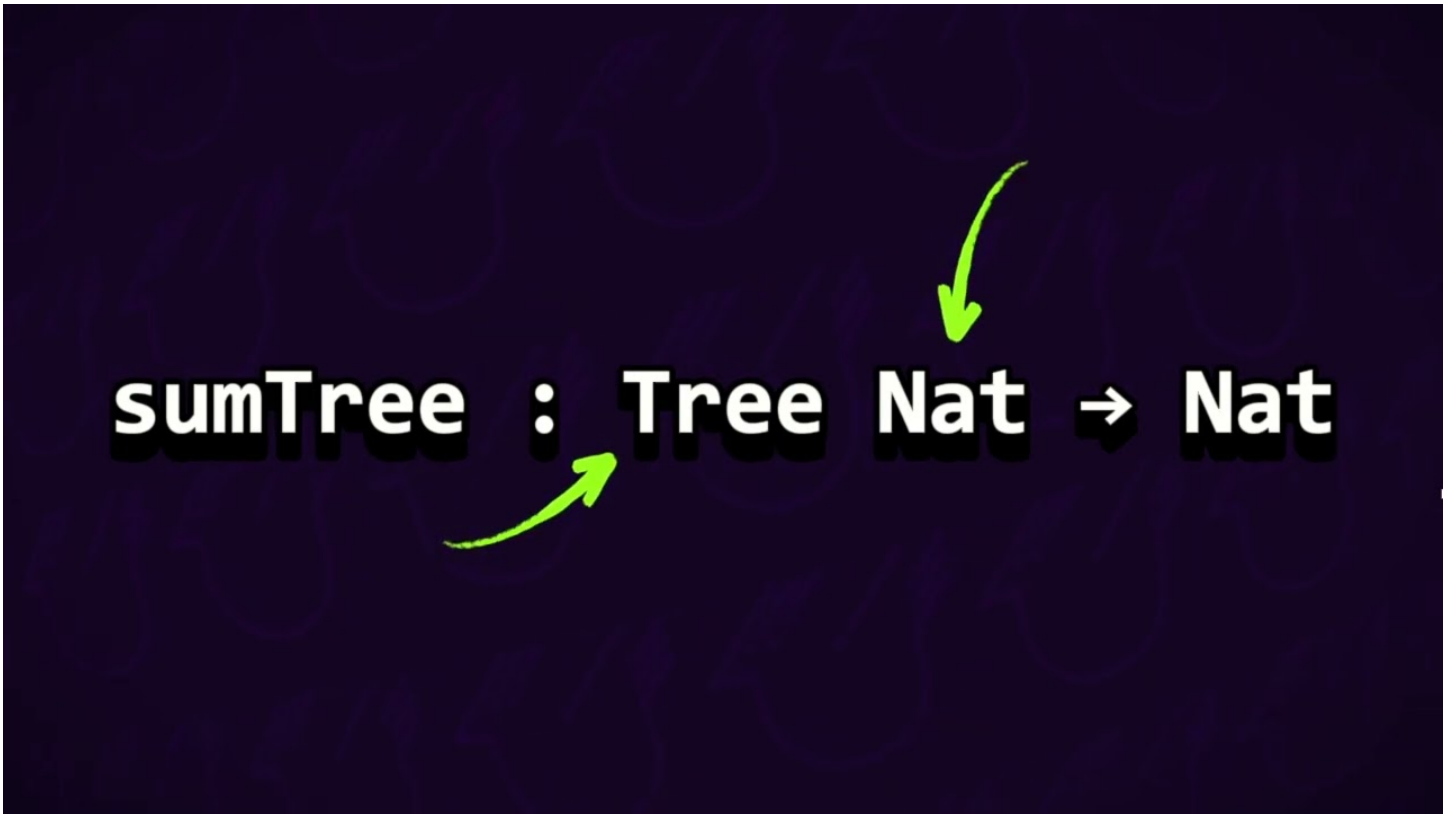
Los árboles son una familia de estructuras de datos recursivas que organizan los valores jerárquicamente. A diferencia de las listas, que son estrictamente lineales, los árboles pueden ramificarse y contener múltiples subestructuras. Esto los hace adecuados para representar datos anidados, búsquedas eficientes y muchas otras aplicaciones.

3.4.1. Árboles binarios

El tipo más fundamental de árbol es el árbol binario, donde cada nodo tiene cero, uno o dos "hijos":

```
1  data Tree (A : Set) : Set where
2    leaf : Tree A
3    node : Tree A → A → Tree A → Tree A
```

Aquí, `leaf` representa un árbol vacío, mientras que `node` construye un árbol a partir de un subárbol izquierdo, un valor en el nodo y un subárbol derecho.



Video 03: Suma de elementos de un árbol

Por ejemplo, podemos definir el siguiente árbol binario de números naturales:

```
1  myTree : Tree Nat
2  myTree =
3    Node
4      (node leaf 2 leaf)
5      5
6      (node
7        (node leaf 7 leaf)
8        11
9        leaf
10     )
```

Esto se visualizaría así:

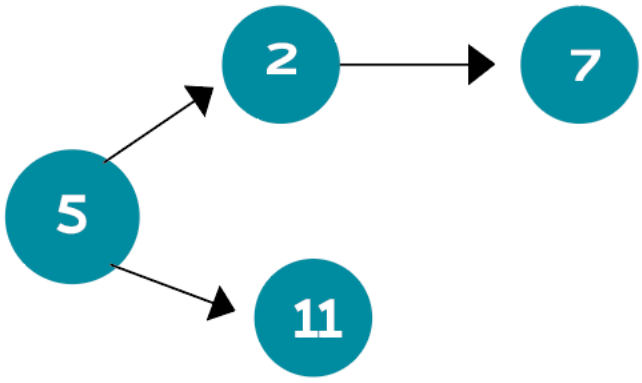


Imagen 9 – Ejemplo árbol en Agda.
(Fuente: Propia)

Al igual que con las listas, muchas funciones sobre árboles siguen la estructura recursiva. Por ejemplo, así calculamos la suma de todos los valores en un árbol:

```
1  sumTree : Tree Nat → Nat
2  sumTree leaf = 0
3  sumTree (node l x r) = sumTree l + x + sumTree r
```

El caso `leaf` es sencillo: un árbol vacío suma a cero. Para un nodo `node`, sumamos recursivamente los subárboles izquierdo y derecho y agregamos el valor en el nodo actual.

Otra operación clásica sobre árboles binarios es recorrerlos para producir una lista plana. Hay varios órdenes de recorrido, pero el más común es "en orden", donde visitamos el subárbol izquierdo, luego el nodo actual y finalmente el subárbol derecho:

```
1  inorder : Tree A → List A
2  inorder leaf = []
3  inorder (node l x r) = inorder l ++ [ x ] ++ inorder r
```

Este código refleja precisamente la especificación del recorrido en orden.

Por ejemplo:

```
1  _ : inorder myTree ≡ 2 :: 5 :: 7 :: 11 :: []
2  _ = refl
```

Esto aplana el árbol en una lista en el orden "izquierda-nodo-derecha".

3.4.2. Árboles genéricos

Los árboles binarios son un caso especial de una noción más general: un árbol donde cada nodo puede tener una cantidad arbitraria de subárboles. Esto se puede representar en Agda usando un parámetro de tipo adicional para la "aridad" de cada nodo:

```
1  data Tree {A : Set} (F : Set → Set) : Set where
2    node : F (Tree F) → Tree F
```

Aquí `F` es un "constructor de tipos" que describe la forma de cada nodo. Por ejemplo, para árboles binarios tendríamos:

```
1  BinF : Set → Set
2  BinF A = ⊤ ⊎ (A × A)
3  BinTree : Set → Set
4  BinTree = Tree BinF
```

Aquí `BinF A` es una suma que representa o un nodo hoja (`⊤`) o un par de subárboles (`A × A`). Envolver esto en `Tree` da el tipo completo de árboles binarios.

Otros constructores dan otros tipos de árboles. Por ejemplo:

```
1  data RoseF (A : Set) : Set where
2    rose : List A → RoseF A
3  RoseTree : Set → Set
4  RoseTree = Tree RoseF
```

Un "árbol rosa" tiene una lista arbitraria de subárboles en cada nodo.

Esta representación genérica de árboles es extremadamente versátil. Mediante la elección de diferentes constructores `F`, podemos modelar muchas estructuras de datos de árbol diferentes, todas usando las mismas funciones recursivas genéricas.

3.4.3. Árboles de búsqueda binaria

Una aplicación importante de los árboles binarios son los árboles de búsqueda binaria (BST).

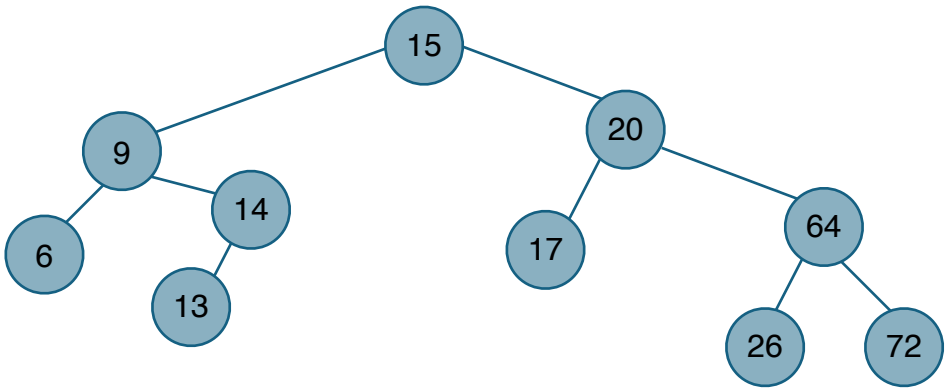


Imagen 10 - Árbol Binario de Búsqueda.
(Fuente: WikiMedia Commons)

En un BST, cada nodo tiene una clave, y las claves se ordenan de tal manera que todas las claves en el subárbol izquierdo son menores que la clave del nodo, mientras que todas las claves en el subárbol derecho son mayores.

Este invariante de ordenación se puede codificar directamente en el tipo:

```
1  data BST (A : Set) : Set where
2    leaf : BST A
3    node : (l : BST A) → (x : A) → (r : BST A) →
4           ((y : A) → y ∈ l → y ≤ x) →
5           ((y : A) → y ∈ r → x ≤ y) →
6           BST A
```

Aquí, los dos últimos argumentos a `node` son pruebas de que el árbol está ordenado: para cada `y` en el subárbol izquierdo, `y ≤ x`, y para cada `y` en el subárbol derecho, `x ≤ y`.

Con esta estructura más refinada en su lugar, las operaciones como la inserción y la búsqueda pueden aprovechar las garantías de ordenación para ser más eficientes. Por ejemplo, así buscamos un elemento en un BST:

```
1  _∈_ : A → BST A → Set
2  x ∈ leaf = ⊥
3  x ∈ node l y r pl pr with compare x y
4  ... | less    = x ∈ l
5  ... | equal   = ⊤
6  ... | greater = x ∈ r
```

Si el árbol es una `leaf`, no encontramos nada. Para un `node`, primero comparamos la clave buscada `x` con la clave `y` del nodo actual. Si es menor, buscamos recursivamente en el subárbol izquierdo; si es igual, hemos

encontrado el elemento; si es mayor, buscamos en el subárbol derecho. Esto reduce efectivamente el espacio de búsqueda a la mitad en cada paso.

Insertar un nuevo elemento en un BST sigue un patrón similar:

```
1 insert : A → BST A → BST A
2 insert x leaf = node leaf x leaf (λ _ () ) (λ _ ())
3 insert x (node l y r pl pr) with compare x y
4 ... | less    = node (insert x l) y r (λ z z∈il → pl z (∈-insert x l z∈il)) pr
5 ... | equal   = node l y r pl pr
6 ... | greater = node l y r (insert x r) pl (λ z z∈ir → pr z (∈-insert x r z∈ir))
```

Si el árbol es una `leaf`, creamos un nuevo `node` con el elemento `x`. Para un `node`, comparamos `x` con la clave `y` del nodo actual. Si `x` es menor, insertamos recursivamente en el subárbol izquierdo `l`; si es igual, dejamos el nodo sin cambios; si es mayor, insertamos en el subárbol derecho `r`.

También debemos actualizar las pruebas `pl` y `pr` en cada caso para mantener el invariante de ordenamiento del BST. Esto se hace envolviendo las pruebas existentes con una prueba `∈-insert` adicional de que la inserción preserva la propiedad de orden.

Estos ejemplos ilustran el poder del sistema de tipos de Agda para capturar invariantes complejas de estructuras de datos y razonar sobre la corrección de las operaciones. El convertir un árbol binario genérico en un BST no solo lo hace más eficiente para ciertas tareas, sino también más seguro, ya que se garantiza que se preservará la propiedad de orden.

3.5. Abstracciones algebraicas

A lo largo de nuestro recorrido por las estructuras de datos funcionales, hemos visto una variedad de ejemplos concretos como listas, mapas y árboles, cada uno con sus propias operaciones y características únicas. Sin embargo, si miramos más allá de los detalles específicos, podemos notar ciertos patrones recurrentes que emergen una y otra vez.

Estos patrones comunes son la base de las abstracciones algebraicas, un concepto poderoso que nos permite escribir código genérico capaz de funcionar con múltiples estructuras de datos, tanto las que ya conocemos como las que aún no hemos imaginado. Al aprovechar estas abstracciones, nuestro código se vuelve más mantenible, reutilizable y componible, lo cual es especialmente útil en un lenguaje como Agda, que se destaca por su expresividad y su potente sistema de tipos.

3.5.1. Monoides

Uno de los patrones más fundamentales es el monoide, que consiste en un conjunto `A` junto con una operación binaria `_<>_`, a veces llamada "append" o "combine". Para que `A` y `_<>_` formen un monoide, deben cumplir dos leyes.

Primero, la operación debe ser asociativa, lo que significa que:

```
(x <> y) <> z ≡ x <> (y <> z)
```

En otras palabras, el orden en que se aplica `_<>_` no importa: podemos combinar `x` e `y` primero y luego `z`, o `y` y `z` primero y luego `x`, y obtendremos el mismo resultado.

Segundo, debe existir un elemento "vacío" o "identidad", llamémoslo `ε`, tal que:

```
ε <> x ≡ x ≡ x <> ε
```

Esto quiere decir que combinar `ε` con cualquier otro elemento `x` simplemente nos devuelve `x`, sin cambios.

En Agda, podemos representar la interfaz de un monoide de la siguiente manera:

```
record Monoid (A : Set) : Set where
  field
    mempty : A
    _<>_   : A → A → A
  infixr 5 _<>_
```

Aquí, `mempty` representa el elemento identidad `ε`, mientras que `_<>_` es la operación de combinación.

Muchas estructuras familiares forman monoides de manera natural. Por ejemplo:

- Las listas, con la concatenación `_++_` y la lista vacía `[]` como identidad.
- Los números naturales, con la suma y `0`, o con el producto y `1`.
- Los booleanos, con la conjunción y `true`, o con la disyunción y `false`.
- Las funciones de tipo `A → A`, con la composición `_∘_` y la función identidad `id`.

La belleza de los monoides radica en que nos permiten razonar y operar de manera genérica en términos de las leyes que los definen, sin preocuparnos por los detalles específicos de la estructura subyacente. Un ejemplo de una operación útil que podemos definir para cualquier monoide es `mconcat`, que combina una lista entera de elementos:

```
mconcat : ∀ {A : Set} → Monoid A → List A → A
mconcat M = foldr _<>_ mempty
  where open Monoid M
```

Esta función simplemente pliega la lista utilizando la operación de combinación del monoide, con `mempty` como caso base. Funciona para listas, números, booleanos, funciones o cualquier otro tipo que forme un monoide, gracias al poder de la abstracción.

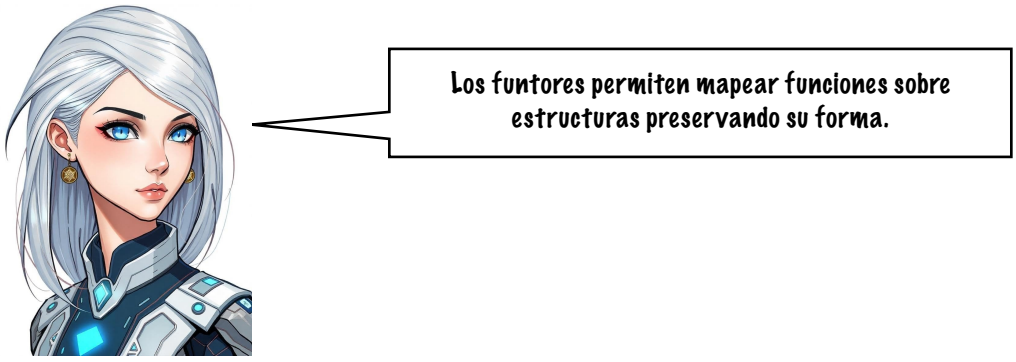
Pero eso no es todo. Los monoides también tienen la propiedad de componerse de manera elegante. Dados dos monoides `A` y `B`, su producto `A × B` también forma un monoide, con las operaciones aplicadas componente a componente:

```
instance
  MonoidProd : ∀ {A B : Set} → Monoid A → Monoid B → Monoid (A × B)
  mempty (MonoidProd MA MB) = mempty MA , mempty MB
  _<>_ (MonoidProd MA MB) (a , b) (c , d) = _<>_ MA a c , _<>_ MB b d
```

Algunos ejemplos interesantes incluyen pares de monoides numéricos como `(Sum Nat, Product Nat)`, que nos permiten llevar un registro simultáneo de la suma y el producto.

Esta capacidad de componer monoides es invaluable para construir abstracciones más grandes y complejas a partir de componentes simples y bien entendidos. Nos permite expresar estructuras y operaciones complicadas de una manera concisa y modular.

3.5.2. Functores



Otra pieza fundamental en nuestro arsenal de abstracciones es los functores. Un functor es cualquier tipo `F` que soporta una operación `map`, la cual nos permite transformar los valores "dentro" de una estructura `F` aplicando una función:

```
record Functor (F : Set → Set₁) : Set₁ where
  infixl 4 _<$>_
  field
    map : ∀ {A B : Set} → (A → B) → F A → F B
  _<$>_ = map
```

La idea central es que `map` nos permite "levantar" cualquier función "regular" `g : A → B` a una función `map g : F A → F B` que opera sobre estructuras `F`.

Muchos tipos comunes forman funtores de manera natural. Por ejemplo:

- Las listas, donde `map` es la función habitual que aplica una función a cada elemento.
- `Maybe`, donde `map f (just x) = just (f x)`, y `map f nothing = nothing`.
- Los árboles, donde `map` aplica recursivamente una función a cada valor en los nodos.

Pero los functores no se limitan a tipos de datos concretos. Muchos "tipos de tipos" también forman functores. Por ejemplo, para cualquier tipo fijo `E`, el tipo `(E →_)` de funciones que toman un argumento de tipo `E` es un functor:

```
instance
  {E : Set} → Functor (E →_)
  map {{functor↦ {E = E}}} g h e = g (h e)
```

En este caso, `map` es precisamente la composición de funciones `_∘_`.

Para que un tipo sea un functor, `map` debe satisfacer algunas leyes. Primero, aplicar `map` a la función identidad debe ser equivalente a no hacer nada:

```
map id ≡ id
```

Segundo, `map` debe preservar la composición de funciones:

```
map (g ∘ h) ≡ (map g) ∘ (map h)
```

Estas leyes garantizan que `map` se comporte de una manera razonable y que los functores puedan componerse arbitrariamente.

Con los functores, podemos escribir código genérico que sea paramétrico en la elección del functor. Por ejemplo, usando solo `map`, podemos implementar una versión genérica de `mconcat`:

```
fmconcat : ∀ {A F} → Functor F → Monoid (F A) → List (F A) → F A
fmconcat F M = foldr _<>_ mempty
  where open Functor F
        open Monoid M
```

Esta función toma un functor `F`, un monoide para `F A`, y una lista de `F A`s, y los concatena genéricamente. Primero mapea `_<>_` sobre la lista y luego la reduce con `mconcat`. Funciona para listas de listas, listas de `Maybe`, listas de funciones o cualquier lista de estructuras functoriales para las que tengamos un monoide.

Esto demuestra la sinergia entre functores y monoides: los functores nos permiten manipular valores dentro de estructuras, mientras que los monoides nos permiten combinar esas estructuras. Juntos forman una base sólida para construir programas genéricos.

3.5.3. Funtores Aplicativos

Estrechamente relacionados con los functores regulares están los functores aplicativos, o simplemente aplicativos.

Los aplicativos son como functores potenciados: no solo nos permiten aplicar funciones a valores dentro de un contexto, sino que además nos permiten aplicar funciones que ya están dentro del contexto a valores que también están en él.

Formalmente, un aplicativo es cualquier tipo `F` con dos operaciones:

```
record Applicative (F : Set → Set) : Set₁ where
  infixl 4 _<*>_
  field
    pure : ∀ {A} → A → F A
    _<*>_ : ∀ {A B} → F (A → B) → F A → F B
```

Aquí, `pure` toma un valor `x` y lo envuelve en el contexto aplicativo `F`, mientras que `_<*>_` toma una función `g` ya envuelta en `F` y un argumento `x` también envuelto en `F`, y produce el resultado de aplicar `g` a `x`, aún dentro de `F`.

Todo aplicativo es también un functor, como se puede ver en la definición por defecto de `map` en términos de `pure` y `_<*>_`. Pero los aplicativos pueden hacer más que solo mapear.

Muchos tipos de datos forman aplicativos de manera natural, como `Maybe`, las listas y las funciones. Los aplicativos realmente brillan cuando se trata de combinar múltiples efectos o contextos. Por ejemplo, si tenemos dos valores `Maybe` y queremos aplicarles una función binaria pura:

```
_<+>_ : Maybe Nat → Maybe Nat → Maybe Nat
mx <+> my = pure _+_ <*> mx <*> my
```

Aquí, primero envolvemos la función `_+_` en el contexto `Maybe` usando `pure`, y luego la aplicamos a `mx` y `my` usando `_<*>_`. Si alguno de `mx` o `my` es `nothing`, el resultado será `nothing`; de lo contrario, obtendremos `just` la suma.

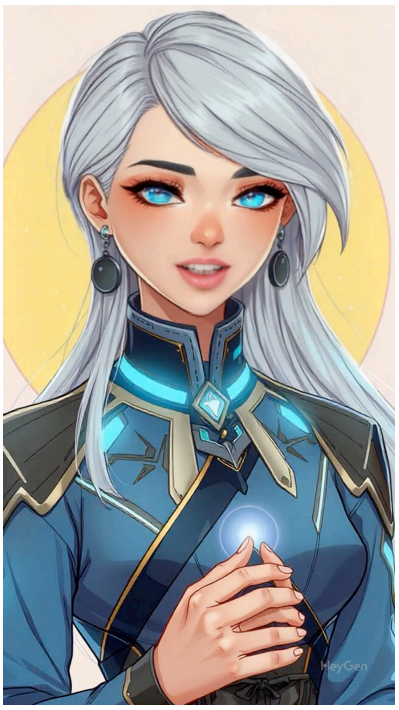
Compárese esto con utilizar solo `map`, que requiere anidar `case`s y mapear manualmente. Con los aplicativos, todo ocurre automáticamente bajo un mismo estilo de composición.

Los aplicativos también tienen sus propias leyes, que aseguran que se comporten de una manera consistente y componible. Con ellos, podemos secuenciar cálculos independientes, ignorar algunos resultados, acumular múltiples resultados o aplicar funciones bajo efectos, todo de forma genérica sobre cualquier estructura de datos que satisfaga las leyes de los aplicativos. Esto nos proporciona un marco expresivo para construir programas complejos a partir de partes más simples y reutilizables.

Conclusiones del Capítulo
En este capítulo, exploramos diversas estructuras de datos funcionales en Agda, desde listas hasta árboles y abstracciones algebraicas como monoides y funtores. Agda nos permite expresar conceptos complejos de manera clara y concisa, aprovechando su sistema de tipos para garantizar la corrección del código. Estas bases nos preparan para temas avanzados como programación genérica y verificación formal de programas en los próximos capítulos. Con este conocimiento, estaremos listos para aprovechar al máximo la programación funcional en Agda.

Ejercicios Propuestos	
1	Define tu propia versión del tipo de datos `List` en Agda. Implementa funciones básicas como `longitud`, `reversa`, y `mapear`.
2	Define un tipo de dato para árboles binarios. Implementa funciones para insertar un elemento, buscar un elemento, y realizar recorridos en orden, preorden y postorden.
3	Implementa un mapa asociativo simple usando una lista de pares. Proporciona funciones para insertar, buscar y eliminar elementos.
4	Define la clase de tipo `Functor` en Agda. Proporciona instancias para `Maybe`, `List`, y tu implementación de árboles binarios.
5	Define la clase de tipo `Monad` en Agda. Implementa las instancias para `Maybe` y `List`. Luego, utiliza la notación do para escribir una función que realice una secuencia de operaciones que puedan fallar (usando `Maybe`).
6	Define la clase de tipo `Applicative` en Agda. Proporciona instancias para `Maybe` y `List`. Demuestra cómo puedes usar el estilo aplicativo para combinar múltiples valores `Maybe` o `List`.
7	Implementa una estructura de datos de cola persistente en Agda. Asegúrate de que las operaciones de encolar y desencolar tengan una complejidad de tiempo amortizado O(1).

Capítulo 4: Sistemas de tipos avanzados



Objetivos del capítulo

En este capítulo, exploraremos las poderosas características del sistema de tipos de Agda. Nos sumergiremos en los tipos dependientes, que nos permiten codificar propiedades directamente en los tipos. Además, aprenderemos sobre los tipos coinductivos para representar estructuras infinitas y la capacidad de inferir automáticamente tipos para reducir la verbosidad. También abordaremos el polimorfismo paramétrico y ad-hoc para escribir código genérico, junto con el concepto de argumentos irrelevantes para mejorar el rendimiento. Al finalizar este capítulo, verás los tipos como un lenguaje preciso para especificar y restringir el comportamiento de tus programas, brindándote las herramientas necesarias para aprovechar al máximo el sistema de tipos de Agda en tu propio código.

4.1. Introducción

A lo largo de nuestra travesía por la programación funcional en Agda, hemos construido cimientos robustos al comprender los conceptos fundamentales y trabajar con estructuras de datos esenciales. Ahora, con ese conocimiento en nuestro haber, estamos preparados para sumergirnos en una de las facetas más fascinantes y potentes de Agda: su sistema de tipos.

El sistema de tipos de Agda es excepcionalmente expresivo, yendo más allá de lo que ofrecen los lenguajes de programación convencionales. No solo nos permite especificar la forma y estructura de los datos, sino también codificar propiedades, relaciones e invariantes profundos directamente en los tipos mismos. Esto eleva el rol del sistema de tipos, pasando de ser un mero mecanismo de clasificación de datos a convertirse en un verdadero lenguaje para expresar la lógica y el comportamiento esperado de nuestros programas.

En este capítulo, exploraremos varias características avanzadas del sistema de tipos de Agda:

- Tipos dependientes que pueden hacer referencia y depender de valores.

- Tipos coinductivos para representar estructuras de datos potencialmente infinitas.
- Inferencia de tipos para reducir la carga de anotaciones explícitas.
- Polimorfismo paramétrico para escribir código genérico y reutilizable.
- Argumentos irrelevantes que pueden ser ignorados durante la ejecución.

Dominar estos conceptos nos permitirá escribir código Agda más expresivo, preciso, genérico, modular y fácil de razonar. Daremos un gran salto, pasando de usar tipos principalmente para clasificación, a emplearlos como un sistema formal para especificar y restringir el comportamiento de nuestros programas.

Al final de este capítulo, verás los tipos bajo una nueva luz: no solo como una herramienta para detectar errores, sino como un lenguaje de precisión, abstracción y prueba. La clave para desbloquear la verdadera potencia de la programación funcional dependiente de tipos.

4.2. Tipos dependientes

La característica definitoria del sistema de tipos de Agda, que lo distingue de la mayoría de los lenguajes de programación, es la presencia de tipos dependientes. Mientras que en lenguajes tradicionales los tipos solo pueden referirse a otros tipos, en Agda los tipos pueden depender de términos.

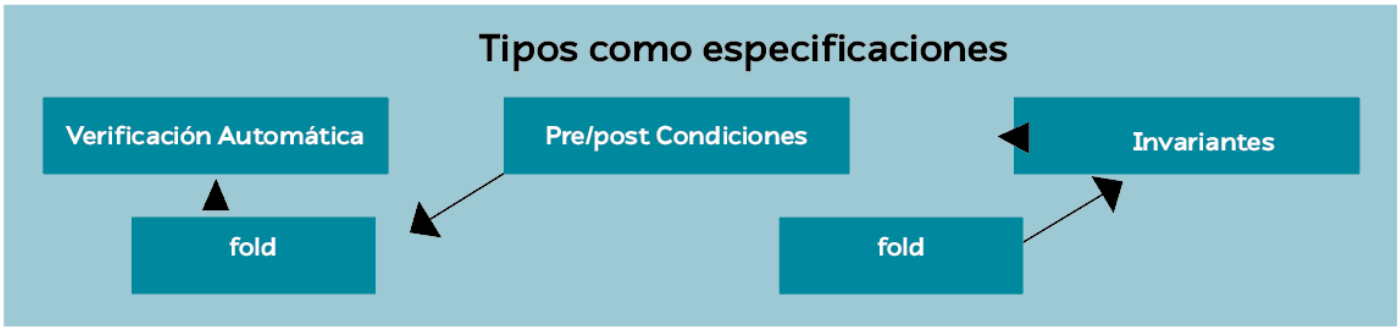


Imagen 11 - Especificaciones como Tipos en Agda.

4.2.1. Tipos indexados

El ejemplo canónico de un tipo dependiente son los llamados tipos indexados, donde los valores de un tipo están marcados o indexados por algún otro valor. Consideremos el tipo `Vec`, que representa vectores o listas de longitud fija:



Video 04: Definición del tipo Vec

```
1  data Vec (A : Set) : Nat → Set where
2    []  : Vec A zero
3    _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

Aquí, `Vec` toma dos argumentos: un tipo `A` y un natural `n`, y produce un tipo `Vec A n` representando vectores de elementos de tipo `A` con longitud exactamente `n`. El constructor `[]` produce el vector vacío, que tiene longitud `zero`, mientras el constructor `_::_` toma una cabeza de tipo `A` y una cola `Vec A n`, y produce un nuevo vector de longitud `suc n`.

Lo interesante es que la longitud del vector no es simplemente una propiedad o invariante extrínseco que se deba mantener por separado, sino que está inextricablemente entretejida en el tipo mismo del vector. Un `Vec Nat 5`, por ejemplo, no es solo una lista de números naturales; es una lista de números naturales que tiene exactamente 5 elementos, por definición.

Esto nos permite escribir funciones que son genéricas en la longitud del vector, pero al mismo tiempo tienen acceso a esa información para fines de especificación del tipo:

```
1  head : {A : Set} {n : Nat} → Vec A (suc n) → A
2  head (x :: xs) = x
3  tail : {A : Set} {n : Nat} → Vec A (suc n) → Vec A n
4  tail (x :: xs) = xs
```

La función `head` toma un `Vec A (suc n)` - es decir, un vector no vacío de longitud `suc n` - y devuelve su primer elemento. Si intentáramos aplicar `head` a un vector vacío `[]`, obtendríamos un error de tipos, ya que `[]` no coincide con el patrón `x :: xs`. De manera similar, `tail` devuelve todos excepto el primer elemento de un vector, y su tipo refleja el hecho de que el vector resultante tiene un elemento menos que la entrada.

4.2.2. Proposiciones como tipos

Más allá de simplemente indexar estructuras de datos con información adicional, los tipos dependientes también nos permiten codificar proposiciones o aserciones lógicas directamente en el sistema de tipos. La idea central es que podemos representar una proposición lógica como un tipo, donde los habitantes o términos de ese tipo corresponden a pruebas o justificaciones de la verdad de esa proposición.

Por ejemplo, consideremos la proposición "n es par", para algún número natural `n`. Podemos representar esto como un tipo dependiente `Par`:

```
1  data Par : Nat → Set where
2    cero : Par zero
3    dos  : {n : Nat} → Par n → Par (suc (suc n))
```

El constructor `cero` asevera que `zero` es par (tomando esto como una verdad axiomática), mientras que `dos` dice que si `n` es par, entonces `suc (suc n)` también debe ser par. Dada esta definición, podemos construir términos del tipo `Par n` para varios números naturales `n`:

```
1  parDos : Par 2
2  parDos = dos cero
3  parCuatro : Par 4
4  parCuatro = dos (dos cero)
```

Número	Término de Prueba
0	cero
2	dos cero
4	dos (dos cero)
6	dos (dos (dos cero))

8	dos (dos (dos (dos cero)))

Aquí, `parDos` y `parCuatro` no son simples valores booleanos afirmando que `2` y `4` son pares; son pruebas reales, construidas utilizando la lógica definida por el tipo `Par`, de que esas afirmaciones son verdaderas.

Ahora, si tuviéramos una función que requiere que su entrada sea par, podríamos expresar eso en su tipo:

```
1  dobleImpar : (n : Nat) → Par n → Nat
2  dobleImpar zero          cero    = zero
3  dobleImpar (suc (suc n)) (dos p) = suc (suc (dobleImpar n p))
```

Proposición	Tipo Dependiente	Explicación
n es par	Par n	Par es un predicado a nivel de tipo que expresa que n es un número par
x es primo	Prime x	Prime es un predicado que captura la noción de primalidad
xs está ordenada	Ordenada xs	Ordenada es un predicado sobre listas que expresa que xs está en orden
t es un árbol binario de búsqueda	BST t	BST es un predicado que expresa que t cumple las propiedades de árbol binario de búsqueda
f es inyectiva	Inyectiva f	Inyectiva captura la propiedad de inyectividad para funciones
x es una prueba de P	x : P	El tipo P puede verse como el conjunto de pruebas de la proposición P

La función `dobleImpar` toma un natural `n` y una prueba `p` de que `n` es par, y devuelve el doble de `n` (que se puede demostrar que siempre es par). Observemos cómo el tipo de la prueba `p` varía según el valor de `n` en cada caso: en el caso base es `cero`, mientras que en el paso inductivo es `dos p`, reflejando la estructura recursiva de la prueba.

Esta profunda correspondencia entre proposiciones y tipos, a veces llamada la correspondencia de Curry-Howard o el isomorfismo de proposiciones-como-tipos, es una de las ideas más potentes y fundamentales no solo en la teoría de tipos dependientes, sino en la lógica matemática y la informática teórica en general. Permite una hermosa simetría donde las especificaciones (expresadas como tipos) y las implementaciones (expresadas como términos) pueden mezclarse libremente, cada una sirviendo como un lenguaje para definir y restringir a la otra.

4.2.3. Ejemplos prácticos

A continuación, se presentan algunos ejemplos más de cómo podemos utilizar tipos dependientes para expresar propiedades e invariantes de manera práctica en el código Agda.

Ordenamiento de listas: Podemos definir una versión del tipo `List` que garantice que sus elementos estén siempre ordenados:

```
1  data OrdList : Set → Set where
2    nil  : {A : Set} → OrdList A
```

```
3    cons : {A : Set} → (x : A) → (xs : OrdList A) → ((y : A) → y ∈ xs → y ≥ x) → OrdList A
4  data OrdList : Set → Set where
```

Aquí, el constructor `cons` requiere una prueba de que todos los elementos `y` en la cola `xs` son mayores o iguales al nuevo elemento `x`. Entonces, por construcción, cualquier término de tipo `OrdList` necesariamente representa una lista ordenada.

Tablas bien tipadas: Otro ejemplo común es garantizar que las tablas o estructuras de datos similares a registros tengan el tipo correcto en cada campo:

```
1  record Tabla (Campos : List Set) : Set where
2    field
3      filas : List (Todos Campos)
4  data Todos : List Set → Set where
5    nil  : Todos []
6    cons : {A : Set} {As : List Set} → A → Todos As → Todos (A :: As)
```

Una `Tabla` está parametrizada por una lista de tipos `Campos`, y tiene un único campo `filas` que almacena una lista de registros, cada uno conteniendo un valor para cada uno de los tipos de `Campos` en orden. La función auxiliar `Todos` asegura que cada registro tenga exactamente los tipos correctos. Por lo tanto, algo del tipo `Tabla [ String , Int , Bool ]` es una tabla con tres columnas (un string, un entero y un booleano) donde cada fila sigue ese esquema exacto.

Listas con longitud: A menudo en la programación necesitamos no solo la estructura de una lista, sino también información sobre su longitud. Podemos codificar esto directamente en el tipo:

```
1  record ListaConLong (A : Set) : Nat → Set where
2    field
3      lista : List A
4      long  : longitud lista
```

Un valor de tipo `ListaConLong A n` es un registro con dos campos: la lista subyacente `lista`, y una prueba `long` de que la longitud de `lista` es exactamente `n`. El campo `long` utiliza una función `longitud` que calcula la longitud de una lista (que se define fácilmente por recursión). Entonces, el cliente de `ListaConLong` tiene acceso no solo a la estructura de la lista, sino también a su longitud, integrada en el tipo.

Estos son solo algunos ejemplos de las innumerables formas en que los tipos dependientes nos permiten codificar información más rica en los tipos. Al indexar tipos con valores y tratar las proposiciones como tipos habitados por pruebas, podemos dar definiciones precisas y matemáticamente exactas de las estructuras de datos y algoritmos, y tener esas definiciones verificadas durante la compilación. Esto eleva el nivel de especificación y abstracción en el que podemos programar, y abre un enorme abanico de posibilidades para crear software correcto, mantenible y reutilizable.

4.3. Tipos coinductivos e infinitos

Hasta ahora, los tipos que hemos visto - tanto básicos como dependientes - han sido tipos inductivos o finitos. Es decir, sus valores se construyen mediante un número finito de aplicaciones de constructores, y los algoritmos que operan sobre ellos eventualmente terminan descomponiendo esas estructuras finitas. Pero ¿qué sucede con las estructuras que podrían continuar indefinidamente, como flujos, procesos o interacciones infinitas?

Agda nos permite definir y manipular tales estructuras infinitas utilizando tipos coinductivos (Tipos que representan estructuras potencialmente infinitas o cíclicas). Mientras que los tipos inductivos se definen especificando cómo construir o descomponer sus valores finitos, los tipos coinductivos se definen especificando cómo observar o interactuar con sus valores potencialmente infinitos.

4.3.1. Flujos

El ejemplo estándar de un tipo coinductivo es el tipo de los flujos o streams:

```
1  record Stream (A : Set) : Set where
2    coinductive
3    field
4      head : A
5      tail : Stream A
```

Un `Stream` de valores de tipo `A` se define como un registro con dos campos: `head`, que es un valor de tipo `A`, y `tail`, que es en sí mismo otro `Stream A`. Observemos que no hay caso base - un `Stream` puede simplemente continuar proporcionando más elementos `head` y `tail` indefinidamente.

Podemos definir algunos streams básicos de la siguiente manera:

```
1  zeros : Stream Nat
2  zeros = record { head = 0 ; tail = zeros }
3  ones : Stream Nat
4  ones = record { head = 1 ; tail = ones }
5  nats : Stream Nat
6  nats = record { head = 0 ; tail = map suc nats }
```

Aquí, `zeros` es el stream constante de solo ceros, `ones` es el stream constante de solo unos, y `nats` es el stream de los números naturales comenzando desde cero. Todos estos streams son infinitos - conceptualmente, podemos pensarlos simplemente como funciones de los números naturales a los valores `A` (también conocidos como series de potencias formales en matemáticas).

Debido a que los streams son infinitos, los consumimos utilizando funciones que solo inspeccionan una parte finita de la estructura a la vez:

```
1  take : {A : Set} → (n : Nat) → Stream A → Vec A n
2  take zero    _  = []
```

```
take (suc n) xs = head xs :: take n (tail xs)
filter : {A : Set} → (A → Bool) → Stream A → Stream A
head (filter p? xs) with p? (head xs)
... | false = head (filter p? (tail xs))
... | true  = head xs
tail (filter p? xs) with p? (head xs)
... | false = tail (filter p? (tail xs))
... | true  = filter p? (tail xs)
```

La función `take n xs` devuelve un vector de los primeros `n` elementos del stream `xs`, mientras que `filter p? xs` devuelve un nuevo stream consistente solo de aquellos elementos de `xs` que satisfacen el predicado `p?`. En ambos casos, solo se inspecciona una parte finita del stream de entrada, y la evaluación se "detiene" en algún punto, a pesar de que el stream subyacente pueda ser infinito conceptualmente.

4.3.2. Procesos e interacciones

Los streams son solo el comienzo: los tipos coinductivos pueden utilizarse para modelar todo tipo de estructuras y procesos infinitos o abiertos. Algunos ejemplos interesantes incluyen:

Procesos de entrada/salida: Es posible representar programas interactivos que se comunican con el mundo exterior como un tipo coinductivo:

```
data Instr (A : Set) : Set where
  Put : String → Instr A
  Get : (String → Instr A) → Instr A
  Return : A → Instr A
record IO (A : Set) : Set where
  coinductive
  field
    run : Instr (IO A)
```

Una instrucción (`Instr`) puede ser `Put`, que emite una cadena al mundo; `Get`, que espera una cadena del mundo y continúa; o `Return`, que finaliza con un valor de tipo `A`. Un programa `IO A` es entonces una secuencia coinductiva de tales instrucciones que eventualmente devuelve un valor de tipo `A`. Esta definición captura la naturaleza potencialmente infinita de las interacciones de entrada/salida.

Árboles infinitos: Así como los streams generalizan las listas a estructuras lineales infinitas, es posible generalizar los árboles para permitir ramas infinitas:

```
record Tree (A : Set) : Set where
  coinductive
  field
    label : A
```

```
    child : List (Tree A)
```

Un árbol (`Tree A`) tiene un elemento `label` en su raíz y una lista (potencialmente infinita) de subárboles `child`. Esto nos permite definir estructuras como árboles binarios infinitos:

```
data Direction : Set where
  Left : Direction
  Right : Direction
record BinTree (A : Set) : Set where
  coinductive
  field
    label : A
    child : Direction → BinTree A
complete : {A : Set} → A → BinTree A
label (complete x) = x
child (complete x) Left = complete x
child (complete x) Right = complete x
```

El árbol binario `complete x` tiene el valor `x` en cada nodo y se bifurca infinitamente tanto a la izquierda como a la derecha.

Automorfismos: En teoría de categorías, un automorfismo es un isomorfismo de un objeto a sí mismo. Los automorfismos suelen codificar simetría o estructura algebraica en un objeto. Con tipos coinductivos, es posible representar automorfismos potencialmente infinitos:

```
record _≅_ (A B : Set) : Set where
  coinductive
  field
    to : A → B
    from : B → A
rotations : {A : Set} → Stream A → Stream A
head (rotations xs) = head xs
tail (rotations xs) = rotations (tail xs)
shift : {A : Set} → List A → List A
shift [] = []
shift (x :: xs) = xs ++ [ x ]
rotation : {A : Set} → (xs : Stream A) → xs ≅ rotations xs
to (rotation xs) = id
from (rotation xs) = id
```

Aquí, `rotations` toma un stream `xs` y produce el stream de todas sus rotaciones (obtenidas tomando repetidamente la cola del stream anterior). Luego, `rotation` produce un isomorfismo entre cualquier stream y el stream de sus rotaciones, capturando una especie de simetría cíclica infinita.

Los tipos coinductivos nos brindan una poderosa herramienta para razonar no solo sobre estructuras de datos infinitas, sino también sobre procesos, interacciones y relaciones infinitas. Al extender el dominio de los valores sobre los cuales los tipos pueden hablar para incluir comportamientos infinitos, los tipos coinductivos amplían significativamente el poder expresivo del sistema de tipos de Agda.

```
graph LR
A[Stream] --> B[head] & C[tail]
C --> D[head] & E[tail]
E --> F[head] & G[tail]
G --> H[head] & I[tail]
I --> J[...]
```

4.4. Inferencia de tipos



Una de las características clave que hace que Agda sea práctico de usar, a pesar de su extremadamente expresivo sistema de tipos, es el soporte para la inferencia de tipos. En muchos casos, Agda puede deducir automáticamente los tipos requeridos basándose en el contexto circundante y la estructura del programa, evitando que el programador tenga que anotar todo explícitamente.

Mecanismo de Inferencia	Explicación	Ejemplo
Argumentos Implícitos	Los argumentos de tipo entre llaves { } son inferidos automáticamente cuando es posible	id : {A : Set} → A → A
Inferencia Bidireccional	El algoritmo alterna entre inferencia (de términos a tipos) y chequeo (de tipos esperados a términos)	suc : Nat → Natsuc n = n + 1
Unificación	Encuentra sustituciones para variables de tipo que hacen que los tipos coincidan	app : {A B : Set} → (A → B) → A → Bapp f x = f x

Resolución de Restricciones	Resuelve restricciones de tipo generadas durante la inferencia usando instancias disponibles	show : {A : Set} ⦃ Show A ⦄ → A → String

4.4.1. Argumentos implícitos

Ya hemos visto una forma básica de inferencia de tipos en acción: los argumentos de tipo implícitos. Cualquier argumento escrito entre llaves `{ }` se trata como "implícito" y se espera que Agda lo infiera en lugar de ser proporcionado explícitamente en el sitio de llamada.

Por ejemplo, consideremos la función polimórfica identidad:

```
1  id : {A : Set} → A → A
2  id x = x
```

El tipo `A` es un argumento implícito, lo que significa que al llamar a `id`, no necesitamos especificar `A` explícitamente:

```
1  foo : Nat
2  foo = id 42
```

Aquí, Agda infiere que `A` debe ser `Nat` para que el tipo del argumento `42` coincida con el tipo declarado de `id`.

Es posible hacer un argumento aún más implícito utilizando `⦃ ⦄` en lugar de `{ }`:

```
1  _∘_ : ⦃A B C : Set⦄ → (B → C) → (A → B) → (A → C)
2  (f ∘ g) x = f (g x)
```

Ahora podemos escribir `(f ∘ g) x` sin proporcionar los tipos `A`, `B` y `C` en absoluto, ya que serán inferidos del contexto circundante.

Los argumentos implícitos son particularmente útiles para los tipos que aparecen repetidamente en una definición pero que rara vez o nunca varían, como los tipos de contenedores en estructuras de datos polimórficas. Nos permiten acortar lo verboso y mejorar la legibilidad de las definiciones y los sitios de llamada.

4.4.2. Resolución de restricciones

La inferencia de argumentos implícitos permite a Agda admitir un estilo de programación similar a las clases de tipo de Haskell, utilizando registros o módulos parametrizados como "restricciones" sobre los tipos.

Por ejemplo, podemos definir un registro `Eq` que representa la restricción de que un tipo admite la igualdad:

```
1  record Eq (A : Set) : Set where
2    field
3      _==_ : A → A → Bool
```

Y luego definir funciones genéricas que solo requieran que sus argumentos de tipo satisfagan `Eq`:

```
elem : ⦃Eq A⦄ → A → List A → Bool
elem x [] = false
elem x (y :: ys) = x == y ∨ elem x ys
```

Observemos cómo el tipo `A` para la función `elem` no aparece como un argumento explícito. En cambio, confiamos en que Agda lo resuelva a partir de los argumentos de valor `x` y `ys`, y luego compruebe que hay una instancia `Eq A` disponible en el ámbito.

Al igual que con las clases de tipo de Haskell, es posible definir instancias de estos registros/restricciones para tipos concretos:

```
instance
  EqNat : Eq Nat
  _==_ ⦃EqNat⦄ zero    zero    = true
  _==_ ⦃EqNat⦄ zero    (suc _) = false
  _==_ ⦃EqNat⦄ (suc _) zero    = false
  _==_ ⦃EqNat⦄ (suc n) (suc m) = n == m
  EqList : ⦃Eq A⦄ → Eq (List A)
  _==_ ⦃EqList eqA⦄ []       []       = true
  _==_ ⦃EqList eqA⦄ []       (y :: ys) = false
  _==_ ⦃EqList eqA⦄ (x :: xs) []       = false
  _==_ ⦃EqList eqA⦄ (x :: xs) (y :: ys) = x == y ∧ xs == ys
```

`EqNat` proporciona igualdad para los números naturales, mientras que `EqList` define la igualdad estructural para listas en términos de la igualdad de sus elementos. Entonces, por ejemplo, el tipo `Bool` que aparece en el tipo de `==` para `EqList` se resolverá a partir de la instancia `Eq A` que se pasa como argumento implícito `eqA`.

Esta combinación de restricciones como registros y resolución implícita de argumentos conduce a un estilo de programación genérico y modular, donde las definiciones que requieren capacidades particulares (como la igualdad) de sus tipos pueden expresar esos requisitos como restricciones, y luego las instancias para tipos concretos pueden ser mezcladas y combinadas de forma transparente.

4.4.3. Inferencia bidireccional

Además de la inferencia de argumentos implícitos, Agda emplea un algoritmo más general llamado "inferencia bidireccional" para deducir tantos tipos como sea posible.

La idea central es dividir la tarea de verificación de tipos en dos fases o modos: comprobación (o elaboración), donde se espera un tipo y se comprueba que un término tenga ese tipo; e inferencia (o restricción de tipos), donde se comienza con un término y se trata de calcular su tipo. Estos dos modos se alternan y alimentan entre sí: la comprobación de un término complejo implica inferir los tipos de sus subexpresiones, mientras que inferir el tipo de una aplicación de función puede requerir comprobar que los argumentos coincidan con los tipos esperados de los parámetros.

Por defecto, Agda intenta inferir el tipo de un término siempre que sea posible. Por ejemplo, en una definición como:

```
1  double : Nat → Nat
2  double x = x + x
```

Agda infiere el tipo del lado derecho `x + x` comenzando con la variable `x : Nat` y construyendo el tipo de resultado `Nat` usando las reglas de tipo para los números naturales y la adición. Esto se propaga hacia arriba, por lo que no necesitamos anotar explícitamente el tipo de `x` o del cuerpo de la función.

Sin embargo, la inferencia no siempre es posible, especialmente en presencia de tipos dependientes donde el espacio de posibles tipos es mucho más grande y complejo. En estas situaciones, podemos proporcionar "sugerencias" de tipos al comprobador, ya sea como anotaciones de tipos locales o argumentos explícitos. Por ejemplo:

```
1  successor : (n : Nat) → Vec A n → Vec A (suc n)
2  successor n xs = {! suc n!} :: xs
```

Aquí anotamos el primer argumento de `_::_` con una expresión `suc n` para ayudar a guiar la inferencia de tipos. De lo contrario, Agda no tiene forma de saber qué tipo se espera que tenga la longitud del vector resultante, ya que depende del valor de `n`.

En general, el algoritmo de inferencia bidireccional de Agda intenta minimizar la necesidad de anotaciones de tipos manteniendo un contexto global de restricciones de tipos y resolviendo las metas (tipos "faltantes") utilizando unificación y resolución de restricciones. Recurre a la comprobación siempre que no puede hacer una elección única de tipo a partir de la información disponible localmente.

4.4.4. Patrones de inferencia de tipos

A la luz del potente soporte de Agda para la inferencia de tipos, veamos algunos patrones y mejores prácticas comunes para aprovechar al máximo esta característica en el código dependiente de tipos.

Argumentos implícitos

Como regla general, cualquier argumento que mencione un tipo o una prueba, pero no se use directamente en el cuerpo de la definición es un buen candidato para ser implícito. Esto incluye la mayoría de los argumentos de tipo, índices de tipo y pruebas de propiedades.

Por ejemplo, al escribir funciones sobre vectores, generalmente querremos hacer que la longitud del vector sea implícita:

```
1  _++_ : ⦃A : Set⦄ → ⦃m n : Nat⦄ → Vec A m → Vec A n → Vec A (m + n)
2  [] ++ ys = ys
3  (x :: xs) ++ ys = x :: (xs ++ ys)
```

Aquí `A`, `m` y `n` son todos implícitos, ya que rara vez los proporcionaremos explícitamente y en cambio dejaremos que Agda los infiera de los vectores de entrada reales.

Encadenamiento de funciones

Al componer o encadenar múltiples funciones, es mejor poner las aplicaciones parciales antes que los argumentos explícitos para maximizar las oportunidades de inferencia:

```
1  proceso : String → Nat
2  proceso = length ∘ words ∘ trim
```

Si hubiéramos escrito `proceso s = length (words (trim s))`, Agda habría requerido anotaciones de tipo para `words` y `trim`, ya que no podría inferir el tipo intermedio `String`. Pero el estilo point-free `length ∘ words ∘ trim` encaja perfectamente con la inferencia bidireccional, ya que cada función produce exactamente el tipo que la siguiente espera.

Retroceso a explícito

En casos donde la inferencia se atasca o infiere el tipo incorrecto, siempre podemos proporcionar anotaciones explícitas para desambiguar o forzar un tipo particular. Esto suele ser necesario cuando hay múltiples restricciones o instancias en el scope que podrían aplicarse, o cuando necesitamos un tipo más específico que el que Agda inferiría por defecto.

```
1  mitad : Nat → Nat
2  mitad = div 2
3   where
4     div : (x y : Nat) → Nat
5     div = quot {Nat} -- inferencia elegiría Integer en su lugar
```

Sin la anotación `{Nat}`, Agda elegiría el tipo de división "predeterminado" que opera en `Integer` en lugar de `Nat`. Al proporcionar el argumento explícito, anulamos ese comportamiento predeterminado y forzamos el uso del `quot` que opera sobre naturales.

Utiliza argumentos implícitos para:

- Tipos y pruebas que se mencionan en tipos pero rara vez se usan explícitamente.
- Índices y parámetros que se pueden inferir fácilmente de los argumentos de valor.
- Restricciones y capacidades requeridas que normalmente se resuelven a partir del contexto.

Utiliza argumentos explícitos para:

- Valores y términos computacionalmente relevantes.
- Tipos o pruebas que no se pueden inferir o que se necesitan para guiar la inferencia.
- Desambiguar entre múltiples posibles restricciones o instancias.

En general, apunta a minimizar la necesidad de anotaciones explícitas al reestructurar las definiciones para aprovechar al máximo la inferencia bidireccional, pero no temas retroceder a explícito cuando sea necesario para la claridad o para forzar un tipo particular.

4.5. Polimorfismo

El polimorfismo es un aspecto clave en cualquier sistema de tipos práctico y expresivo. Nos permite escribir código genérico que funciona de manera uniforme sobre diversos tipos, maximizando la reutilización mientras preservamos sólidas garantías de corrección. Agda soporta varias formas de polimorfismo, cada una con sus propias ventajas y casos de uso.

Tipo de Polimorfismo	Explicación	Ejemplo en Agda
Paramétrico	Funciones que trabajan uniformemente sobre muchos tipos, sin poder distinguirlos	id : {A : Set} → A → A id x = x
Ad-hoc	Funciones que pueden tener diferentes implementaciones para diferentes tipos	_==_ : {A : Set} → A → A → Bool instance eqNat : {x y : ℕ} → x == y = primEqNat x y eqVec : {n : ℕ} {A : Set} → (eqA : A → A → Bool) → (xs ys : Vec A n) → xs == ys = primEqVec n eqA xs ys
De Subtipado	Funciones que trabajan sobre un tipo y sus subtipos	size : {A : Set} → {n : ℕ} → Vec A n → ℕ size {_} {n} _ = n
Dependiente	Funciones cuyos tipos dependen de los valores	replicate : {n : ℕ} → {A : Set} → A → Vec A n replicate {zero} _ = [] replicate {suc n} x = x :: replicate {n} x

A continuación, exploraremos cada uno de estos tipos de polimorfismo en detalle, viendo cómo nos permiten escribir código más genérico, modular y reutilizable.

4.5.1. Polimorfismo paramétrico

La forma más fundamental de polimorfismo en Agda es el polimorfismo paramétrico (Capacidad de escribir código que funciona con múltiples tipos de datos). Con él, las funciones o tipos toman otros tipos como parámetros. Esto les permite trabajar de manera uniforme sobre todos los posibles tipos que podrían sustituir a esos parámetros. Veamos algunos ejemplos:

```
1  id : {A : Set} → A → A
2  map : {A B : Set} → (A → B) → List A → List B
3  data Maybe (A : Set) : Set where
4    nothing : Maybe A
5    just    : A → Maybe A
```

Una propiedad clave del polimorfismo paramétrico es la parametricidad. Esto significa que las funciones paramétricamente polimórficas deben tratar sus argumentos de tipo de manera completamente genérica y abstracta, sin poder inspeccionarlos o desarmarlos de ninguna manera.

Esta propiedad tiene poderosas consecuencias. Por ejemplo, si consideramos el tipo `{A : Set} → A → A`, nos damos cuenta de que sólo puede tener un habitante: la función identidad. Cualquier otra implementación requeriría "saber" algo concreto sobre `A`, lo cual está prohibido por la parametricidad.

En general, los tipos de funciones paramétricamente polimórficas restringen mucho sus posibles implementaciones. Esto nos permite derivar "teoremas gratis" sobre ellas sólo a partir de sus tipos, sin siquiera mirar su código. Aquí radica el verdadero poder de la parametricidad.

A nivel práctico, el polimorfismo paramétrico promueve un estilo de programación basado en la composición de pequeñas abstracciones reusables. Podemos definir conceptos genéricos como funtores y mónadas (Estructura que encapsula cómputos y efectos secundarios de manera componible), y luego mezclarlos y combinarlos en programas más grandes que funcionan con muchos tipos de datos diferentes. La generalidad de los componentes polimórficos los hace ampliamente aplicables y reutilizables.

```
graph TD
  A{(A : Set) → F A} --> B(F Nat)
  A --> C(F Bool)
  A --> D(F String)
  A --> E(F (List Nat))
  A --> F(...)
```

4.5.2. Polimorfismo ad-hoc

Si bien el polimorfismo paramétrico es extremadamente potente, a veces necesitamos la flexibilidad de tratar ciertos tipos de manera especial o de restringir las posibles instancias de un tipo polimórfico. Aquí es donde entra en juego el polimorfismo ad-hoc (Comportamiento diferente de una función según el tipo de sus argumentos) en Agda.

La forma más común de polimorfismo ad-hoc en Agda es a través de registros o módulos que representan interfaces. Por ejemplo, podemos definir un registro para tipos que soportan igualdad:

```
record Eq (A : Set) : Set where
  field
    _==_ : A → A → Bool
```

Luego podemos escribir funciones genéricas que requieren que sus argumentos de tipo sean "igualables":

```
elem : ⦃ Eq A ⦄ → A → List A → Bool
elem x [] = false
elem x (y ∷ ys) = x == y ∨ elem x ys
```

La función `elem` usa la operación `_==_` que se resuelve implícitamente a partir de la restricción `Eq A` en el contexto. Podemos definir instancias de `Eq` para tipos específicos:

```
instance
  EqNat : Eq Nat
  _==_ ⦃ EqNat ⦄ = eqNat
  EqList : ⦃ Eq A ⦄ → Eq (List A)
  _==_ ⦃ EqList eqA ⦄ [] []             = true
  _==_ ⦃ EqList eqA ⦄ (x :: xs) (y :: ys) = x == y ∧ xs == ys
  _==_ ⦃ EqList eqA ⦄ _ _               = false
```

`EqNat` simplemente usa la igualdad incorporada `eqNat`, mientras que `EqList` define la igualdad de listas recursivamente en términos de la igualdad `_==_` de sus elementos, que se obtiene de la restricción `Eq A` pasada implícitamente.

Así logramos sobrecargar definiciones genéricas con implementaciones específicas para cada tipo, obteniendo lo mejor de ambos mundos: código genérico y modular, pero con la posibilidad de especialización donde sea necesario. Los diccionarios de métodos implícitos provistos por los registros garantizan que se use la implementación correcta en cada sitio.

4.5.3. Clases de tipos

Las clases de tipos son otra forma de estructurar el polimorfismo ad-hoc. Si bien Agda no las soporta directamente, podemos emularlas usando un patrón de codificación basado en módulos parametrizados.

La idea es definir una "clase de tipos" como un módulo parametrizado por algún tipo `A`:

```
module Eq (A : Set) where
  infix 4 _==_
  field
    _==_ : A → A → Bool
```

Luego, definimos las "instancias" abriendo este módulo con tipos específicos y brindando implementaciones para los campos requeridos:

```
open Eq Nat renaming (_==_ to eqNat)
  where
    _==_ = primEqNat
module ListEq (A : Set) ⦃ _ : Eq A ⦄ where
  open Eq (List A) public
    renaming (_==_ to eqList)
    where
      [] == []             = true
      (x :: xs) == (y :: ys) = x == y ∧ xs `eqList` ys
      _ == _               = false
  infix 4 _≈_
```

```
  _≈_ : List A → List A → Bool
  xs ≈ ys = xs `eqList` ys
```

Aquí abrimos `Eq Nat` y `Eq (List A)` (este último con una restricción `Eq A` implícita), proporcionamos definiciones concretas para `_==_`, y también definimos algunos operadores adicionales como `_≈_` en términos de la igualdad subyacente `eqList`.

Usar módulos parametrizados nos permite definir múltiples instancias de una interfaz para el mismo tipo. Por ejemplo, podríamos tener una segunda noción de igualdad de listas que ignore el orden:

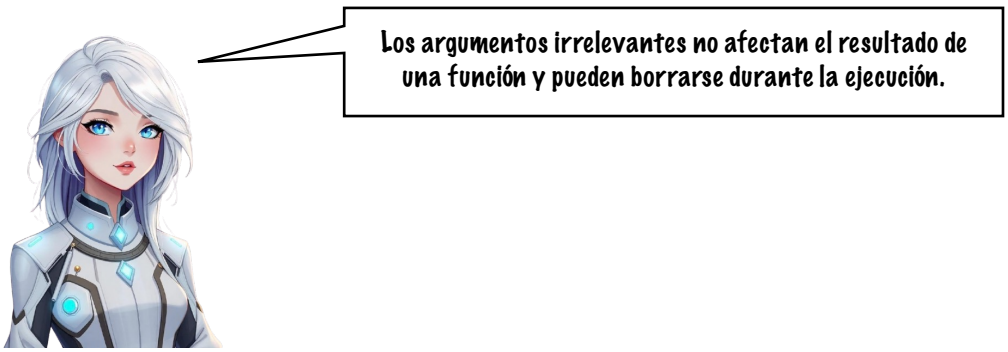
```
module UnorderedListEq (A : Set) ⦃ _ : Eq A ⦄ where
  open Eq (List A) public
    renaming (_==_ to eqUnorderedList)
    where
      xs == ys = toSet xs == toSet ys
  infix 4 _≋_
  _≋_ : List A → List A → Bool
  xs ≋ ys = xs `eqUnorderedList` ys
```

Esto nos da una especie de polimorfismo abierto y extensible, donde podemos agregar nuevos requerimientos a tipos existentes simplemente definiendo un nuevo módulo e implementando los campos necesarios.

El patrón de codificación de type classes en Agda es muy similar a las type classes en lenguajes como Haskell o Scala. Algunas diferencias clave:

- En Agda, no existe una palabra clave `class` o `instance` especial - solo usamos el módulo y el mecanismo de registro estándar.
- Las instancias no se resuelven globalmente, sino que deben importarse explícitamente al scope local. Esto evita problemas de ambigüedad o incoherencia.
- Los programas Agda no generan diccionarios de métodos en tiempo de ejecución - la resolución de método se realiza en su totalidad durante la compilación.
- Las type classes de Agda son totalmente verificadas durante la compilación y no tienen ningún costo en tiempo de ejecución. Son puramente un dispositivo de estructuración estática.

4.6. Relevancia de argumentos



Un aspecto singular del sistema de tipos de Agda es que nos permite indicar la relevancia computacional de los argumentos de una función. Algunos argumentos juegan un papel en el comportamiento en tiempo de ejecución, mientras que otros son meramente "indicaciones de tipo" que pueden borrarse de forma segura durante la compilación.

4.6.1. Argumentos irrelevantes

En Agda, declaramos un argumento como irrelevante prefijando su nombre con un punto (`.`). Por ejemplo, consideremos la función `replicate`:

```
1 replicate : (n : Nat) → (x : A) → Vec A n
2 replicate zero    x = []
3 replicate (suc n) x = x :: replicate n x
```

El argumento `n` aparece en el tipo para indicar el largo del vector resultante, pero no se usa de ninguna manera computacionalmente interesante en el cuerpo de la función. Podemos marcar `n` como irrelevante:

```
1 replicate : .(n : Nat) → (x : A) → Vec A n
```

La definición queda igual, pero ahora el compilador sabe que `n` puede borrarse durante la ejecución, ya que no se usa para tomar ninguna decisión computacional real.

Los argumentos irrelevantes son similares a los implícitos en que a menudo se infieren y pueden omitirse. Pero van un paso más allá, garantizando que realmente no se usan de ninguna manera significativa. De hecho, Agda nos impide usar un argumento irrelevante de forma computacionalmente relevante:

```
1 bad : .(x : Nat) → Nat
2 bad x = x + 1
```

Esto falla en el chequeo de tipos, ya que `x` se declaró como irrelevante.

¿Por qué entonces molestarse en tomar un argumento que nunca se va a usar? Resulta que los argumentos irrelevantes a menudo juegan un papel crucial en el tipado y especificación de una función, incluso si son computacionalmente prescindibles.

4.6.2. Aplicaciones en estructuras de datos

Muchas estructuras de datos tienen componentes o índices que son necesarios para expresar invariantes, pero que pueden borrarse con seguridad en tiempo de ejecución. Algunos ejemplos incluyen:

Longitud de vectores: Frecuentemente indexamos los vectores con su longitud, la cual resulta irrelevante computacionalmente:

```
1 data Vec (A : Set) : .Nat → Set where
```

```
2   []  : Vec A 0
3   _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

La longitud no se necesita durante la ejecución para construir o deconstruir vectores, sólo para asegurar estáticamente que tengan el tamaño correcto.

Índices acotados: Otro ejemplo común son los índices "seguros" en alguna estructura de datos, como `Fin n` para números entre `0` y `n-1`:

```
1   data Fin : .Nat → Set where
2     zero : {n : Nat} → Fin (suc n)
3     suc : {n : Nat} → Fin n → Fin (suc n)
```

Los constructores `zero` y `suc` no utilizan realmente el parámetro `Nat` en tiempo de ejecución; sólo está ahí para garantizar que siempre tengamos un índice válido para el tamaño dado.

Pruebas: Muchos tipos de datos llevan pruebas o evidencia de ciertas propiedades como parámetros. Por lo general, estos son irrelevantes computacionalmente. Por ejemplo:

```
1   data SortedList (A : Set) ⦃ _ : Ord A ⦄ : Set where
2     []  : SortedList A
3     _::_ : (x : A) → .(xs : SortedList A) → .⦃ _ : (y : A) → y ∈ xs → y ≥ x ⦄ → SortedList A
4   data SortedList (A : Set) ⦃ _ : Ord A ⦄ : Set where
```

El tipo `SortedList A` representa listas ordenadas de elementos de tipo `A`, donde el orden viene dado por una restricción `Ord A` implícita. El constructor `_::_` toma un elemento `x`, una sublista ordenada `xs` y una prueba implícita de que todos los elementos `y` en `xs` son mayores o iguales a `x`. Esta prueba se necesita para garantizar el invariante de orden, pero puede borrarse con seguridad en tiempo de ejecución.

4.6.3. Relevancia y eficiencia

Además de aliviar la carga en la especificación de tipos, marcar argumentos como irrelevantes también puede tener un impacto significativo en la eficiencia. Como el compilador sabe que estos argumentos nunca se inspeccionan en tiempo de ejecución, puede eliminarlos por completo durante la compilación, evitando el costo de pasarlos y almacenarlos.

Esto puede conducir a mejoras de rendimiento masivas, especialmente para tipos de datos complejos construidos principalmente con fines de verificación. Por ejemplo, consideremos un tipo dependiente que representa matrices junto con sus dimensiones:

```
1   record Matrix (A : Set) (m n : Nat) : Set where
2     field
3       elements : Vec (Vec A n) m
```

Si marcamos `m` y `n` como irrelevantes, el compilador puede representar `Matrix A m n` simplemente como un puntero plano a un bloque de `m * n` elementos de tipo `A`, en lugar de tener que llevar las dimensiones como valores separados y verificar los límites en cada acceso.

De manera más general, los argumentos irrelevantes nos permiten aprovechar la expresividad de los tipos dependientes para verificación y especificación, sin pagar el costo de arrastrarlos en tiempo de ejecución. Esto es crucial para hacer que los lenguajes con tipos dependientes sean prácticos para la programación en el mundo real.

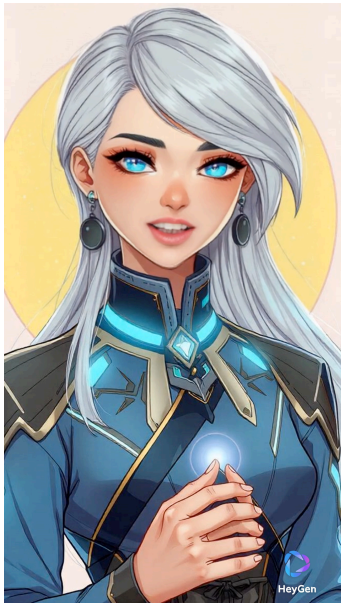
Tipo de argumento	Sintaxis	Inferencia	Relevancia en tiempo de ejecución
Explícito	(x : A)	Nunca inferido - debe proporcionarse en el sitio de llamada	Relevante - se pasa y utiliza durante la ejecución
Implícito	{x : A}	Inferido a partir del contexto de tipo - puede omitirse	Relevante - se pasa y utiliza durante la ejecución
Nivel de instancia	⦃ x : A ⦄	Inferido a partir de instancias en scope - usualmente omitido	Relevante - se pasa y utiliza durante la ejecución (como diccionario de métodos)
Irrelevante	.(x : A)	Inferido a partir del contexto de tipo - puede omitirse	Irrelevante - se borra durante la compilación, nunca se usa en tiempo de ejecución

Conclusiones del Capítulo
En este capítulo, exploramos las características avanzadas del sistema de tipos de Agda que lo distinguen por su expresividad y potencia. Aprendimos cómo los tipos dependientes nos permiten capturar propiedades complejas directamente en los tipos, fusionando especificación e implementación. También vimos el poder de la coinducción para modelar estructuras infinitas y el rol de la inferencia para reducir el overhead en la anotación de tipos. Cubrimos varias formas de polimorfismo - paramétrico, ad-hoc y with-abstraction - para promover la reutilización de código y la modularidad. Finalmente, discutimos la noción de relevancia de los argumentos, la cual nos permite aprovechar los tipos dependientes para verificación y especificación sin pagar un costo en tiempo de ejecución.

Ejercicios Propuestos	
1	Define un tipo `MatrizBien` que represente matrices rectangulares con dimensiones conocidas en tiempo de compilación. Implementa una función `multiplicar` que solo permita multiplicar matrices con dimensiones compatibles.
2	Define un tipo `ListaOrdenada` que represente listas ordenadas de manera ascendente. Implementa funciones `insertar` y `fusionar` que mantengan la propiedad de ordenación.
3	Define un tipo coinductivo `Stream` que represente secuencias potencialmente infinitas. Implementa funciones para generar la secuencia de Fibonacci y los números primos como streams.
4	Implementa una función `zip` que sea lo más general posible, trabajando con cualquier tipo de contenedor que tenga una noción de "longitud" y acceso a elementos.
5	Define una clase de tipo `Ordenable` con una función `comparar`. Proporciona instancias para tipos básicos y demuestra cómo se puede usar para ordenar listas de diferentes tipos.
6	Define un tipo `Vector` con longitud indexada y una función `concatenar`. Usa argumentos irrelevantes para la prueba de que las longitudes se suman correctamente en la concatenación.

7	Define un pequeño universo de tipos que incluya naturales, booleanos y pares. Implementa funciones para codificar y decodificar valores de este universo.

Capítulo 5: Efectos y concurrencia en Agda



Objetivos del capítulo

En este capítulo, nos adentraremos en el fascinante mundo de los efectos y la concurrencia en Agda. Aprenderemos a manejar operaciones con efectos secundarios de manera segura y composicional utilizando tipos como `IO`, `ExceptT` para errores y excepciones, y `StateT` para programación con estado. Al finalizar, estarás listo para crear programas más complejos e interactivos en Agda, llevando tus habilidades al siguiente nivel. ¡Prepárate para una emocionante aventura!

5.1. Introducción

Hasta ahora, hemos explorado los conceptos fundamentales de la programación funcional en Agda, sumergiéndonos en el mundo de las funciones puras y el código modular y correcto. Sin embargo, para crear programas verdaderamente útiles y prácticos, necesitamos extender nuestras habilidades más allá de la programación puramente funcional.

Imagina poder interactuar con el entorno, comunicarte con los usuarios, acceder a recursos externos y realizar tareas de manera concurrente. En este capítulo, descubrirás cómo Agda te brinda las herramientas necesarias para trabajar con efectos y concurrencia de una manera segura, expresiva y composicional.

Exploraremos el tipo `IO`, que te permitirá realizar operaciones impuras manteniendo la integridad de tu código. Aprenderás a interactuar con el mundo exterior, como leer la entrada del usuario, escribir en la consola y acceder a archivos y recursos de red.

Además, dominarás técnicas avanzadas para el manejo de errores y excepciones utilizando el tipo `ExceptT`. Podrás propagar y manejar errores de manera elegante y composicional, evitando la necesidad de verificaciones de errores anidadas y engorrosas.

También te sumergirás en el mundo de la programación con estado utilizando el tipo `StateT`. Descubrirás cómo encapsular y manejar el estado mutable de manera segura y composicional, sin comprometer la pureza y la corrección de tu código.

A lo largo de este capítulo, presentaremos ejemplos prácticos y casos de uso realistas que te mostrarán cómo aplicar estos conceptos en situaciones concretas. Construirás aplicaciones interactivas, te comunicarás con servicios externos y manejarás la concurrencia de manera eficiente y segura.

Al final de este capítulo, tendrás una comprensión sólida de cómo trabajar con efectos y concurrencia en Agda. Estarás equipado con las herramientas y técnicas necesarias para crear programas más complejos y realistas, manteniendo la elegancia y la seguridad que caracterizan a la programación funcional.

Así que prepárate para embarcarte en un emocionante viaje donde los efectos y la concurrencia se encuentran con la pureza y la expresividad de Agda. ¡Comencemos esta increíble aventura!

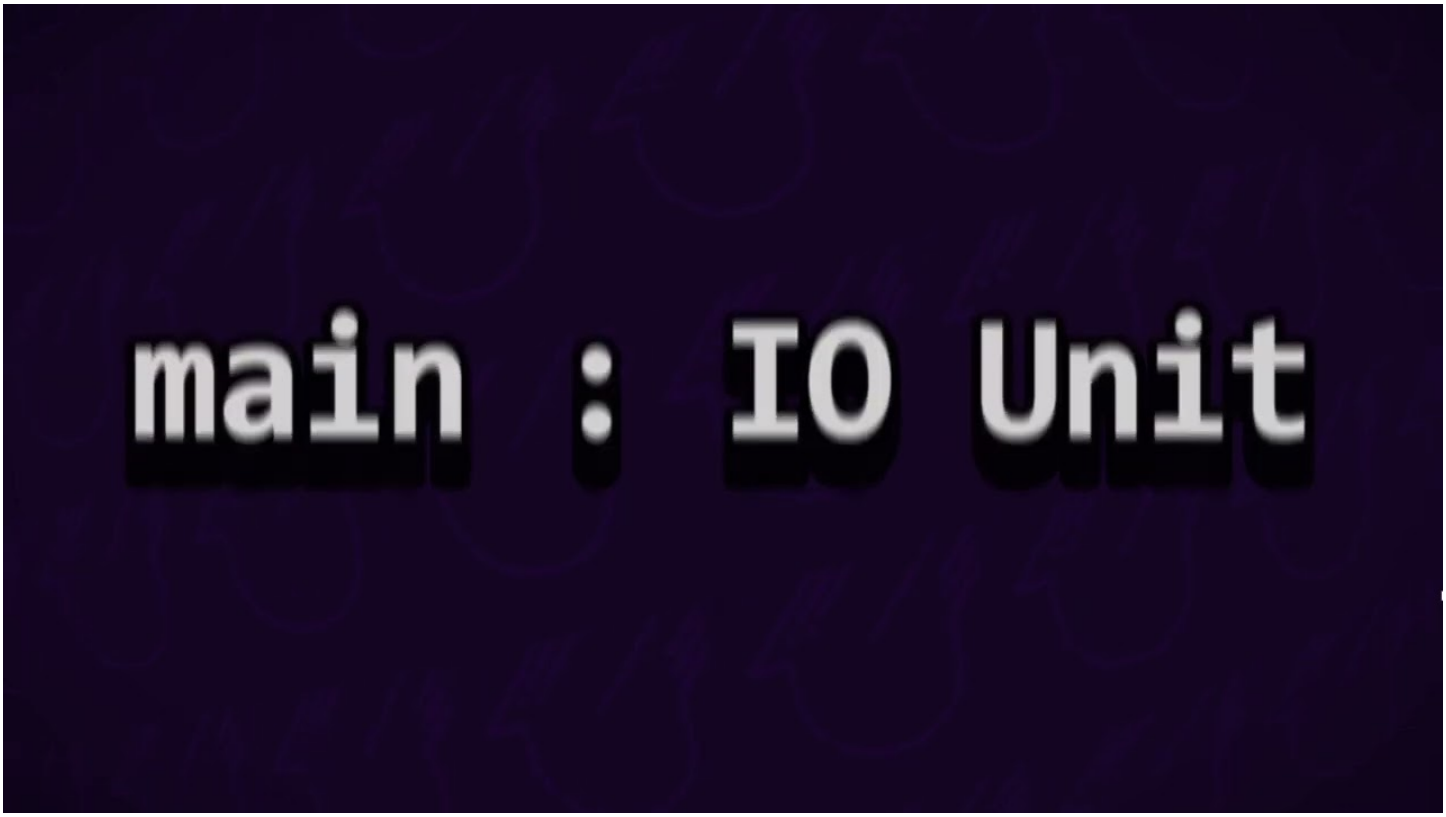
5.2. Interacción con el exterior

Cuando hablamos de interactuar con el mundo exterior en un programa, nos referimos a todas aquellas operaciones que involucran entrada y salida (E/S) de datos. Estas operaciones son fundamentales para crear programas útiles y prácticos que puedan interactuar con el entorno y realizar tareas significativas.

Sin embargo, en el paradigma de la programación funcional pura, donde las funciones son deterministas y no tienen efectos secundarios, la interacción con el exterior puede ser un desafío. Afortunadamente, Agda nos proporciona herramientas poderosas para manejar los efectos de manera segura y composicional.

5.2.1. Manejo de efectos con IO

En Agda, el tipo `IO` es el mecanismo principal para representar y manejar los efectos de entrada y salida. Un valor de tipo `IO A` representa una acción o computación que puede tener efectos secundarios y, en última instancia, produce un resultado de tipo `A`. Veamos un ejemplo básico para entender cómo funciona:



Video 05: manejo de entrada/salida

```
1  module IOExample where
2    postulate
3      getLine : IO String
4      putStrLn : String → IO Unit
5    {-# FOREIGN GHC import qualified Data.Text as Text #-}
6    {-# COMPILE GHC getLine = fmap Text.unpack Text.getLine #-}
7    main : IO Unit
8    main = do
9      putStrLn "¿Cuál es tu nombre?"
10     name ← getLine
11     putStrLn ("Hola " ++ name ++ ", ¡bienvenido a Agda!")
```

En este ejemplo, comenzamos postulando dos funciones: `getLine` y `putStrLn`. La función `getLine` tiene tipo `IO String`, lo que significa que es una acción de E/S que, cuando se ejecuta, lee una línea de la entrada estándar y devuelve esa línea como una cadena. Por otro lado, la función `putStrLn` tiene tipo `String → IO`

`Unit`, lo que indica que toma una cadena como argumento y devuelve una acción de E/S que, cuando se ejecuta, imprime esa cadena en la consola.

A continuación, definimos la función `main`, que es el punto de entrada de nuestro programa. La función `main` tiene tipo `IO Unit`, lo que señala que es una acción de E/S que no produce un resultado significativo.

Dentro de la función `main`, utilizamos la notación `do` para secuenciar las acciones de E/S. Primero, utilizamos `putStrLn` para imprimir un mensaje de saludo en la consola. Luego, utilizamos el operador ← (se lee "bind") para ejecutar la acción `getLine` y asignar el resultado a la variable `name`. Finalmente, utilizamos `putStrLn` nuevamente para imprimir un saludo personalizado que incluye el nombre ingresado por el usuario.

Es importante destacar que las acciones de E/S no se ejecutan inmediatamente cuando se definen, sino que se construyen como valores que representan las operaciones deseadas. Solo cuando se ejecuta la acción de E/S principal, en este caso `main`, se llevan a cabo los efectos en el orden especificado.

El tipo `IO` nos brinda una forma segura y controlada de realizar operaciones con efectos secundarios en Agda. Nos permite encapsular y razonar sobre los efectos de manera explícita, sin comprometer la pureza del resto de nuestro código.

Recuerda que, aunque el tipo `IO` nos permite realizar operaciones con efectos secundarios, no debemos abusar de su uso. Siempre debemos esforzarnos por mantener la mayor parte de nuestro código puro y limitar el uso de `IO` a las partes del programa que realmente lo necesiten.

A medida que avancemos en este capítulo, exploraremos más ejemplos y casos de uso del tipo `IO` y veremos cómo se integra con otras técnicas y conceptos para construir programas más complejos y realistas.

5.2.2. Interacción con E/S

Ahora que hemos visto cómo el tipo `IO` nos permite manejar los efectos de entrada y salida de manera segura y composicional, veamos algunos ejemplos concretos de cómo podemos interactuar con diferentes fuentes de E/S en Agda.

Leyendo y escribiendo archivos

Una tarea común en la programación es leer y escribir archivos en el sistema de archivos. Agda nos proporciona funciones y bibliotecas para realizar estas operaciones de manera sencilla y segura.

Para leer el contenido de un archivo, podemos utilizar la función `readFile` de la biblioteca estándar de Agda. Veamos un ejemplo:

```
1  module ReadFileExample where
2    postulate
3      readFile : String → IO String
4    main : IO Unit
5    main = do
```

```
6     content ← readFile "example.txt"
7     putStrLn content
```

En este ejemplo, postulamos la función `readFile` con tipo `String → IO String`. Esta función toma la ruta del archivo como argumento y devuelve una acción de E/S que, cuando se ejecuta, lee el contenido del archivo y lo devuelve como una cadena.

En la función `main`, utilizamos `readFile` para leer el contenido del archivo `"example.txt"`. Utilizamos el operador `←` para asignar el resultado de la acción de lectura a la variable `content`. Luego, utilizamos `putStrLn` para imprimir el contenido del archivo en la consola.

De manera similar, para escribir en un archivo, podemos utilizar la función `writeFile` de la biblioteca estándar de Agda. Veamos un ejemplo:

```
1  module WriteFileExample where
2    postulate
3      writeFile : String → String → IO Unit
4    main : IO Unit
5    main = do
6      let message = "¡Hola, mundo!"
7      writeFile "output.txt" message
```

En este ejemplo, postulamos la función `writeFile` con tipo `String → String → IO Unit`. Esta función toma dos argumentos: la ruta del archivo donde se escribirá y el contenido que se escribirá en el archivo. La función devuelve una acción de E/S que, cuando se ejecuta, escribe el contenido en el archivo especificado.

En la función `main`, definimos una variable `message` con el contenido que queremos escribir en el archivo. Luego, utilizamos `writeFile` para escribir el contenido de `message` en el archivo `"output.txt"`.

Estos ejemplos demuestran cómo podemos interactuar con archivos en Agda de manera sencilla y segura utilizando las funciones `readFile` y `writeFile`. Estas funciones nos permiten leer y escribir archivos como acciones de E/S, manteniendo la separación entre los efectos y el resto del código.

Función	Tipo	Descripción
readFile	String → IO String	Lee el contenido de un archivo y devuelve una acción IO con el contenido
writeFile	String → String → IO Unit	Escribe un contenido en un archivo y devuelve una acción IO
appendFile	String → String → IO Unit	Añade un contenido al final de un archivo y devuelve una acción IO
openFile	String → Mode → IO Handle	Abre un archivo en un modo específico y devuelve una acción IO con un manejador de archivo
hClose	Handle → IO Unit	Cierra un manejador de archivo y devuelve una acción IO

hIsEOF	Handle → IO Bool	Comprueba si se ha alcanzado el final de un archivo y devuelve una acción IO con un booleano
hGetLine	Handle → IO String	Lee una línea de un manejador de archivo y devuelve una acción IO con la línea
hPutStrLn	Handle → String → IO Unit	Escribe una cadena en un manejador de archivo, seguida de una nueva línea, y devuelve una acción IO

Comunicación en red

En el mundo interconectado de hoy, la comunicación en red es fundamental para muchas aplicaciones. Agda nos brinda bibliotecas y herramientas para interactuar con servicios web, APIs y otros recursos en línea.

Veamos un ejemplo de cómo podemos realizar una solicitud HTTP utilizando la biblioteca `agdax-http` en Agda:

```
1  module HttpExample where
2    import Agdax.Http as Http
3    import Agdax.Json as Json
4    postulate
5      processData : Json.Value → IO Unit
6    main : IO Unit
7    main = do
8      response ← Http.get "https://api.example.com/data"
9      case Json.decodeString response of
10       Left err   → putStrLn ("Error: " ++ err)
11       Right data → processData data
```

En este ejemplo, importamos los módulos `Agdax.Http` y `Agdax.Json` para acceder a las funcionalidades de comunicación HTTP y manejo de JSON, respectivamente.

Postulamos una función `processData` con tipo `Json.Value → IO Unit`, que representa una acción de E/S que procesa los datos JSON obtenidos de la respuesta HTTP.

En la función `main`, utilizamos `Http.get` para realizar una solicitud GET a la URL `"https://api.example.com/data"`. Esta función devuelve una acción de E/S que, cuando se ejecuta, envía la solicitud HTTP y devuelve la respuesta como una cadena.

Luego, utilizamos `Json.decodeString` para decodificar la respuesta JSON. Esta función toma la cadena de respuesta y devuelve un valor de tipo `Either String Json.Value`, donde `Left err` representa un error de decodificación y `Right data` representa los datos JSON decodificados correctamente.

Utilizamos una expresión `case` para manejar los posibles resultados de la decodificación. Si se produce un error (`Left err`), imprimimos un mensaje de error utilizando `putStrLn`. Si la decodificación es exitosa (`Right data`), llamamos a la función `processData` para procesar los datos JSON obtenidos.

Este ejemplo ilustra cómo podemos realizar solicitudes HTTP y manejar respuestas JSON en Agda utilizando las bibliotecas `agdax-http` y `agdax-json`. Estas bibliotecas nos permiten interactuar con servicios web y APIs de manera sencilla y segura, aprovechando la potencia y expresividad de Agda.

La comunicación en red es solo un ejemplo de cómo podemos interactuar con el mundo exterior en Agda. Existen otras bibliotecas y herramientas que nos permiten interactuar con bases de datos, sistemas de mensajería, protocolos de comunicación y más. La clave es utilizar estas herramientas de manera segura y composicional, aprovechando el sistema de tipos de Agda para garantizar la corrección y la integridad de nuestro código.

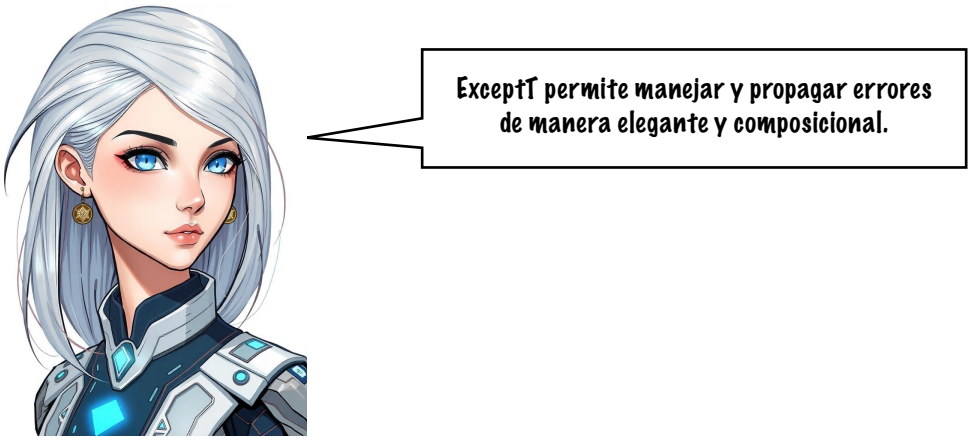
A medida que construyamos aplicaciones más complejas y realistas, la capacidad de interactuar con el mundo exterior se volverá cada vez más importante. Agda nos brinda las herramientas y abstracciones necesarias para manejar estos efectos de manera elegante y segura, permitiéndonos crear programas robustos y confiables.

5.3. Manejo de errores

Cuando interactuamos con el mundo exterior y realizamos operaciones con efectos secundarios, es importante tener en cuenta que las cosas pueden salir mal. Los archivos pueden no existir, las conexiones de red pueden fallar, los datos pueden ser inválidos, entre otros problemas. En esos casos, necesitamos una forma de manejar y propagar los errores adecuadamente para evitar que nuestro programa se bloquee o se comporte de manera inesperada.

En esta sección, exploraremos cómo Agda nos permite manejar los errores y las excepciones de manera segura y composicional, utilizando el tipo `ExceptT`. Veremos cómo podemos encapsular y propagar los errores de manera elegante, sin tener que recurrir a verificaciones de errores anidadas o engorrosas.

5.3.1. Manejo de excepciones



El tipo `ExceptT` en Agda brinda una forma poderosa y composicional de manejar excepciones y errores en el código. `ExceptT` permite definir una estructura de datos que representa una computación que puede fallar con un error de un tipo específico.

Veamos un ejemplo para entender su funcionamiento:



Video 06: Ejemplo de manejo de errores con EXCET

```
1  module ExceptExample where
2    open import Data.Either
3    open import Data.String
4    open import IO
5    data FileError : Set where
6      FileNotFound : String → FileError
7      PermissionDenied : String → FileError
8    readFile : String → ExceptT FileError IO String
9    readFile path = do
10     exists ← lift $ IO.doesFileExist path
11     if exists
12       then do
13         content ← lift $ IO.readFile path
14         pure content
15       else throwE (FileNotFound path)
```

```
16  main : IO Unit
17  main = do
18    result ← runExceptT (readFile "example.txt")
19    case result of
20      Left (FileNotFound path) →
21        putStrLn ("File " ++ path ++ " not found.")
22      Left (PermissionDenied path) →
23        putStrLn ("Permission denied for file " ++ path)
24      Right content →
25        putStrLn ("File content: " ++ content)
```

En este ejemplo, se define un tipo de datos personalizado `FileError` que representa los posibles errores al leer un archivo: `FileNotFound` para indicar que el archivo no existe y `PermissionDenied` para indicar falta de permisos de lectura.

La función `readFile` toma la ruta del archivo y devuelve una computación `ExceptT FileError IO String`. Utilizando `lift`, se elevan las acciones de E/S al contexto de `ExceptT`. Primero, se verifica la existencia del archivo con `IO.doesFileExist`. Si existe, se lee su contenido con `IO.readFile` y se devuelve con `pure`. Si no existe, se lanza una excepción con `throwE` y el constructor `FileNotFound`.

En `main`, se ejecuta la computación `readFile` utilizando `runExceptT`. Luego, se utiliza un `case` para manejar los posibles resultados: si se produce un error `FileNotFound`, se imprime un mensaje indicando que el archivo no se encontró; si se produce un error `PermissionDenied`, se imprime un mensaje de falta de permisos; si la lectura es exitosa, se imprime el contenido del archivo.

`ExceptT` permite manejar los errores de manera clara y composicional. Se pueden definir tipos de errores personalizados y utilizar `throwE` para lanzar excepciones en cualquier punto de la computación. Luego, con `runExceptT`, se ejecuta la computación y se manejan los posibles errores adecuadamente.

5.3.2. Propagación de errores

Uno de los beneficios de `ExceptT` es la propagación composicional de errores. Se pueden encadenar múltiples computaciones que pueden fallar y manejar los errores en un nivel superior, evitando verificaciones de errores anidadas o engorrosas.

Veamos un ejemplo más complejo que ilustra la propagación de errores:

```
1  module PropagateExample where
2    open import Data.Either
3    open import Data.String
4    open import IO
5    data ValidationError : Set where
```

```
    EmptyField : String → ValidationError
    InvalidEmail : String → ValidationError
  validateName : String → ExceptT ValidationError IO String
  validateName name =
    if name == ""
      then throwE (EmptyField "name")
      else pure name
  validateEmail : String → ExceptT ValidationError IO String
  validateEmail email =
    if "@" `isInfixOf` email
      then pure email
      else throwE (InvalidEmail email)
  data User : Set where
    User : String → String → User
  registerUser : String → String → ExceptT ValidationError IO User
  registerUser name email = do
    validName ← validateName name
    validEmail ← validateEmail email
    pure (User validName validEmail)
  main : IO Unit
  main = do
    result ← runExceptT (registerUser "Alice" "alice@example.com")
    case result of
      Left (EmptyField field) →
        putStrLn ("Error: Field " ++ field ++ " cannot be empty.")
      Left (InvalidEmail email) →
        putStrLn ("Error: Invalid email " ++ email)
      Right (User name email) →
        putStrLn ("User registered: " ++ name ++ " (" ++ email ++ ")")
```

En este ejemplo, se define un tipo de datos `ValidationError` que representa los posibles errores de validación: `EmptyField` para indicar que un campo está vacío y `InvalidEmail` para indicar un formato de correo electrónico inválido.

Las funciones `validateName` y `validateEmail` realizan las validaciones correspondientes. Si el nombre está vacío, se lanza una excepción con `EmptyField`; si el correo electrónico no contiene "@", se lanza una excepción con `InvalidEmail`.

La función `registerUser` toma el nombre y el correo electrónico, y devuelve una computación `ExceptT ValidationError IO User`. Utilizando `do`, se encadenan las validaciones: primero se valida el nombre con `validateName` y luego el correo electrónico con `validateEmail`. Si ambas validaciones son exitosas, se crea un valor de tipo `User` con los datos validados y se devuelve con `pure`.

En `main`, se ejecuta la computación `registerUser` con `runExceptT`. Se utiliza un `case` para manejar los posibles resultados: si se produce un error `EmptyField`, se imprime un mensaje indicando que el campo no puede estar vacío; si se produce un error `InvalidEmail`, se imprime un mensaje de correo electrónico inválido; si el registro es exitoso, se imprime un mensaje de confirmación con los datos del usuario registrado.

Este ejemplo muestra cómo encadenar múltiples computaciones que pueden fallar utilizando `ExceptT` y manejar los errores de manera composicional. Los errores se propagan hacia arriba y se manejan en un nivel superior, evitando verificaciones de errores en cada paso intermedio.

La propagación de errores permite escribir código más claro, conciso y fácil de mantener. En lugar de verificar los errores en cada paso, estos se propagan automáticamente y se manejan en el punto adecuado.

`ExceptT` permite combinar diferentes tipos de errores y manejarlos de manera uniforme. Se pueden definir tipos de errores personalizados y utilizar `throwE` para lanzar excepciones en cualquier punto de la computación. Luego, con `runExceptT`, se ejecuta la computación y se manejan los diferentes tipos de errores adecuadamente.

El manejo de excepciones y la propagación de errores son fundamentales en la programación con efectos en Agda. `ExceptT` brinda una forma elegante y composicional de manejar los errores, permitiendo escribir código más robusto y fácil de mantener. Con `ExceptT`, se encapsulan los errores, se propagan automáticamente y se manejan en un nivel superior, evitando verificaciones de errores anidadas o engorrosas.

5.4. Estado

En las secciones anteriores, exploramos cómo Agda nos permite trabajar con efectos y manejar errores de manera segura y composicional. Sin embargo, en muchos casos prácticos, también necesitamos la capacidad de trabajar con estado mutable en nuestros programas. El estado nos permite almacenar y modificar información durante la ejecución, lo cual es esencial para modelar sistemas dinámicos y reactivos.

En esta sección, nos sumergiremos en las herramientas que Agda nos brinda para trabajar con estado de manera segura y composicional, utilizando el tipo `StateT`. Aprenderemos a encapsular y manejar el estado mutable de forma elegante, sin comprometer la pureza y la corrección de nuestro código. ¿Estás listo para dar el siguiente paso en tu viaje por la programación funcional? ¡Vamos a ello!

5.4.1. Estado y mutabilidad

Imagina que estás desarrollando un videojuego y necesitas llevar un registro del puntaje del jugador a medida que avanza en el juego. Aquí es donde entra en juego el estado mutable. El tipo `StateT` en Agda es una herramienta poderosa que nos permite representar computaciones que pueden leer y modificar un estado mutable de manera segura y composicional. `StateT` encapsula el estado dentro de una estructura de datos y nos proporciona operaciones para acceder y actualizar ese estado de manera controlada.

Veamos un ejemplo concreto para entender mejor su funcionamiento:

```
module StateExample where
  open import Control.Monad.State
  open import Data.Nat
  open import IO
  data Event : Set where
    Increment : Event
    Decrement : Event
  handleEvent : Event → StateT Nat IO Unit
  handleEvent Increment = do
    count ← get
    put (count + 1)
    lift $ putStrLn ("Count incremented: " ++ show (count + 1))
  handleEvent Decrement = do
    count ← get
    if count > 0 then do
      put (count - 1)
      lift $ putStrLn ("Count decremented: " ++ show (count - 1))
    else
      lift $ putStrLn "Count is already zero"
  main : IO Unit
  main = do
    let initialState = 0
    _ ← runStateT initialState $ do
      handleEvent Increment
      handleEvent Increment
      handleEvent Decrement
      handleEvent Decrement
      handleEvent Decrement
    pure unit
```

En este ejemplo, definimos un tipo de datos `Event` que representa los eventos que pueden ocurrir en nuestro programa: `Increment` para incrementar un contador y `Decrement` para decrementarlo.

La función `handleEvent` toma un evento y devuelve una computación `StateT Nat IO Unit`. Dentro de esta función, utilizamos las operaciones `get` y `put` proporcionadas por `StateT` para acceder y modificar el estado mutable de manera segura. En el caso de `Increment`, obtenemos el valor actual del contador con `get`, lo incrementamos en 1 y actualizamos el estado con `put`. Luego, utilizamos `lift` para imprimir un mensaje con el

nuevo valor del contador. En el caso de `Decrement`, verificamos si el contador es mayor que cero antes de decrementarlo y actualizar el estado. Si el contador ya es cero, simplemente imprimimos un mensaje indicando que no se puede decrementar más.

En la función `main`, definimos un estado inicial para el contador (en este caso, 0) y utilizamos `runStateT` para ejecutar una secuencia de eventos. Dentro de la computación `StateT`, utilizamos `handleEvent` para manejar los eventos `Increment` y `Decrement` en un orden específico. Finalmente, utilizamos `pure unit` para indicar que la computación no devuelve ningún resultado significativo.

`StateT` nos brinda una forma elegante y composicional de manejar el estado mutable. Podemos encapsular el estado dentro de una estructura de datos y utilizar las operaciones `get` y `put` para acceder y modificar ese estado de manera controlada. Además, `StateT` se integra perfectamente con otras estructuras de efectos, como `IO`, lo que nos permite combinar el manejo de estado con operaciones de entrada/salida u otros efectos.

Es importante destacar que, aunque `StateT` nos permite trabajar con estado mutable, debemos utilizarlo de manera cuidadosa y limitada. En general, es preferible mantener la mayor parte de nuestro código puro y funcional, y recurrir al estado solo cuando sea estrictamente necesario. El uso excesivo o inapropiado del estado puede llevar a un código más complejo, propenso a errores y difícil de razonar. Como programadores funcionales, nuestro objetivo es aprovechar la pureza y la composicionalidad siempre que sea posible.

Concepto	Explicación	Ejemplo
Efecto	Cambio observable fuera del ámbito local de una función, como I/O, estado mutable, excepciones, etc.	readFile : String → IO String
Acción	Un valor que representa una computación con efectos, pero no los ejecuta inmediatamente	getLine : IO String
Ejecución	Realizar los efectos de una acción, obteniendo su resultado	main : IO ()main = do name ← getLine putStrLn ("Hello, " ++ name ++ "!")
Composición secuencial	Combinar acciones de manera que los efectos de una ocurran antes que los de la siguiente	do a ← actionA b ← actionB return (a, b)
Composición paralela	Combinar acciones de manera que sus efectos ocurran en paralelo o entrelazados	concurrently : IO a → IO b → IO (a, b)
Manejo de errores	Detectar y recuperarse de condiciones anormales o excepcionales	try : IO a → IO (Either e a)
Interacción con el exterior	Comunicación con sistemas externos, como archivos, redes, interfaces de usuario, etc.	httpGet : String → IO String

5.4.2. Argumentos irrelevantes

Ahora, adentrémonos en un concepto interesante en Agda: los argumentos irrelevantes. En Agda, tenemos la posibilidad de marcar ciertos argumentos de funciones como "irrelevantes" utilizando el operador `@0`. Pero, ¿qué significa esto exactamente? Los argumentos irrelevantes son aquellos que no afectan el comportamiento computacional de la función, pero pueden proporcionar información adicional a nivel de tipos.

Veamos un ejemplo para entender mejor este concepto:

```
module IrrelevantExample where
  open import Data.Nat
  open import Relation.Binary.PropositionalEquality
  replicate : ∀ {A : Set} → (n : Nat) → @0 A → Vec A n
  replicate zero    x = []
  replicate (suc n) x = x :: replicate n x
  example : replicate 3 "hello" ≡ "hello" :: "hello" :: "hello" :: []
  example = refl
```

En este ejemplo, definimos una función `replicate` que toma un número natural `n`, un elemento `x` de tipo `A` y devuelve un vector de longitud `n` que contiene `x` replicado `n` veces. El argumento `x` está marcado como irrelevante utilizando `@0`, lo cual indica que su valor no se utiliza en la definición de la función.

La función `replicate` se define por casos:

- En el caso base, cuando `n` es cero, se devuelve un vector vacío `[]`.
- En el caso inductivo, cuando `n` es `suc n`, se construye un vector que consiste en `x` seguido de `replicate n x`, que es el resultado de replicar `x` `n` veces.

En el ejemplo `example`, afirmamos que `replicate 3 "hello"` es igual a `"hello" :: "hello" :: "hello" :: []`. Utilizamos la función `refl` para demostrar que esta igualdad se cumple.

El uso de argumentos irrelevantes tiene varios beneficios. En primer lugar, nos permite marcar explícitamente qué argumentos no afectan el comportamiento computacional de la función, haciendo el código más claro y autodocumentado. Además, permite al compilador de Agda realizar optimizaciones adicionales, ya que sabe que el valor del argumento irrelevante no influye en el resultado de la función. Por último, ayuda a evitar dependencias innecesarias a nivel de tipos. Al marcar un argumento como irrelevante, indicamos que no es necesario para el sistema de tipos y puede ser ignorado en ciertas situaciones.

Sin embargo, es importante utilizar los argumentos irrelevantes con precaución. Solo debemos marcar un argumento como irrelevante si tenemos la certeza de que su valor no afecta el comportamiento de la función. Marcar un argumento como irrelevante cuando en realidad sí afecta el resultado puede llevar a errores sutiles y difíciles de detectar. Como programadores responsables, debemos ser cuidadosos al emplear esta característica y asegurarnos de usarla correctamente.

En resumen, los argumentos irrelevantes en Agda nos permiten marcar explícitamente qué argumentos no afectan el comportamiento computacional de una función. Esto mejora la claridad del código, permite optimizaciones adicionales y ayuda a evitar dependencias innecesarias a nivel de tipos. Sin embargo, debemos usarlos con precaución y asegurarnos de que realmente sean irrelevantes para el comportamiento de la función. ¡Aprovecha esta característica sabiamente en tus programas en Agda!

Tipo de Argumento	Sintaxis	Inferencia	Relevancia en Tiempo de Ejecución
Explícito	(x : A)	Nunca inferido - debe proporcionarse en el sitio de llamada	Relevante - se pasa y utiliza durante la ejecución
Implícito	{x : A}	Inferido a partir del contexto de tipo - puede omitirse	Relevante - se pasa y utiliza durante la ejecución
Nivel de Instancia	⦃ x : A ⦄	Inferido a partir de instancias en scope - usualmente omitido	Relevante - se pasa y utiliza durante la ejecución (como diccionario de métodos)
Irrelevante	.(x : A)	Inferido a partir del contexto de tipo - puede omitirse	Irrelevante - se borra durante la compilación, nunca se usa en tiempo de ejecución

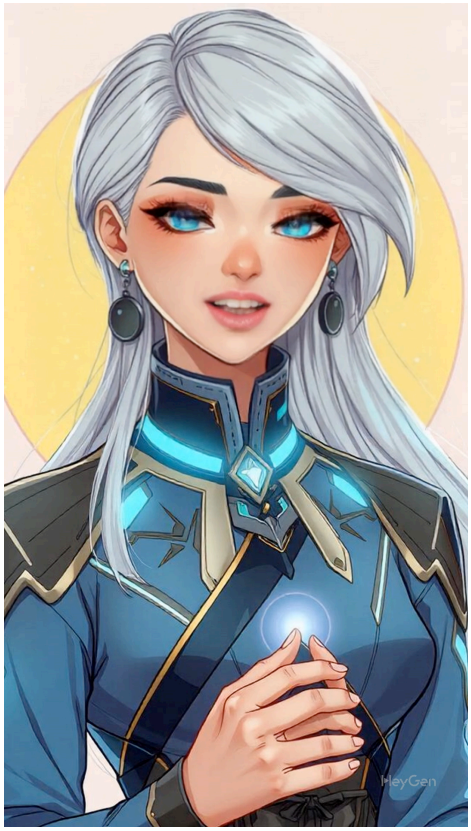
Conclusiones del Capítulo

En este capítulo, avanzamos en nuestro viaje por la programación funcional en Agda, explorando efectos y concurrencia. Descubrimos cómo Agda nos proporciona herramientas poderosas para manejar operaciones con efectos secundarios de manera segura y composicional, a través del tipo `IO`. Dominamos técnicas avanzadas para el manejo de errores utilizando `ExceptT`, lo que nos permite escribir código más robusto y resiliente. También nos sumergimos en la programación con estado usando `StateT`, aprendiendo a manejar el estado mutable de forma segura y realizar operaciones de lectura y escritura controladas. Además, exploramos el concepto de argumentos irrelevantes en Agda y cómo pueden mejorar la claridad y el rendimiento de nuestro código.

Ejercicios Propuestos

1	Implementa un programa simple que lea una línea de entrada del usuario, la procese (por ejemplo, convirtiéndola a mayúsculas), y luego imprima el resultado.
2	Define un tipo `DivisionSegura` que represente el resultado de una división que puede fallar (división por cero). Implementa la función de división y una serie de operaciones que usen este tipo, manejando adecuadamente los posibles errores.
3	Implementa un pequeño intérprete para un lenguaje de programación simple que tenga variables y asignaciones. Usa la mónada `State` para manejar el estado de las variables.
4	Crea una mónada personalizada que combine los efectos de `State` y `Maybe`. Usa esta mónada para implementar un juego simple donde el jugador tiene un inventario (manejado por `State`) y puede realizar acciones que pueden fallar (manejado por `Maybe`).
5	Implementa una versión simplificada de "futures" o "promesas" en Agda. Demuestra cómo se pueden usar para realizar operaciones concurrentes simples.
6	Define un sistema simple de efectos algebraicos que incluya operaciones para manejar estado y errores. Implementa un intérprete para este sistema.
7	Escribe pruebas formales para demostrar propiedades de programas que usan IO, como que ciertas operaciones no modifican ciertos aspectos del estado del programa.

Capítulo 6: Mónadas en Agda



Objetivos del capítulo

En este capítulo, daremos un gran salto en nuestro viaje por la programación funcional en Agda, sumergiéndonos en el fascinante mundo de los efectos y la concurrencia. Descubriremos cómo Agda nos brinda herramientas poderosas para manejar operaciones con efectos secundarios de forma segura y composicional, mediante el tipo `IO`. Aprenderemos técnicas avanzadas para el manejo de errores usando `ExceptT`, lo que nos permitirá escribir código más robusto y resistente a fallas. Además, exploraremos la programación con estado utilizando `StateT`, aprendiendo a manejar el estado mutable de manera segura y a realizar operaciones controladas de lectura y escritura.

6.1. Introducción

En los capítulos previos, hemos explorado varias técnicas para trabajar con efectos en Agda, como `IO`, `ExceptT` y `StateT`. Si bien estas abstracciones son poderosas, pueden parecer un tanto ad-hoc y desconectadas a primera vista. En este capítulo, revelaremos que todas estas construcciones son en realidad instancias de un concepto más general y unificador: las mónadas.

Las mónadas son una de las ideas más influyentes y prevalentes en la programación funcional. Nos permiten estructurar y componer cómputos con efectos de una manera elegante y expresiva. A lo largo de este capítulo, cubriremos los fundamentos matemáticos de las mónadas, su implementación en la librería estándar de Agda, cómo definir nuestras propias instancias monádicas, técnicas para componerlas y transformarlas, así como patrones comunes para trabajar con cómputos monádicos de forma efectiva.

Dominar las mónadas te dará una herramienta increíblemente versátil para tu arsenal de programación funcional. Las mónadas tienen profundas conexiones con diversas áreas de las matemáticas, como la teoría de

categorías, la lógica y el álgebra. Además, han sido aplicadas exitosamente en una amplia gama de dominios, desde el parseo y la interfaz de usuario hasta la concurrencia y la verificación formal.

En cierto sentido, las mónadas son como la "salsa secreta" de la programación funcional. Una vez que las comprendes, muchos patrones y bibliotecas aparentemente dispares comienzan a tener sentido. Te permiten elevar tu pensamiento a un nuevo nivel de abstracción y ver estructuras comunes en medio del aparente caos.

Sin embargo, las mónadas también son notorias por ser un concepto difícil de explicar. Hay una famosa cita en la comunidad de Haskell que dice: "una mónada es un monoide en la categoría de endofuntores, ¿cuál es el problema?" Aunque es una broma que pretende ejemplificar cómo no explicar las mónadas a un principiante, ilustra la complejidad del tema. En este capítulo, intentaremos un enfoque diferente. Construiremos una comprensión intuitiva capa por capa, comenzando por motivaciones concretas y gradualmente trabajando hacia las abstracciones más elevadas.

Vale la pena reflexionar sobre por qué las mónadas encajan de manera tan natural en un lenguaje como Agda. En Agda, que tiene efectos nativos como `IO`, la razón clave por la que las mónadas brillan es su sistema de tipos dependientes. Con tipos ordinarios, las mónadas son útiles para el razonamiento y la estructuración del código. Pero con tipos dependientes, se vuelven esenciales para la modularidad y la abstracción.

Las mónadas nos permiten ocultar información, agrupar invariantes y crear interfaces compatibles con diferentes implementaciones subyacentes. Por ejemplo, podríamos usar una mónada para representar cómputos que podrían fallar o no terminar, ocultando los detalles de manejo de errores o terminación al código cliente. O podríamos definir una interfaz monádica para el acceso a bases de datos que abstrae las diferencias entre implementaciones SQL, NoSQL y en memoria.

Aunque las mónadas son enormemente útiles en lenguajes funcionales en general, brillan especialmente en el contexto de un asistente de pruebas como Agda. Nos ayudan a domesticar la complejidad inherente en la codificación de matemáticas y lógica en un sistema de tipos, y a extraer interfaces computacionales claras a partir de las entrañas teóricas de las categorías.

6.2. Fundamentos

6.2.1. Definición Formal de Mónadas



Desde una perspectiva matemática, una mónada es un concepto de la teoría de categorías. Sin embargo, en el contexto de la programación, podemos pensarla como una interfaz genérica con las siguientes operaciones:

```
record Monad (M : Set → Set) : Set₁ where
  field
    return : {A : Set} → A → M A
    _>>=_  : {A B : Set} → M A → (A → M B) → M B
```

Aquí, `return` es una función que toma un valor y lo envuelve en un contexto monádico. Por otro lado, `_>>=_` (pronunciado "bind") es una operación que toma un cómputo monádico y una función que puede generar más cómputos a partir de su resultado, y los compone en un solo cómputo.

Para que `M` sea una mónada legal, `return` y `_>>=_` deben satisfacer las siguientes leyes:

```
return x >>= f ≡ f x                -- left identity
m >>= return   ≡ m                  -- right identity
(m >>= f) >>= g ≡ m >>= (λ x → f x >>= g) -- associativity
```

Estas leyes establecen que `return` debe ser un elemento neutro para `_>>=_`, y que `_>>=_` debe ser asociativo. Esto nos permite razonar ecuacionalmente sobre cómputos monádicos de una manera análoga a propiedades familiares como la multiplicación de números.

En Agda, podemos codificar estas leyes como tipos y proporcionar pruebas para instancias específicas de mónadas. Por ejemplo, aquí están las leyes para la mónada `Maybe`:

```
maybe-left-id : {A B : Set} (x : A) (f : A → Maybe B)
              → (just x >>= f) ≡ f x
maybe-left-id x f = refl
maybe-right-id : {A : Set} (m : Maybe A) → (m >>= just) ≡ m
maybe-right-id (just x) = refl
maybe-right-id nothing  = refl
maybe-assoc : {A B C : Set} (m : Maybe A)
              (f : A → Maybe B) (g : B → Maybe C)
            → ((m >>= f) >>= g) ≡ (m >>= (λ x → f x >>= g))
maybe-assoc (just x) f g = refl
maybe-assoc nothing  f g = refl
```

Probar las leyes de mónada para cada instancia puede ser bastante tedioso, pero vale la pena para ganar confianza en la corrección de la implementación y habilitar el razonamiento ecuacional. Herramientas como el resolutor de Agda pueden automatizar muchas de las pruebas.

Desde una perspectiva de teoría de categorías, una mónada es un endofuntor (un funtor de una categoría a sí misma) equipado con dos transformaciones naturales: `return` (también conocida como `unit`) y `join` (relacionada con `_>>=_`). Estas transformaciones deben satisfacer leyes de coherencia análogas a las anteriores. Esta definición categórica captura elegantemente la esencia de las mónadas y guía el diseño de librerías monádicas.

Sin embargo, para la mayor parte de la programación del día a día, pensar en las mónadas como "cómputos con superpoderes" que obedecen algunas leyes razonables es totalmente adecuado. Lo importante es que las mónadas proporcionan una interfaz genérica para secuenciar y componer cálculos de una manera que abstrae los detalles del efecto que se está realizando. Son una poderosa herramienta tanto para estructurar código como para razonar sobre él.

6.2.2. Explicación Intuitiva de Mónadas

Pero, ¿qué significan las mónadas intuitivamente? Una manera de pensarlas es como computaciones con un "superpoder". El tipo `M A` representa un cómputo que eventualmente producirá un valor `A`, pero también puede hacer otras cosas interesantes en el camino, dependiendo de cómo se defina la mónada `M`.

Por ejemplo:

- Para `IO`, el superpoder es realizar efectos secundarios.
- Para `Maybe`, es fallar tempranamente.
- Para listas, es ramificarse en múltiples valores.
- Para `State`, es llevar un estado mutable.
- Para `Reader`, es depender de un entorno.
- Para `Writer`, es generar valores adjuntos.
- Para `Cont`, es suspender y resumir cómputos.
- Para `Parser`, es consumir entrada y lidiar con el fracaso.

La magia está en que la interfaz monádica (`return`/`_>>=_`) abstrae el superpoder específico, permitiéndonos escribir código genérico que funciona con cualquier mónada. Luego podemos usar las mónadas como bloques de construcción, combinándolas para crear cómputos con superpoderes personalizados.

Otra forma de pensar en las mónadas es como "programación orientada a flujo". En lugar de pasar explícitamente valores intermedios de una función a otra, usamos `_>>=_` para encadenar cómputos en un flujo continuo. Entonces las funciones se convierten en "tuberías" que transforman estos flujos.

También podemos ver las mónadas como una forma de "programación implícita". En lugar de mencionar explícitamente detalles como manejo de errores, estado o registros en cada función, los empujamos a la capa monádica. Entonces las funciones pueden enfocarse en la lógica del dominio, mientras la mónada coordina silenciosamente todos los efectos secundarios necesarios. Esto nos da código más limpio y componible.

Otro punto de vista es considerar `M A` como un contenedor o contexto que envuelve un valor (o valores) de tipo `A`, junto con alguna estructura adicional proporcionada por `M`. `return` coloca un valor en este contexto, mientras que `_>>=_` aplica una función a un valor en un contexto preservando la estructura circundante. Las leyes de mónada aseguran que esta estructura se maneja de una manera sensata y consistente.

De hecho, muchas estructuras comunes en programación pueden verse como casos especiales de mónadas:

Abstracción	Descripción	Leyes
Functor	Mapea funciones sobre estructuras	map id ≡ idmap (g . f) ≡ map g . map f
Applicative	Permite aplicar funciones dentro de estructuras	pure id <*> v ≡ vpure f <*> pure x ≡ pure (f x)u <*> pure y ≡ pure ($ y) <*> uu <*> (v <*> w) ≡ pure (.) <*> u <*> v <*> w
Monad	Secuencia cómputos dependientes dentro de estructuras	return a >>= k ≡ k am >>= return ≡ m(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)
Arrow	Generaliza funciones para soportar entradas y salidas múltiples	arr id ≡ idarr (f >>> g) ≡ arr f >>> arr gfirst (arr f) ≡ arr (first f)first (f >>> g) ≡ first f >>> first gfirst f >>> arr fst ≡ arr fst >>> ffirst f >>> arr (id *** g) ≡ arr (id *** g) >>> first f
Monoid	Tipo con una operación asociativa y una identidad	mempty <> x ≡ xx <> mempty ≡ x(x <> y) <> z ≡ x <> (y <> z)

Entonces las mónadas realmente son un patrón muy general y fundamental. Entender cómo encajan todas estas variaciones puede ser muy esclarecedor.

Pero quizás la mejor manera de desarrollar una intuición para las mónadas es usarlas mucho y en diferentes contextos. Con cada nueva instancia monádica que encuentres, pregúntate: ¿cuál es el superpoder computacional que está proporcionando? ¿Cómo abstrae los detalles y reduce la complejidad accidental? ¿Qué patrones de código permite? Con el tiempo, comenzarás a reconocer la "forma" monádica en muchos lugares, tanto dentro como fuera de la programación.

Y no te preocupes mucho si las mónadas no hacen clic de inmediato. Como cualquier abstracción poderosa, puede llevar tiempo apreciar completamente por qué son útiles y cuándo aplicarlas. Pero el viaje de aprendizaje vale la pena, ya que las mónadas verdaderamente representan una de las grandes ideas en programación.

6.3. Mónadas en Agda

6.3.1. La interfaz Monad en la librería estándar de Agda

La librería estándar de Agda define la interfaz `Monad` de manera muy similar a como la presentamos anteriormente:

```
record Monad (M : Set → Set) : Set₁ where
  field
    return : {A : Set} → A → M A
    _>>=_  : {A B : Set} → M A → (A → M B) → M B
```

Además, proporciona algunas funciones auxiliares útiles:

```
module Monad (M : Set → Set) ⦃ Mon : Monad M ⦄ where
  _>>_ : {A B : Set} → M A → M B → M B
  _>>_ ma mb = ma >>= λ _ → mb
  _=<<_ : {A B : Set} → (A → M B) → M A → M B
  f =<< c = c >>= f
  _>=>_ : {A B C : Set} → (A → M B) → (B → M C) → (A → M C)
  f >=> g = λ a → (f a >>= g)
  join : {A : Set} → M (M A) → M A
  join mma = mma >>= id
  mapM : {A B : Set} → (A → M B) → List A → M (List B)
  mapM f []       = return []
  mapM f (x ∷ xs) = do
    y  ← f x
    ys ← mapM f xs
    return (y ∷ ys)
  forM : {A B : Set} → List A → (A → M B) → M (List B)
  forM = flip mapM
  sequence : {A : Set} → List (M A) → M (List A)
```

La función `_>>_` descarta el resultado de la primera computación y sólo ejecuta los efectos. Por su parte, `=<<` es `_>>=_` con sus argumentos invertidos.

El operador `_>=>_` (pronunciado "fish") compone dos funciones que producen valores monádicos, pasando el resultado de la primera como entrada a la segunda. Esto permite encadenar cómputos monádicos de una manera que se asemeja al flujo de control imperativo.

La función `join` colapsa dos capas de estructura monádica en una, `mapM` aplica una función monádica sobre una lista y `sequence` convierte una lista de cómputos monádicos en un cómputo monádico de una lista. En conjunto, estas funciones permiten una forma limitada de recorrido ("traverse") monádico.

Agda también define azúcar sintáctico para trabajar con mónadas de una manera más legible:

```
syntax _>>=_ x (λ x → y) = x >>= y
do_then_ : ∀{a b} → M a → M b → M b
```

```
do x then y = x >> y
_>>=do_ : ∀{a b} → M a → (a → M b) → M b
x >>=do f = x >>= f
infixr 1 do_then_ _>>=do_
```

La notación `do` hace que el encadenamiento y anidamiento de cómputos monádicos sea mucho más conveniente:

```
postInc : State Nat Unit
postInc = do
  i ← get
  put (suc i)
```

Internamente, este azúcar simplemente se traduce a las operaciones `_>>=_` y `_>>_` correspondientes.

Estas diversas facilidades sintácticas hacen que trabajar con mónadas en Agda sea bastante agradable. Sin embargo, la biblioteca estándar actualmente no proporciona muchas herramientas para definir mónadas genéricas o transformadores de mónadas, por lo que usualmente se requiere algo de plomería adicional en comparación con lenguajes como Haskell.

Una limitación de la interfaz `Monad` en Agda es que requiere que los cómputos devuelvan un valor en `Set`, el universo de tipos "pequeños". Esto excluye tipos de orden superior (Tipos que pueden tomar otros tipos como parámetros) como `Set → Set`. Existen variantes de `RawMonad` y `RawMonadT` en `Category.Monad.Indexed` que relajan esta restricción, pero requieren un poco más de anotación de tipos.

Otra diferencia con Haskell es que Agda no tiene una restricción de clase `Applicative` separada. Se espera que las instancias de `Monad` también proporcionen `pure` y `_<*>_` que satisfacen las leyes de aplicativos. Esto se debe a que en Agda, a diferencia de Haskell, es posible definir `return` y `_>>=_` en términos de `pure` y `_<*>_`.

En general, la presencia de tipos dependientes complica un poco la situación de las mónadas en Agda en comparación con Haskell. No se puede asumir que sólo porque algo tiene el tipo correcto, automáticamente obedece las leyes de mónada. También es necesario escribir pruebas para las propiedades que se espera que se mantengan.

Pero el sistema de tipos más rico de Agda también permite expresar invariantes más precisas y empujar más verificación a tiempo de compilación. Con un poco de trabajo extra, es posible hacer que el sistema de tipos garantice no sólo que el código está bien escrito, sino que realmente hace lo que se espera. Esta es una de las grandes promesas de la programación dependientemente tipada.

6.3.2. Creación de nuevas instancias monádicas

Para utilizar la interfaz `Monad`, es necesario proporcionar instancias para tipos específicos. A continuación se muestra cómo hacerlo para `Maybe`:

```
data Maybe (A : Set) : Set where
```

```
  nothing : Maybe A
  just    : A → Maybe A
maybe : {A B : Set} → B → (A → B) → Maybe A → B
maybe z f nothing  = z
maybe z f (just x) = f x
instance
  MonadMaybe : Monad Maybe
  MonadMaybe = record
    { return = just
    ; _>>=_  = maybe nothing
    }
```

Aquí, `return` simplemente envuelve un valor en `Maybe` usando el constructor `just`, mientras que `_>>=_` propaga `nothing` en caso de falla, y de lo contrario aplica la función al valor envuelto utilizando `maybe`.

De manera similar, es posible definir instancias para otros tipos como `List`, `IO`, `State`, etc.:

```
MonadList : Monad List
MonadList = record
  { return = [_]
  ; _>>=_  = concatMap
  }
  where concatMap : {A B : Set} → (A → List B) → List A → List B
        concatMap f []       = []
        concatMap f (x :: xs) = f x ++ concatMap f xs
postulate
  IO : Set → Set
  returnIO : {A : Set} → A → IO A
  bindIO : {A B : Set} → IO A → (A → IO B) → IO B
instance
  MonadIO : Monad IO
  MonadIO = record
    { return = returnIO
    ; _>>=_  = bindIO
    }
record State (S A : Set) : Set where
  constructor state
  field
    runState : S → A × S
```

```
open State
instance
  MonadState : {S : Set} → Monad (State S)
  MonadState = record
    { return = λ x → state (λ s → x , s)
    ; _>>=_  = λ sa f → state (λ s →
        let (a , s') = runState sa s
        in  runState (f a) s')
    }
```

La instancia de lista encapsula un único valor como una lista singleton (Tipo de lista que contiene exactamente un valor, útil para restricciones estáticas) para `return` y concatena los resultados de mapear la función de encuadernación sobre la lista de entrada para `_>>=_`. La instancia `IO` simplemente llama a las primitivas `returnIO` y `bindIO` correspondientes (que se dejan sin definir como postulados por simplicidad). La instancia `State` encapsula valores puros como computaciones de estado que devuelven el valor y dejan el estado sin cambios, y compone computaciones de estado pasando el valor y el estado actualizados de la primera a la segunda.

Una vez que se tiene una instancia `Monad` para un tipo, se obtiene toda la funcionalidad y notación genérica que se vio anteriormente de forma gratuita. Esto permite escribir código abstracto y reutilizable que funciona con cualquier mónada.

Por supuesto, cada nueva instancia de mónada debe venir con pruebas de que obedece a las leyes de mónada. Para `Maybe`, se verían así:

```
maybe-left-id : {A B : Set} → (x : A) (f : A → Maybe B)
              → (just x >>= f) ≡ f x
maybe-left-id x f = refl
maybe-right-id : {A : Set} → (mx : Maybe A)
               → (mx >>= just) ≡ mx
maybe-right-id (just x) = refl
maybe-right-id nothing = refl
maybe-assoc : {A B C : Set} → (mx : Maybe A)
                            → (f : A → Maybe B) (g : B → Maybe C)
            → ((mx >>= f) >>= g) ≡ (mx >>= (λ x → f x >>= g))
maybe-assoc (just x) f g = refl
maybe-assoc nothing f g = refl
```

Aquí se utiliza el hecho de que `just` y `nothing` forman un álgebra inicial para `Maybe`. Patrones similares aplicarían para verificar otras instancias como `List` y `State`.

Para `IO` sería necesaria una noción de igualdad más refinada que `_≡_`, ya que dos cómputos `IO` pueden ser semánticamente equivalentes sin ser estructuralmente idénticos. Usualmente se maneja postulando una relación de igualdad apropiada.

En general, definir una instancia de `Monad` en Agda requiere un poco más de trabajo manual que en Haskell, debido al énfasis de Agda en la corrección demostrable. Sin embargo, también se obtienen beneficios de claridad conceptual y verificación automática que pueden valer la pena el esfuerzo adicional, especialmente en el contexto de código crítico para la seguridad o sensible a errores.

6.4. Composición y transformación de mónadas

Las mónadas son una herramienta poderosa en la programación funcional, y una de sus características más destacadas es la capacidad de componerse y transformarse. En esta sección, exploraremos cómo podemos combinar mónadas para crear efectos más complejos y cómo podemos agregar funcionalidad adicional a las mónadas existentes mediante transformadores de mónada.

6.4.1. Composición de mónadas

Imagina que tienes dos tipos monádicos, `M` y `N`, cada uno con sus propias características y comportamientos. ¿No sería genial si pudiéramos combinarlos en una nueva mónada `M ∘ N` que tenga la funcionalidad de ambos? Pues bien, ¡eso es exactamente lo que nos permite hacer la composición de mónadas!

En Agda, podemos expresar esta composición utilizando un tipo de datos parametrizado:

```
1  record _∘_ (M N : Set → Set) (A : Set) : Set where
2    constructor comp
3    field uncomp : M (N A)
4  infixr 10 _∘_
```

Luego, siempre que tanto `M` como `N` sean mónadas, podemos proporcionar una instancia de `Monad` para `M ∘ N`:

```
1  module _ {M N : Set → Set} ⦃ MonadM : Monad M ⦄ ⦃ MonadN : Monad N ⦄ where
2    instance
3      MonadT : Monad (M ∘ N)
4      MonadT = record
5        { return = comp ∘ return ∘ return
6        ; _>>=_  = λ mnx f →  comp $ uncomp mnx >>= λ nx →
7                     uncomp $ nx >>= λ x → uncomp $ f x
8        }
```

Esta capacidad de componer mónadas nos permite apilar efectos como si fueran bloques de construcción. Por ejemplo, `State S ∘ Either E` representa computaciones que pueden mantener un estado y lanzar excepciones, mientras que `IO ∘ List` nos da computaciones no deterministas con efectos secundarios.

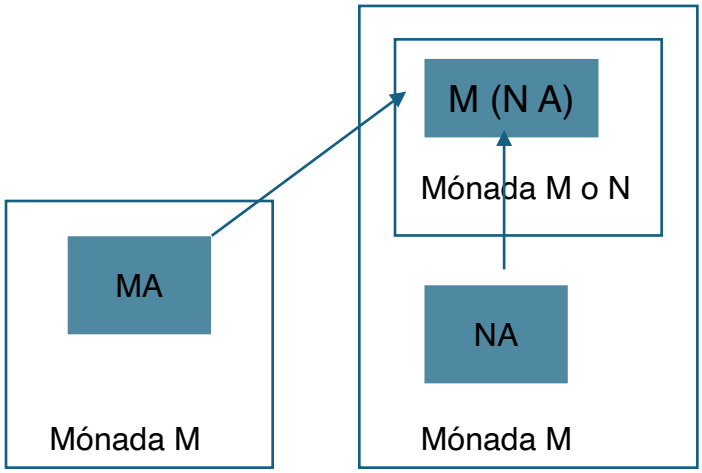


Imagen 12 - Composición de Mónadas.

La implementación de `return` para la mónada compuesta simplemente envuelve el valor dado en los contextos `return` de ambas mónadas base. Por otro lado, la implementación de `_>>=_` primero desenvuelve el valor de la mónada exterior, luego desenvuelve el valor resultante de la mónada interior, aplica la función de enlace a ese valor puro y vuelve a envolver el resultado en ambas mónadas.

Esta capacidad de construir mónadas más grandes y complejas a partir de partes más simples y enfocadas nos permite un estilo de programación algebraico muy expresivo y componible. Sin embargo, es importante tener en cuenta algunas advertencias:

- No todas las mónadas se componen de manera limpia, ya que las leyes de mónada para el compuesto pueden no cumplirse en general.
- Es necesario elegir un orden de envoltura, y diferentes elecciones pueden dar lugar a semánticas sutilmente distintas.
- El uso directo de mónadas compuestas puede llevar a un código bastante verboso y difícil de leer, debido a todos los envoltores `comp` y `uncomp` necesarios.

A pesar de estas limitaciones, la composición de mónadas sigue siendo una herramienta poderosa para construir abstracciones complejas a partir de componentes simples. Demuestra el poder del razonamiento algebraico y la reutilización en la programación funcional.

Transformador de Mónada	Explicación	Operaciones Adicionales
StateT s m a	Agrega estado mutable s a la mónada base m	get : StateT s m s put : s → StateT s m ()
ReaderT r m a	Agrega un entorno de solo lectura r a la mónada base m	ask : ReaderT r m r local : (r → r) → ReaderT r m a → ReaderT r m a
WriterT w m a	Agrega una salida auxiliar w (usualmente un monoide) a la mónada base m	tell : w → WriterT w m () listen : WriterT w m a → WriterT w m (a, w)

ExceptT e m a	Agrega manejo de errores con excepciones de tipo e a la mónada base m	throwE : e → ExceptT e m a catchE : ExceptT e m a → (e → ExceptT e' m a) → ExceptT e' m a
ContT r m a	Agrega continuaciones con resultado final de tipo r a la mónada base m	callCC : ((a → ContT r m b) → ContT r m a) → ContT r m a

6.4.2. Transformadores de mónada



Otra técnica común y poderosa es definir "transformadores de mónada" independientes que agregan un solo efecto a una mónada base arbitraria. Estos transformadores nos permiten agregar funcionalidad adicional a las mónadas existentes de una manera modular y componible.

Por ejemplo, a continuación, se muestra un transformador de mónada para agregar manejo de excepciones:

```
1  record ExceptT (E : Set) (M : Set → Set) (A : Set) : Set where
2    constructor mkExceptT
3    field runExceptT : M (Either E A)
4  except : {E A : Set} {M : Set → Set} ⦃ MonadM : Monad M ⦄
5         → Either E A → ExceptT E M A
6  except x = mkExceptT (return x)
7  module _ {E : Set} {M : Set → Set} ⦃ MonadM : Monad M ⦄ where
8    instance
9      MonadExceptT : Monad (ExceptT E M)
10     MonadExceptT = record
11       { return = except ∘ right
12       ; _>>=_  = λ ma f → mkExceptT $ do
13           x ← runExceptT ma
14           case x of λ {
15             (left e)  → return (left e)
16           ; (right a) → runExceptT (f a) }
17       }
```

El tipo `ExceptT E M A` representa una computación en la mónada `M` que puede lanzar una excepción de tipo `E` o producir un valor de tipo `A`. Su implementación almacena una computación `M (Either E A)` envuelta.

La instancia `Monad` para `ExceptT E M` propaga la estructura monádica de `M` de una manera que agrega un cortocircuito de los efectos `M` en caso de una excepción. `return` simplemente envuelve el valor dado en el constructor `Right` de `Either` y lo coloca en el contexto `M` utilizando `return`, mientras que `_>>=_` primero ejecuta la computación monádica excepcional de la izquierda, inspecciona su resultado y, en caso de una excepción, la vuelve a envolver, y en caso de un valor exitoso, aplica la función de enlace para obtener una nueva computación monádica excepcional.

Otros transformadores de mónada comunes, como `StateT`, `ReaderT`, `WriterT`, etc., siguen un patrón similar. Envuelven una computación monádica del tipo base `M` con un efecto adicional y definen `return` y `_>>=_` de una manera que agrega ese efecto mientras preserva el comportamiento original de `M`.

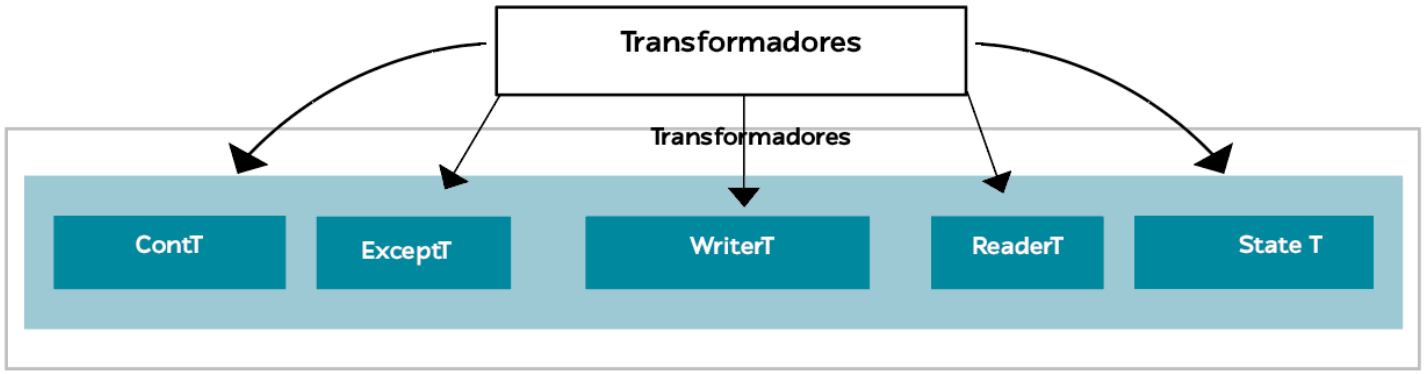


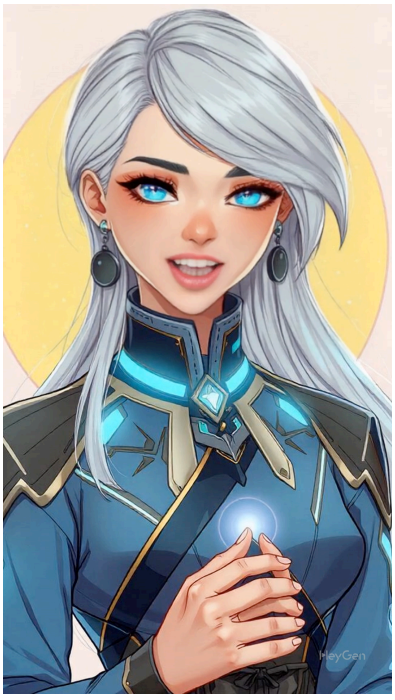
Imagen 13 - Transformadores de Mónada Comunes.
(Fuente: Propi

Las mónadas son una herramienta increíblemente poderosa y flexible en la programación funcional. La capacidad de componerlas y transformarlas nos permite construir abstracciones complejas a partir de componentes simples y reutilizables. Al reconocer patrones comunes y factorizarlos en interfaces genéricas, podemos evitar reinventar la rueda y, en su lugar, ensamblar las herramientas ideales a partir de piezas estándar.

Conclusiones del Capítulo
En este capítulo, exploramos las mónadas en Agda, desde su motivación hasta su implementación práctica. Cubrimos la definición categórica y las leyes, junto con el uso de interfaces estándar como `Monad`, `MonadFail`, `MonadPlus`, y la notación `do`. También examinamos la creación de instancias monádicas personalizadas y técnicas avanzadas como transformadores y composición. Se presentaron ejemplos de mónadas como `Maybe`, `List`, `State`, `Reader`, `Writer`, `IO`, mostrando diferentes estilos de programación y efectos computacionales.

Ejercicios Propuestos	
1	Implementa la mónada `Writer` en Agda. Asegúrate de que cumple con las leyes monádicas y proporciona pruebas formales de estas leyes.
2	Implementa el transformador de mónadas `ReaderT`. Demuestra cómo se puede usar para agregar un entorno de solo lectura a cualquier mónada existente.
3	Define una mónada libre para un conjunto simple de operaciones (por ejemplo, operaciones de estado `get` y `put`). Implementa un intérprete para esta mónada libre.
4	Implementa la composición de dos mónadas arbitrarias. Demuestra que la composición resultante también es una mónada proporcionando pruebas de las leyes monádicas.
5	Define un tipo de datos para representar un lenguaje de efectos simple (por ejemplo, con operaciones de estado y excepciones). Implementa una mónada que interprete este lenguaje de efectos.
6	Implementa una versión indexada de la mónada `State` donde el tipo del estado puede cambiar. Usa esto para implementar un pequeño lenguaje tipado con un entorno de tipado que evoluciona.
7	Escribe pruebas formales para demostrar propiedades de programas monádicos, como que ciertas secuencias de operaciones producen resultados específicos o mantienen ciertos invariantes.

Capítulo 7: Verificación y pruebas en Agda



Objetivos del Capítulo

En este capítulo, exploraremos la fascinante área de la programación verificada usando Agda. Aprenderás a aprovechar el poderoso sistema de tipos de Agda para especificar y probar propiedades de tu código, desde algoritmos clásicos como ordenamiento hasta aplicaciones más exóticas en álgebra abstracta y criptografía. Desarrollarás habilidades para traducir especificaciones informales a definiciones formales precisas y dominarás técnicas de prueba interactiva usando el asistente de pruebas integrado de Agda. También veremos cómo integrar componentes verificados con código no verificado mediante la extracción. Al final, estarás equipado con las herramientas conceptuales y prácticas para escribir tus propias pruebas en Agda, viendo la programación bajo una nueva luz como un medio para el razonamiento lógico y la construcción de conocimiento replicable.

7.1. Introducción

A lo largo de este libro, hemos explorado el fascinante mundo de la programación funcional usando el lenguaje Agda. Hemos aprendido sobre su expresivo sistema de tipos, las funciones puras, los datos inmutables y muchas otras características poderosas. Estos conceptos fundamentales forman el núcleo del estilo de programación funcional y nos permiten escribir código elegante, componible y fácil de razonar.

Sin embargo, hasta ahora nos hemos enfocado principalmente en la especificación e implementación de programas. Aunque hemos aprovechado el sistema de tipos de Agda para guiar el diseño y atrapar muchas clases de errores, aún no hemos explotado al máximo su verdadero potencial. Uno de los aspectos más asombrosos de Agda es su capacidad de no solo escribir programas, sino de probar formalmente su corrección.

Imagina poder especificar con precisión las propiedades que tu código debe satisfacer, similar a cómo escribes tipos para funciones.

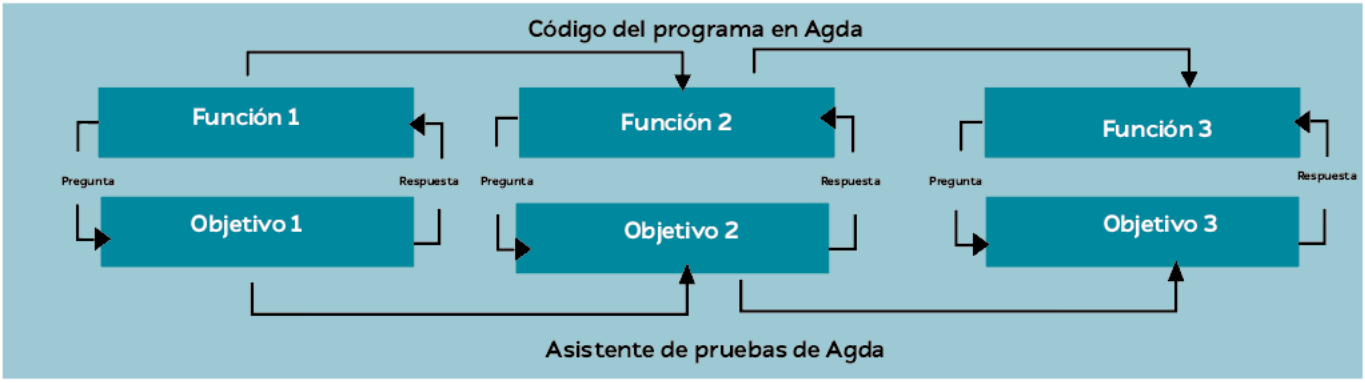


Imagen 14 - Interactuando con el Asistente de Pruebas de Agda.
(Fuente: Propia)

Pero en lugar de solo etiquetar las entradas y salidas, puedes hacer afirmaciones profundas y expresivas sobre el comportamiento y las relaciones.

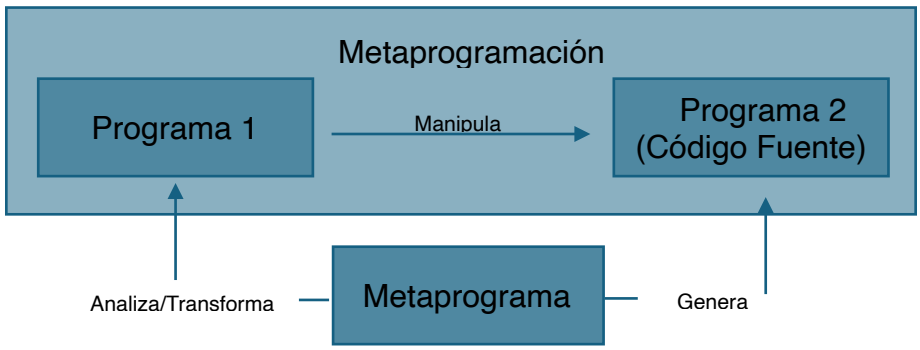


Imagen 15 - Metaprogramación: Programas que Manipulan Programas.
(Fuente: Propia)

Propiedades como "ordenar una lista preserva sus elementos", "este algoritmo de compresión es invertible por este descompresor" o "este solucionador SAT respeta las reglas de la lógica proposicional".

Ahora imagina realmente poder probar, con el rigor de las matemáticas, que tu implementación cumple esas especificaciones.

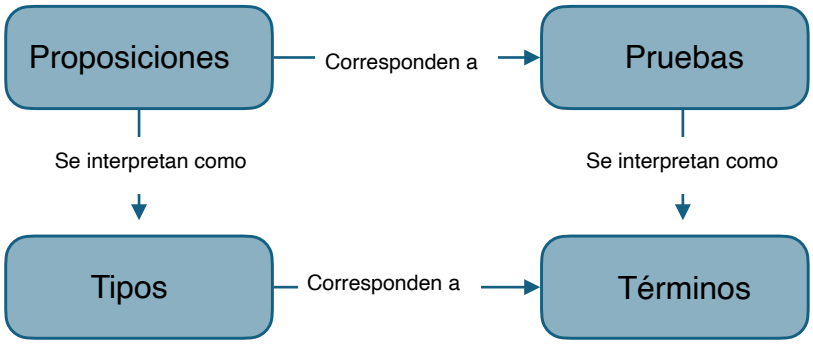


Imagen 16 - El Isomorfismo de Curry-Howard: Unificando Pruebas y Programas. (Fuente: Propia)

Es decir, construir un argumento lógico inapelable desde axiomas básicos y reglas de inferencia que establece sin lugar a dudas que tu código hace exactamente lo que dice. No más incertidumbre, no más pruebas insuficientes, no más rincones inexplorados, solo la garantía cristalina que se deriva de una prueba formal.

Esto es precisamente lo que permite el asistente de pruebas de Agda. Aprovechando la correspondencia de Curry-Howard entre programas y pruebas, podemos usar el mismo lenguaje y herramientas tanto para implementar funciones como para razonar sobre ellas. Podemos codificar especificaciones como tipos, escribir pruebas como términos y tener la seguridad de que la lógica subyacente de nuestros programas es sólida e incontestable.

En este capítulo, daremos un gran salto al reino de la programación verificada. Veremos cómo Agda nos permite declarar propiedades precisas que nuestro código debe cumplir y luego probar matemáticamente que nuestras implementaciones satisfacen esas propiedades. A través de la fusión de lógica y programación, podemos alcanzar un nuevo nivel de confiabilidad, robustez y claridad en nuestro software.

Algunas de las preguntas clave que abordaremos incluyen:

Pregunta	Descripción
¿Qué significa para un programa ser "correcto" y cómo podemos especificar esa corrección en Agda?	Se explora el concepto de corrección de un programa y cómo puede ser formalizado en Agda.
¿Cómo podemos usar los tipos para expresar propiedades y relaciones generales entre entradas y salidas?	Se discute cómo los tipos en Agda pueden utilizarse para expresar propiedades y relaciones entre datos de entrada y salida.
¿Cuál es el flujo de trabajo para construir interactivamente una prueba de un programa usando Agda?	Se describe el proceso paso a paso de construcción de una prueba interactiva en Agda.
¿Qué ventajas tienen las pruebas formales sobre las técnicas tradicionales basadas en testing?	Se analizan las ventajas de las pruebas formales en comparación con las pruebas tradicionales basadas en testing.
¿Cuáles son algunas de las compensaciones y limitaciones de la verificación en la práctica?	Se examinan las compensaciones y limitaciones asociadas con la verificación en la práctica.

A lo largo del camino, veremos una variedad de ejemplos, desde clásicos como ordenamiento y búsqueda hasta aplicaciones más inusuales en álgebra abstracta y criptografía. Ganaremos experiencia en traducir propiedades informales a especificaciones formales y desarrollaremos un repertorio de técnicas comunes de prueba. También

exploraremos cómo integrar componentes verificados con código no verificado mediante el mecanismo de extracción de Agda.

Al final de este capítulo, tendrás las herramientas conceptuales y prácticas para comenzar a escribir tus propias pruebas en Agda. Verás la programación bajo una nueva luz, no simplemente como una forma de instruir computadoras, sino como un medio para el razonamiento lógico preciso y la construcción de conocimiento demostrable.

7.2. Especificando propiedades

Antes de poder probar que un programa se comporta correctamente, necesitamos articular con precisión qué significa esa corrección. Es decir, debemos escribir especificaciones claras y completas de las propiedades que esperamos que satisfaga el programa. Estas especificaciones actuarán como los faros que guiarán el resto del proceso de verificación.

En la programación tradicional, las especificaciones a menudo toman la forma de documentación informal escrita en lenguaje natural, quizás complementada con algunos ejemplos o casos de prueba. Considera una función típica como la siguiente:

```
1  -- Ordena una lista de enteros en orden ascendente
2  sort :: [Int] -> [Int]
```

Su especificación podría verse así en Haskell:

```
1  -- | Ordena una lista de enteros en orden ascendente.
2  -- Ejemplos:
3  --   sort [3,1,4,1,5,9,2,6,5,3] == [1,1,2,3,3,4,5,5,6,9]
4  --   sort [1,2,3,4,5] == [1,2,3,4,5]
5  --   sort [] == []
6  sort :: [Int] -> [Int]
```

Aunque esto es mejor que nada, tales especificaciones informales tienen serias limitaciones. Son ambiguas, incompletas y propensas a inconsistencias. No están integradas con el sistema de tipos y no se verifican mecánicamente. En resumen, son una base débil para el razonamiento riguroso y las pruebas formales (Demostraciones matemáticas rigurosas de propiedades de programas usando lógica).

Especificación	Ventajas	Desventajas
Informal	- Fácil de escribir- No requiere conocimiento de sistemas formales	- Ambigua y parcial- Propensa a inconsistencias- No verificable por el compilador
Formal	- Precisa y completa- Verificable por el sistema de tipos- Permite razonamiento riguroso y pruebas	- Requiere más esfuerzo para escribir- Necesita familiaridad con lógica y sistemas de tipos

En cambio, en Agda podemos codificar especificaciones directamente como tipos, aprovechando toda la expresividad y maquinaria del sistema de tipos.

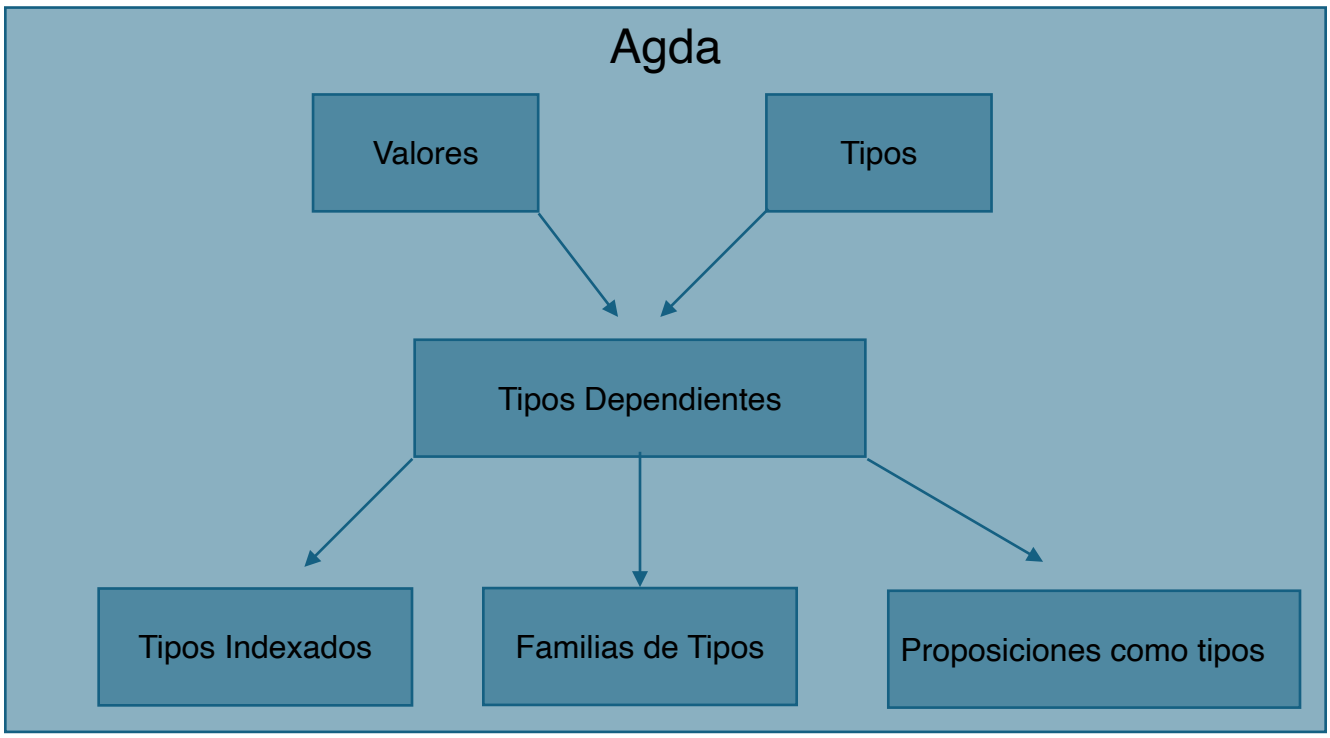


Imagen 17- Tipos Dependientes.

Estos tipos de especificación capturan relaciones y propiedades mucho más ricas que las simples etiquetas de tipos de Haskell. Son entidades de primera clase que podemos nombrar, combinar, computar y cuantificar.

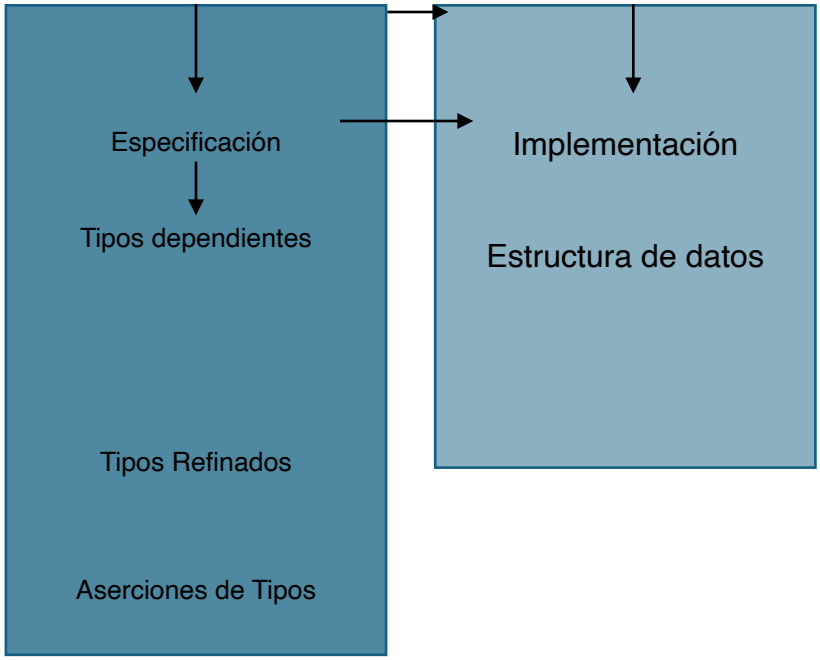


Imagen 18 - Especificaciones como Tipos en Agda.

Por ejemplo, aquí hay una posible especificación para la función de ordenamiento en Agda:

```
1  data Ordered {A : Set} : List A → Set where
2    nil  : Ordered []
3    cons : ∀ {x xs} → (y : A) → y ∈ xs → x ≤ y → Ordered (x :: xs) → Ordered (y :: x :: xs)
4  sorted : List Nat → Set
5  sorted []        = ⊤
6  sorted (x :: xs) = x ≤ xs × sorted xs
7  specification : (xs : List Nat) → Set
8  specification xs = Permutation xs (sort xs) × sorted (sort xs)
9  data Ordered {A : Set} : List A → Set where
```

Desgranemos esto pieza por pieza:

- `Ordered` es un predicado sobre listas que captura lo que significa que una lista esté en orden ascendente. Una lista vacía trivialmente está ordenada. Una lista no vacía está ordenada si y solo si su cabeza es menor o igual que todos los elementos restantes y su cola también está ordenada.
- `sorted` es una versión simplificada del predicado `Ordered` especializada para listas de números naturales. Establece que cada elemento debe ser menor o igual que el siguiente.
- `specification` es el tipo real que especifica la función `sort`. Dice que para cualquier lista de entrada `xs`, la salida `sort xs` debe ser una permutación de `xs` (es decir, contener los mismos elementos, posiblemente reordenados) y también debe estar ordenada.

Aquí _∈_, _≤_, _≈_, ⊤ son funciones y tipos definidos en la biblioteca estándar de Agda que capturan conceptos matemáticos comunes.

La belleza de esta especificación es que es completamente precisa y verificable. No hay ambigüedad sobre lo que constituye una lista ordenada o una implementación correcta de ordenamiento. Cualquier función que satisfaga el tipo `specification` está garantizada por construcción de ser un algoritmo de ordenamiento válido según nuestra definición.

Propiedad	Especificación en Tipos	Explicación
Ordenamiento	sorted : List Nat → Setsorted [] = ⊤sorted (x :: xs) = (x ≤ head xs) × sorted xs	Una lista está ordenada si está vacía o su cabeza es menor o igual que el primer elemento de su cola, y su cola está ordenada
Permutación	_≈_ : {A : Set} → List A → List A → Set[] ≈ [] = ⊤ x :: xs ≈ y :: ys = (x ≡ y) × (xs ≈ ys)_ ≈ _ = ⊥	Dos listas son permutaciones si tienen los mismos elementos, aunque posiblemente en diferente orden
Especificación de sort	specification : (xs : List Nat) → Setspecification xs = (xs ≈ sort xs) × sorted (sort xs)	Una función sort es correcta si su resultado es una permutación de la entrada y está ordenado

Por supuesto, llegar a especificaciones tan formales no siempre es fácil. A menudo requiere iteración y refinamiento, comenzando desde ideas vagas en lenguaje natural y gradualmente destilándolas en definiciones matemáticas precisas. Debemos balancear expresividad, simplicidad y facilidad de uso. Demasiado detalle se vuelve abrumador, mientras que muy poco conduce a pruebas débiles.

Un buen flujo de trabajo es comenzar escribiendo algunas propiedades clave como tipos, luego intentar probar esas propiedades para una implementación concreta. Cuando surjan obstáculos o lagunas en las especificaciones iniciales, podemos volver atrás y ajustarlas, en un ciclo continuo de refinamiento y retroalimentación.

Además de las propiedades específicas del dominio como el ordenamiento, a menudo queremos capturar propiedades genéricas pero importantes como equivalencia, idempotencia, invertibilidad, etc. Agda tiene un rico ecosistema de bibliotecas para expresar tales conceptos abstractos. Por ejemplo, aquí hay algunas especificaciones genéricas comunes de la biblioteca estándar:

```
1  _≡_ : {A : Set} → A → A → Set
2  x ≡ y = ∀ (P : A → Set) → P x → P y
3  Associative _·_ = ∀ x y z → (x · y) · z ≡ x · (y · z)
4  Idempotent _⊕_ = ∀ x → x ⊕ x ≡ x
5  Inverses _⁻¹ _·_ = ∀ x → x · x ⁻¹ ≡ ε × x ⁻¹ · x ≡ ε
```

Estas capturan, respectivamente, igualdad proposicional, asociatividad, idempotencia e invertibilidad. Están parametrizadas por los tipos y operaciones relevantes, haciéndolas ampliamente aplicables. Podemos instanciarlas y combinarlas con nuestras propias especificaciones para construir requisitos ricos y expresivos.

He aquí un ejemplo de especificar que la concatenación de listas es asociativa:

```
1  ++-assoc : {A : Set} → (xs ys zs : List A) → ((xs ++ ys) ++ zs) ≡ (xs ++ (ys ++ zs))
```

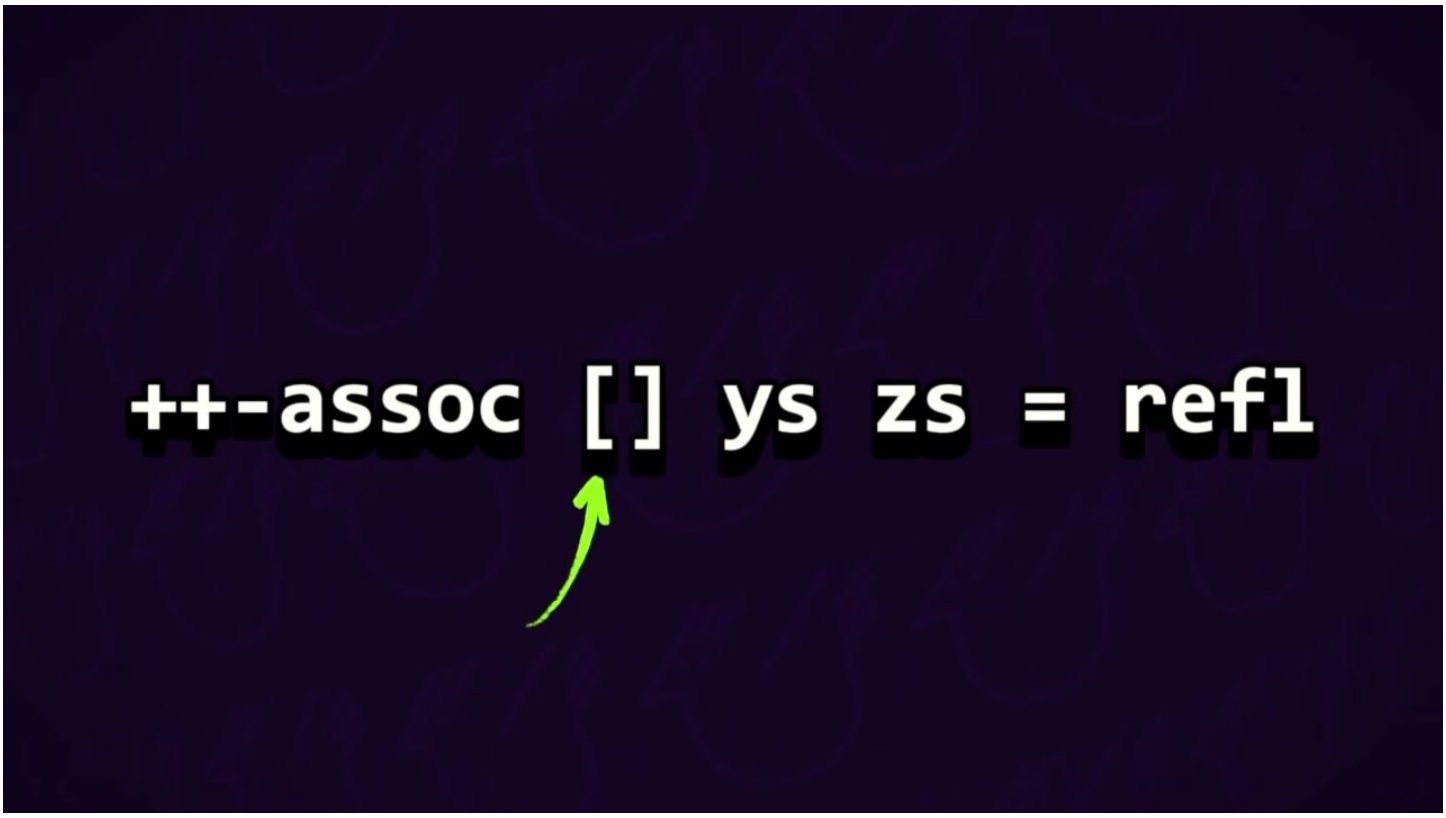
```
++-assoc xs ys zs = refl
++-assoc : {A : Set} → (xs ys zs : List A) → ((xs ++ ys) ++ zs) ≡ (xs ++ (ys ++ zs))
```

El tipo `++-assoc` establece que para cualesquiera listas `xs`, `ys`, y `zs`, concatenar `(xs ++ ys)` y `zs` es igual a concatenar `xs` y `(ys ++ zs)`. El término `refl` es una prueba trivial de reflexividad que completaremos más adelante.

Por ahora, lo principal a recordar es que en la programación con tipos dependientes, las especificaciones y los tipos son una y la misma cosa. Al definir un tipo, estamos haciendo una afirmación precisa sobre el comportamiento y las propiedades que deben cumplir sus habitantes. Esto se aplica tanto a los tipos completos de funciones como `specification` como a las declaraciones genéricas de propiedades como `++-assoc`.

En la siguiente sección, veremos cómo realmente probar que una pieza de código tiene el tipo especificado. Al escribir términos explícitos de prueba, podemos verificar mecánicamente que la implementación y la especificación realmente coinciden. Esto lleva las especificaciones de ser meros comentarios inteligentes a verdaderos contratos ejecutados por el compilador.

El camino hacia las especificaciones formales no siempre es fácil, pero los beneficios son inmensos. Con las herramientas y la mentalidad adecuadas, podemos crear islas de certeza lógica dentro del vasto océano de la ambigüedad del software. Tomando la pluma del poeta y el rigor del matemático, podemos escribir programas como proclamas inequívocas de propósito y verdad.



Video 10: Prueba de propiedad asociativa

7.3. Pruebas de propiedades

Una vez que hemos aprendido a escribir especificaciones en forma de tipos, el siguiente paso es probar que nuestro código cumple con esas especificaciones. En Agda, las pruebas son simplemente términos que habitan un tipo dado. Para verificar una propiedad, construimos explícitamente un término de ese tipo, que representa esencialmente una serie de pasos lógicos que conducen a la conclusión deseada.

Consideremos la propiedad asociativa de `_++_` del capítulo anterior:

```
1  ++-assoc : {A : Set} → (xs ys zs : List A) → ((xs ++ ys) ++ zs) ≡ (xs ++ (ys ++ zs))
2  ++-assoc xs ys zs = {! !}
```

Para completar la prueba, necesitamos reemplazar el marcador de posición `{! !}` con un término real del tipo:

```
1  ((xs ++ ys) ++ zs) ≡ (xs ++ (ys ++ zs))
```

En otras palabras, necesitamos proporcionar evidencia de que estos dos términos son realmente iguales. Afortunadamente, la definición de `_++_` nos da exactamente la información que necesitamos. Aquí está la prueba completa:

```
1  ++-assoc : {A : Set} → (xs ys zs : List A) → ((xs ++ ys) ++ zs) ≡ (xs ++ (ys ++ zs))
2  ++-assoc []       ys zs = refl
3  ++-assoc (x :: xs) ys zs = cong (x ::_) (++-assoc xs ys zs)
4  ++-assoc : {A : Set} → (xs ys zs : List A) → ((xs ++ ys) ++ zs) ≡ (xs ++ (ys ++ zs))
```

Esta prueba procede por inducción estructural en `xs`:

```
1  (x :: (xs ++ ys)) ++ zs ≡ x :: ((xs ++ ys) ++ zs)
```

La hipótesis inductiva nos dice que `(xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)`. Podemos aplicar `cong` para levantar esta igualdad al contexto `(x ::_)`, obteniendo exactamente la igualdad que necesitamos.

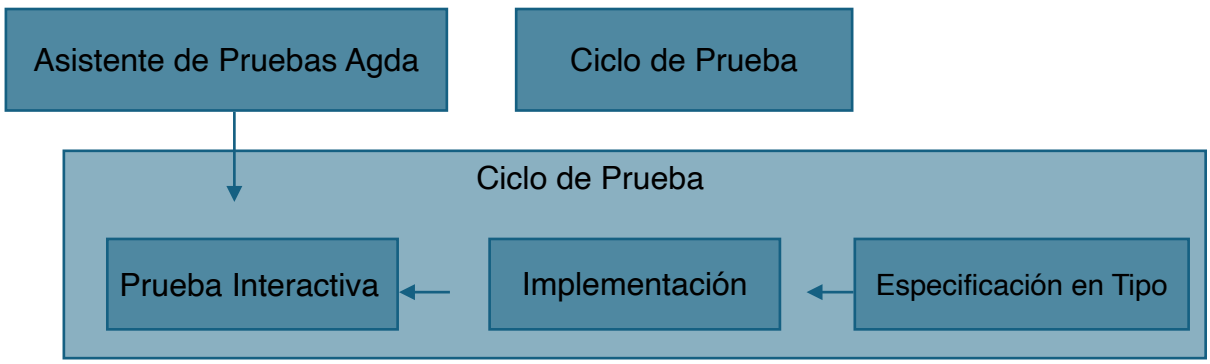


Imagen 19 - Proceso de Prueba Interactiva.

Aquí `refl` y `cong` son dos constructores de prueba fundamentales provistos por la biblioteca estándar:

- `refl` es una prueba de `x ≡ x` para cualquier `x`.
- `cong` toma una igualdad `x ≡ y` y una función `f`, y produce una prueba de `f x ≡ f y`.

Juntos, nos permiten construir pruebas complejas a partir de igualdades más simples.

Veamos otro ejemplo: una prueba de que invertir una lista dos veces produce la lista original.

```
1  reverse-involutive : {A : Set} → (xs : List A) → reverse (reverse xs) ≡ xs
2  reverse-involutive [] = refl
3  reverse-involutive (x :: xs) =
4    begin
5      reverse (reverse (x :: xs))
6    ≡⟨⟩
7      reverse (reverse xs ++ [ x ])
8    ≡⟨ cong (λ ys → reverse (ys ++ [ x ])) (reverse-involutive xs) ⟩
9      reverse (xs ++ [ x ])
10   ≡⟨ reverse-++-commute xs [ x ] ⟩
11     reverse [ x ] ++ xs
12   ≡⟨⟩
13     x :: xs
14   ∎
15 where
16 reverse-++-commute : {A : Set} → (xs ys : List A) →
17                     reverse (xs ++ ys) ≡ reverse ys ++ reverse xs
18 reverse-++-commute xs []       = refl
19 reverse-++-commute xs (y :: ys) =
20   begin
21     reverse ((xs ++ [ y ]) ++ ys)
22   ≡⟨ reverse-++-commute (xs ++ [ y ]) ys ⟩
23     reverse ys ++ reverse (xs ++ [ y ])
24   ≡⟨ cong (reverse ys ++_) (reverse-++-commute xs [ y ]) ⟩
25     reverse ys ++ (reverse [ y ] ++ reverse xs)
26   ≡⟨⟩
27     reverse ys ++ (y :: reverse xs)
```

```
28      ≡⟨⟩
29        reverse (y :: ys) ++ reverse xs
30      ∎
```

Aunque esta prueba también utiliza inducción, la cadena de razonamiento es más extensa. Usamos la sintaxis "`begin-_≡⟨_⟩_-∎`" para encadenar una secuencia de igualdades justificadas por varios subobjetivos y lemas auxiliares, como `reverse-++-commute`, que a su vez se prueba por inducción y cálculo.

El flujo general consiste en trabajar "de adentro hacia afuera", reduciendo la conclusión deseada a subobjetivos más simples utilizando ecuaciones y propiedades conocidas, hasta llegar a algo trivialmente cierto como `refl`. El asistente interactivo de pruebas de Agda resulta invaluable en este proceso, ya que muestra el estado actual de la prueba y el objetivo inmediato, permitiéndonos mantenernos encaminados y ver exactamente qué nos falta demostrar.

A medida que nuestras pruebas se vuelven más complejas, a menudo es útil extraer subpruebas reutilizables como lemas separados, como hicimos con `reverse-++-commute`. Esto mantiene limpio y modular el flujo de razonamiento. También podemos aprovechar bibliotecas como `Relation.Binary.PropositionalEquality`, que proporcionan combinadores de prueba adicionales para igualdad proposicional.

En general, construir una prueba es una habilidad que mejora con la práctica. Involucra una combinación de razonamiento hacia adelante desde hipótesis y definiciones, razonamiento hacia atrás desde el objetivo, y un instinto para encontrar el "punto dulce" entre los dos. Requiere creatividad para encontrar la ruta correcta, pero también disciplina mecánica para ejecutar los pasos necesarios.

Con el tiempo, uno desarrolla un repertorio de técnicas y estrategias para abordar diferentes situaciones de prueba. Patrones comunes como inducción estructural, cálculo ecuacional y razonamiento por casos surgen repetidamente. También aprendemos cuándo buscar bibliotecas existentes o resultados previamente demostrados para no reinventar la rueda.

Probar completamente propiedades de programas no triviales a menudo requiere un esfuerzo significativo. Sin embargo, los beneficios hacen que valga la pena:

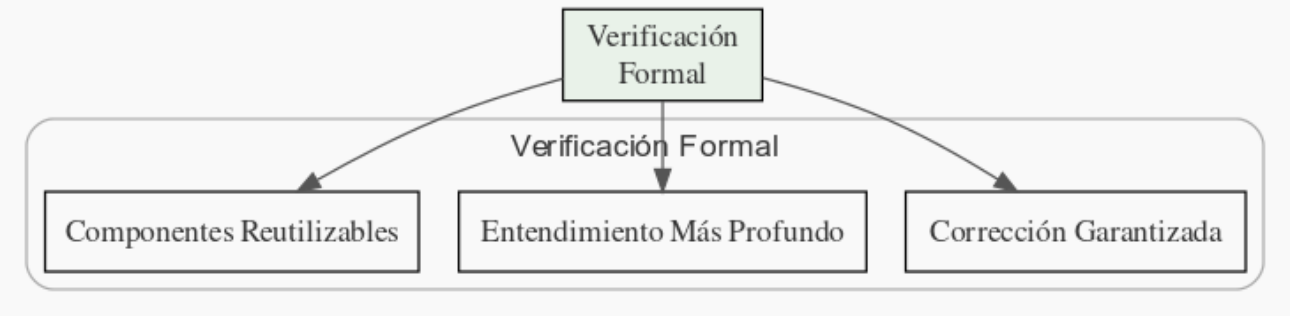


Imagen 20 - Ventajas de la Verificación Formal.
(Fuente: Propia)

La certeza absoluta de corrección, la comprensión más profunda ganada al pensar rigurosamente sobre el comportamiento, y la capacidad de reutilizar las pruebas como bloques de construcción para sistemas más grandes.

Sin embargo, escribir pruebas detalladas no siempre es práctico o necesario para cada pieza de código. En la siguiente sección, exploraremos algunas formas de incorporar el razonamiento especificaciones directamente en las definiciones de tipos usando tipos refinados. Al restringir el conjunto mismo de valores representables, podemos obtener ciertas garantías de corrección "gratis", sin escribir pruebas separadas.

7.4. Pruebas Formales

Las pruebas formales son una herramienta poderosa en el desarrollo de software que nos permite verificar matemáticamente la corrección de nuestros programas. En el contexto de la programación funcional y, específicamente, en Agda, las pruebas formales se integran de manera natural con el código, permitiéndonos desarrollar software correcto por construcción.

Una prueba formal es una demostración matemática rigurosa de que un programa cumple con su especificación. A diferencia de las pruebas tradicionales basadas en pruebas, que solo pueden mostrar la presencia de errores, pero no su ausencia, las pruebas formales pueden garantizar la ausencia de ciertos tipos de errores en todas las posibles ejecuciones del programa.

En Agda, las pruebas formales se escriben como funciones que habitan ciertos tipos. Esto se basa en el isomorfismo de Curry-Howard, que establece una correspondencia entre proposiciones lógicas y tipos, y entre pruebas y programas. La siguiente tabla ilustra esta correspondencia:

Lógica	Teoría de Tipos
Proposición	Tipo
Prueba	Término (programa)
Implicación ($P \to Q$)	Tipo función ($A \to B$)
Conjunción ($P \wedge Q$)	Tipo producto ($A \times B$)
Disyunción ($P \vee Q$)	Tipo suma ($A \uplus B$)
Verdadero ($\top$)	Tipo unidad ($\top$)
Falso ($\bot$)	Tipo vacío ($\bot$)

Veamos un ejemplo simple de cómo escribir una prueba formal en Agda. Probaremos que para cualquier número natural n, n + 0 = n.

```
1 open import Data.Nat
2 open import Relation.Binary.PropositionalEquality
3 +-right-identity : ∀ (n : ℕ) → n + 0 ≡ n
4 +-right-identity zero = refl
5 +-right-identity (suc n) = cong suc (+-right-identity n)
```

En este ejemplo, declaramos un teorema `+-right-identity` que afirma que para todo número natural `n`, `n + 0 ≡ n`. Probamos el teorema por inducción sobre `n`. En el caso base, cuando `n` es `zero`, `zero + 0 ≡ zero` por definición, lo cual probamos con `refl` (reflexividad). En el paso inductivo, asumimos que el teorema es cierto para `n` y lo probamos para `suc n`. Usamos `cong` para aplicar `suc` a ambos lados de la igualdad inductiva.

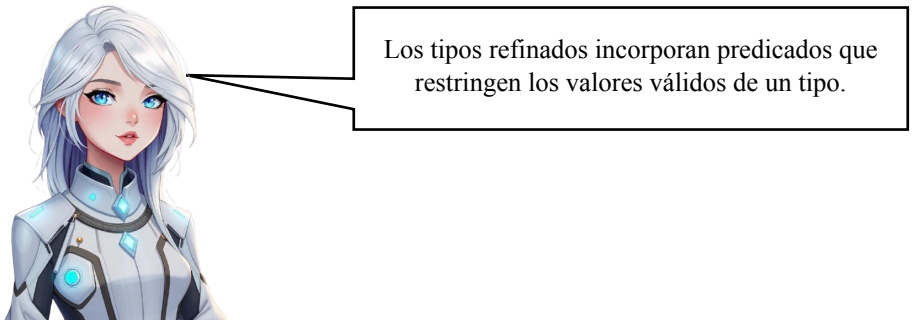
Las pruebas formales pueden volverse bastante complejas. Veamos un ejemplo más elaborado: probaremos que la suma de números naturales es asociativa.

```
+-assoc : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = refl
+-assoc (suc m) n p =
  begin
    (suc m + n) + p
  ≡⟨⟩
    suc (m + n) + p
  ≡⟨⟩
    suc ((m + n) + p)
  ≡⟨ cong suc (+-assoc m n p) ⟩
    suc (m + (n + p))
  ≡⟨⟩
    suc m + (n + p)
  ∎
```

En esta prueba, usamos la sintaxis de razonamiento ecuacional de Agda para hacer la prueba más legible. Cada paso de la prueba está justificado, ya sea por la definición de la suma (≡⟨⟩) o por la aplicación de la hipótesis inductiva (≡⟨ cong suc (+-assoc m n p) ⟩).

Una de las ventajas de usar Agda es que las pruebas y los programas coexisten en el mismo lenguaje. Esto significa que podemos usar nuestras pruebas para guiar el desarrollo de nuestros programas y viceversa.

7.5. Tipos refinados



Hasta ahora, hemos mantenido una clara separación entre los tipos de nuestros programas y sus especificaciones lógicas. Primero escribimos algunas definiciones de tipos y funciones, y luego añadimos por separado propiedades y pruebas sobre esas definiciones. Sin embargo, a veces es posible y beneficioso fusionar la especificación directamente en la definición del tipo utilizando tipos refinados.

Un tipo refinado es simplemente un tipo regular "decorado" con algunos predicados adicionales que cada valor del tipo debe satisfacer. Estos predicados actúan como una especie de contrato o invariante que restringe el conjunto de valores que se pueden construir. Al empaquetar propiedades adicionales directamente en la definición del tipo, obtenemos esas garantías "de fábrica" sin tener que escribir pruebas externas.

Por ejemplo, en lugar de un simple tipo `Int` para enteros, podría ser deseable tener un tipo `Pos` para enteros estrictamente positivos. He aquí una forma de definir tal tipo en Agda:

```
1  data Pos : Set where
2    pos : (n : ℕ) → {n≠0 : False (n ≟ 0)} → Pos
```

Aquí, `Pos` se define como un tipo de dato con un solo constructor `pos`. Para construir un valor de tipo `Pos`, necesitamos proporcionar un número natural `n` junto con una prueba `n≠0` de que `n` no es cero. El tipo `False` es el tipo vacío sin constructores, por lo que un valor de tipo `False (n ≟ 0)` solo existe si `n ≟ 0` es `false`, es decir, si `n` no es cero.

Con este tipo refinado, ahora es imposible incluso representar un entero no positivo, y mucho menos pasarlo a funciones que esperan un `Pos`. Intentar construir un valor `pos 0` resultará en un error de tipo, porque no hay forma de proporcionar la prueba requerida `False (0 ≟ 0)`. Las operaciones que esperan un `Pos` ahora pueden asumir con seguridad la positividad sin comprobaciones en tiempo de ejecución.

Veamos un ejemplo de una función segura de división para `Pos`:

```
1  _/_ : (n m : Pos) → Pos
2  pos n / pos m = pos (div n m)
```

Debido a que tanto `n` como `m` son de tipo `Pos`, sabemos que no pueden ser cero, por lo que la división está bien definida. No necesitamos una cláusula `if` alrededor de `div` para evitar una división por cero.

Los tipos refinados son especialmente efectivos para capturar invariantes de estructuras de datos. Por ejemplo, podríamos representar listas ordenadas como:

```
1  data OList (A : Set) : Set where
2    nil  : OList A
3    cons : (x : A) (xs : OList A) → (y : A) → y ∈ xs → x ≤ y → OList A
```

Aquí, el constructor `cons` requiere una prueba de que el elemento insertado `x` es menor o igual que la cabeza existente `y`. Esto asegura estructuralmente que todas las listas construidas con `cons` estén automáticamente ordenadas, sin necesidad de pruebas de ordenamiento post-hoc.

Otro ejemplo común es representar árboles binarios de búsqueda:

```
data BST (A : Set) : Set where
  leaf : BST A
  node : (x : A) (l r : BST A) →
         ((y : A) → y ∈ l → y < x) →
         ((z : A) → z ∈ r → x < z) →
         BST A
```

Cada nodo lleva consigo pruebas de que todos los elementos en el subárbol izquierdo son menores que la raíz `x`, y todos en el subárbol derecho son mayores que `x`. Nuevamente, esto hace imposible construir un valor de tipo `BST` que viole las invariantes de orden.

La verdadera potencia surge al combinar tipos refinados con otras características de tipos dependientes. Por ejemplo, podemos representar vectores (listas con longitud codificada por tipo) junto con un predicado de ordenamiento:

```
data Vec (A : Set) : ℕ → Set where
  []   : Vec A zero
  _::_ : {n : ℕ} (x : A) (xs : Vec A n) → Vec A (suc n)
data OVec (A : Set) : ℕ → Set where
  []   : OVec A zero
  _::_ : {n : ℕ} (x : A) (xs : OVec A n) → ((y : A) → y ∈ xs → x ≤ y) → OVec A (suc n)
data Vec (A : Set) : ℕ → Set where
```

Ahora tenemos listas ordenadas cuya longitud se conoce en tiempo de compilación. Podemos escribir funciones que sean genéricas en la longitud `n` sin perder la información de ordenamiento.

Si bien son poderosos, los tipos refinados tienen algunas compensaciones. Pueden hacer que las firmas de los tipos y las pruebas sean mucho más complicadas, potencialmente oscureciendo el flujo computacional subyacente. A menudo tenemos que sopesar la garantía adicional de corrección contra la complejidad adicional de implementación y uso.

Técnica	Ventajas	Desventajas
Especificación Informal	- Fácil de escribir y entender- No agrega complejidad sintáctica	- No verificable- Puede volverse inconsistente o desactualizada
Especificación Formal	- Precisión y completitud- Verificable por el sistema de tipos	- Requiere más esfuerzo- Puede oscurecer la intención
Pruebas Explícitas	- Máxima confianza en la corrección- Separa especificación de implementación	- Requiere escribir pruebas- Las pruebas pueden ser complejas
Tipos Refinados	- Integra especificación y implementación- Garantías desde los tipos	- Puede complicar los tipos- No siempre es posible o práctico

También es importante recordar que los tipos refinados, como cualquier forma de especificación, solo son significativos con respecto a alguna definición formal subyacente. Las propiedades que codifican no son axiomáticas, sino que derivan de cómo hemos elegido expresarlas en el lenguaje del asistente de pruebas. Como siempre, necesitamos asegurarnos de que esas definiciones realmente capturen el propósito previsto.

A pesar de estas advertencias, los tipos refinados siguen siendo una herramienta extremadamente potente y un área clave en la que los tipos dependientes brillan. Nos permiten elevar invariantes y contratos al nivel del sistema de tipos, obteniendo una seguridad adicional sin pruebas adicionales. "Hacen que los estados incorrectos sean irrepresentables", por así decirlo.

Al aprovechar los tipos refinados junto con pruebas explícitas, podemos especificar y verificar sistemas cada vez más complejos. En la siguiente sección presentaremos algunos ejemplos del mundo real de este flujo de trabajo integral, y cómo integrar componentes verificados con código no verificado utilizando el sistema de extracción de Agda.

7.6. Extracción de código verificado

Las técnicas de especificación y verificación cubiertas en este capítulo son extremadamente poderosas para construir software robusto y correcto. Sin embargo, toda esta maquinaria de tipos y pruebas puede ser computacionalmente costosa de ejecutar directamente. Además, a menudo necesitamos integrar componentes verificados con bases de código existentes en lenguajes convencionales.

Afortunadamente, Agda admite la extracción, que nos permite traducir programas de Agda a código en un lenguaje anfitrión más tradicional como Haskell, JavaScript o Scheme.

La idea básica es separar las partes computacionales de un programa (que contribuyen al comportamiento en tiempo de ejecución) de las partes meramente lógicas o de prueba (relevantes solo para la verificación).

Durante la extracción, todas las computaciones que viven únicamente en "fragmentos de prueba" del programa se borran, dejando solo el núcleo computacional crítico. Más precisamente, cualquier término cuyo tipo es una mera Proposición (un tipo con al más un habitante irrelevante) puede ser borrado de forma segura. El resultado es un programa con tipos simples en el lenguaje anfitrión que preserva el comportamiento original en tiempo de ejecución.

Consideremos el siguiente ejemplo simple. Supongamos que hemos definido y verificado una función para duplicar un entero:

```
1  double : ℕ → ℕ
2  double zero    = zero
3  double (suc n) = suc (suc (double n))
4  double-prop : (n : ℕ) → double n ≡ n + n
5  double-prop zero    = refl
6  double-prop (suc n) = cong suc (cong suc (double-prop n))
```

Aquí, `double-prop` prueba que `double n` siempre es igual a `n + n`. Sin embargo, aunque proporciona una garantía valiosa, `double-prop` no tiene ningún efecto computacional. Todo lo que importa para el tiempo de ejecución es la definición original de `double`.

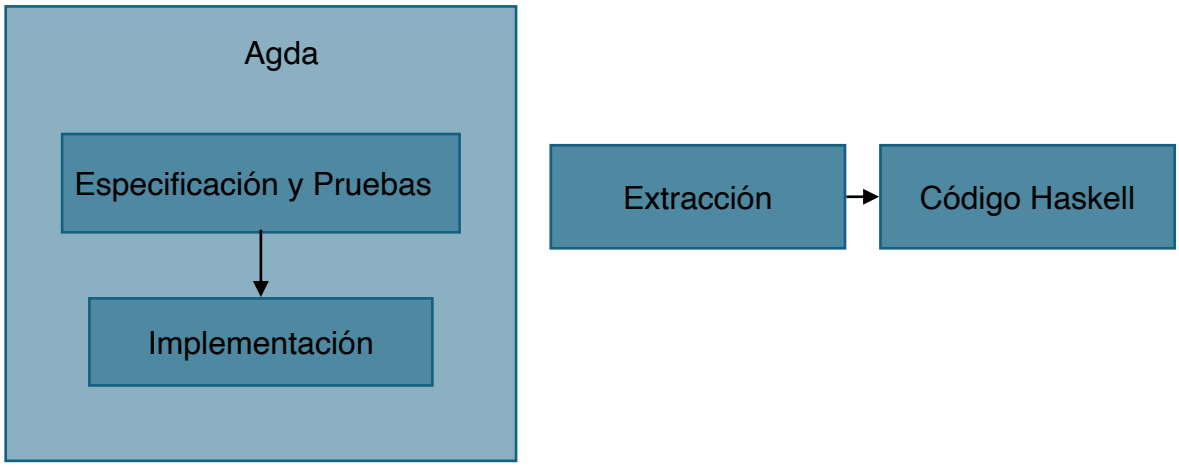


Imagen 21 - Extracción de Código Verificado.

Podemos instruir a Agda para que extraiga solo el fragmento computacional relevante especificando una directiva de extracción como:

```
{-# COMPILE JS double = function (n) { return n * 2; } #-}
```

Esto le indica a Agda que traduzca `double` a la función JavaScript correspondiente que duplica su argumento. La prueba `double-prop` se borra por completo durante la extracción, ya que no contribuye al comportamiento en tiempo de ejecución.

Veamos un ejemplo más sustancial. Recordemos el tipo `OList` para listas ordenadas de la sección anterior:

```
data OList (A : Set) : Set where
  nil  : OList A
  cons : (x : A) (xs : OList A) → (y : A) → y ∈ xs → x ≤ y → OList A
```

Técnica	Propósito	Ejemplo en Agda
Tipos Refinados	Capturar invariantes y restricciones directamente en los tipos	data OList (A : Set) : Set where [] : OList A cons : (x : A) (xs : OList A) → (y : A) → y ∈ xs → x ≤ y → OList A
Aserciones de Tipos	Especificar los tipos esperados de los argumentos y resultados	head : {A : Set} → OList A → A head (cons x _ _ _ _) = x
Comprobaciones en Tiempo de Ejecución	Verificar condiciones que dependen de valores en tiempo de ejecución	safeDiv : (x y : ℕ) → {y≠0 : False (y =? 0)} → ℕ safeDiv x y {y≠0} = div x y

Pruebas de Teoremas	Demostrar propiedades de los programas dentro del sistema de tipos	++-assoc : {A : Set} (xs ys zs : List A) → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs) ++-assoc [] ys zs = refl ++-assoc (x :: xs) ys zs = cong (x ::_) (++-assoc xs ys zs)

Supongamos que queremos extraer funciones que operan en `OList` a Haskell. Primero, podemos definir una traducción de `OList` a un tipo de datos de Haskell:

```
{-# FOREIGN GHC type OList a = [] | a : OList a #-}
```

Esto mapea `OList` a un tipo algebráico de datos en Haskell, ignorando los predicados de prueba. Ahora podemos proporcionar traducciones para varias funciones:

```
{-# COMPILE GHC nil  = [] #-}
{-# COMPILE GHC cons = (:) #-}
{-# COMPILE GHC head = \case
    []     -> error "head: empty list"
    (x:xs) -> x
#-}
{-# COMPILE GHC tail = \case
    []     -> error "tail: empty list"
    (x:xs) -> xs
#-}
```

Aquí, los constructores `nil` y `cons` se traducen a `[]` y `(:)` respectivamente. Las funciones `head` y `tail` se implementan mediante coincidencia de patrones, con casos de error para listas vacías.

Observemos cómo todas las pruebas asociadas con `OList` (por ejemplo, que el elemento insertado es mayor que la cabeza) se borran por completo en el código Haskell extraído. Lo que queda es una implementación de lista enlazada completamente estándar. Sin embargo, debido a las propiedades probadas por separado, como la preservación del orden, podemos tener la certeza de que estas funciones extraídas son correctas con respecto a esas especificaciones.

El borrado de pruebas durante la extracción no solo reduce la sobrecarga, sino que también es crucial para la integración con el código existente. La mayoría de los lenguajes convencionales simplemente no tienen la maquinaria de tipos para representar las ricas especificaciones y pruebas que podemos expresar en Agda. Al borrar este andamiaje lógico, obtenemos componentes concretos que pueden interoperar a la perfección con bibliotecas y frameworks regulares.

Por supuesto, esto significa que el código extraído en sí mismo no lleva ningún rastro directo de haber sido verificado. Alguien que mira solo la traducción de Haskell no tiene forma de saber que hemos probado ciertas propiedades. La fuente de la verdad sigue siendo el desarrollo de Agda; la extracción es meramente un "aplanamiento" para fines prácticos.

Por lo tanto, al integrar componentes verificados con bases de código no verificadas, debemos tener cuidado de razonar solo a partir de las especificaciones escritas formalmente en Agda, en lugar de las traducciones potencialmente débiles.

Las pruebas en el mundo de los tipos dependientes no se transfieren automática o sintácticamente a los artefactos extraídos.

Aun así, la capacidad de aprovechar el poder de las matemáticas para desarrollar y verificar bibliotecas críticas, al mismo tiempo que producimos implementaciones optimizadas para el consumo convencional, es inmensamente valiosa. De hecho, este enfoque híbrido es exactamente cómo se construyen varios sistemas del mundo real en torno a los asistentes de pruebas.

Por ejemplo, el compilador optimizador de Haskell GHC utiliza este flujo de trabajo para sus fases centrales. El núcleo de GHC y varias de las optimizaciones clave están implementadas y verificadas en Coq, y luego extraídas a código Haskell equivalente. Los componentes verificados se integran sin problemas con el resto de la base de código Haskell, pero tienen la certeza adicional de la prueba formal.

A una escala aún mayor, toda la pila del kernel seL4, que incluye un micronúcleo, controladores de dispositivos, sistema de archivos y otros componentes, se verifica formalmente en Isabelle/HOL y se extrae a C ejecutable. Este es quizás el ejemplo más impresionante hasta la fecha del desarrollo de sistemas operativos de alto rendimiento con una base matemáticamente probada.

La extracción nos brinda lo mejor de ambos mundos: nos permite aprovechar toda la maquinaria expresiva de los asistentes de pruebas para especificar, implementar y verificar software, mientras generamos artefactos concretos y optimizados para el despliegue y la integración. Nos permite ser tanto matemáticos rigurosos como ingenieros prácticos.

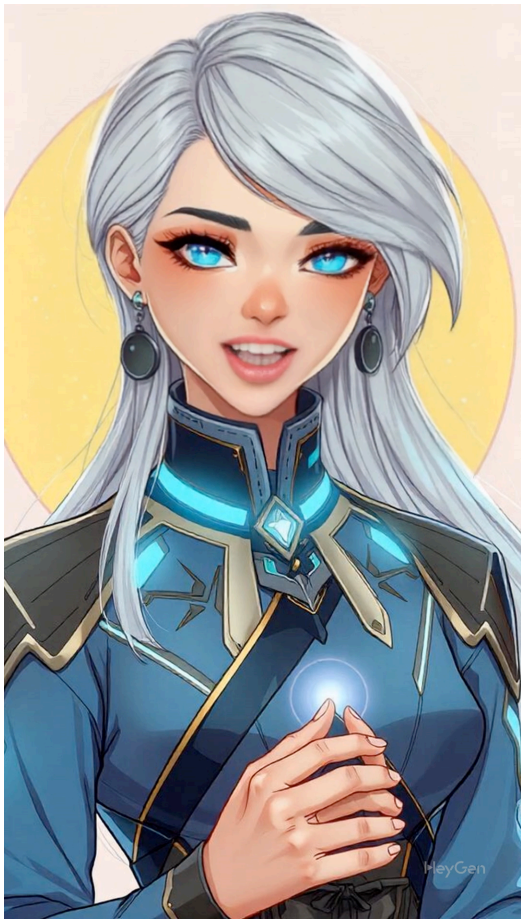
Al adoptar la mentalidad de la programación verificada, no estamos condenados a la torre de marfil de la teoría. En su lugar, contamos con las herramientas para construir puentes sólidos entre la especificación abstracta y la implementación concreta. Las pruebas se convierten no solo en garantías de corrección, sino en planos para construir sistemas del mundo real que son robustos, confiables y fundamentalmente sólidos.

Conclusiones del Capítulo
En este capítulo, exploramos las potentes capacidades de Agda para la especificación y verificación. Aprendimos a codificar propiedades como tipos, escribir pruebas como términos y obtener certeza mediante el razonamiento matemático riguroso. Utilizamos tipos refinados y dependientes para expresar especificaciones, probándolas desde la inducción simple hasta cadenas de razonamiento más complejas. Además, ganamos una nueva perspectiva sobre la programación: no solo escribimos instrucciones, sino que construimos conocimiento replicable. Con estas herramientas, podemos abordar problemas complejos con confianza, construyendo sistemas garantizados por pruebas matemáticas y extrayendo implementaciones eficientes.

Ejercicios Propuestos	
1	Demuestra formalmente las propiedades de asociatividad y conmutatividad para la suma de números naturales en Agda.
2	Prueba que la función `reverse` (inversión de una lista) es involutiva, es decir, que `reverse (reverse xs) ≡ xs` para cualquier lista `xs`.

3	Implementa el algoritmo de ordenamiento por inserción y proporciona una prueba formal de su corrección (que el resultado está ordenado y es una permutación de la entrada).
4	Define una relación bien fundada en los números naturales y úsala para demostrar la terminación de la función de Ackermann.
5	Define un tipo `Vector` con longitud indexada y prueba que la concatenación de vectores preserva la longitud (es decir, que la longitud del resultado es la suma de las longitudes de los vectores de entrada).
6	Define un tipo de dato para árboles binarios de búsqueda y prueba que las operaciones de inserción y búsqueda mantienen la propiedad de orden del árbol.
7	Implementa y verifica un algoritmo simple (por ejemplo, factorial o fibonacci) en Agda. Luego, extrae el código a Haskell y demuestra que se ejecuta correctamente.
8	Prueba que no existe un número natural que sea tanto par como impar.
9	Demuestra que los tipos `A × (B + C)` y `(A × B) + (A × C)` son isomorfos, donde `×` representa el producto y `+` la suma de tipos.
10	Implementa una pequeña máquina de estados finitos usando la mónada `State` y prueba algunas propiedades sobre su comportamiento.

Capítulo 8: Temas avanzados en Agda



Objetivos del Capítulo

En este fascinante capítulo, nos adentraremos en los conceptos más avanzados y emocionantes de la programación funcional en Agda. Exploraremos abstracciones elegantes como mónadas y funtores, que nos permitirán estructurar y razonar sobre efectos y computaciones de manera clara y modular. Descubriremos cómo las técnicas de parsing y compilación nos abren las puertas a la creación de nuestros propios lenguajes e intérpretes. Aprovecharemos el poder de la metaprogramación para automatizar tareas y extender las capacidades expresivas de Agda. Además, incursionaremos en el mundo de la criptografía funcional, donde la seguridad y la corrección son primordiales. Prepárate para expandir los límites de lo que creías posible en el desarrollo de software funcional.

8.1. Introducción

A lo largo de este emocionante recorrido por la programación funcional en Agda, hemos construido una sólida base de conocimientos. Desde los tipos básicos y las funciones puras hasta el asombroso poder de los tipos dependientes y las pruebas formales, hemos desvelado los secretos de este fascinante paradigma.

Sin embargo, nuestro viaje no termina aquí. Agda es un lenguaje vasto y poderoso, con una profundidad y riqueza que constantemente desafía los límites de lo que es posible en el mundo de la programación. En este capítulo culminante, nos adentraremos en algunos de los temas más avanzados y cautivadores que Agda tiene para ofrecer.

Prepárate para sumergirte en un universo de abstracciones elegantes, técnicas sofisticadas y aplicaciones vanguardistas. Descubrirás cómo las mónadas y la teoría de categorías pueden elevar tu capacidad de

razonamiento sobre efectos y composición. Te maravillarás con las posibilidades que ofrecen las técnicas de parsing y compilación para crear tus propios lenguajes e intérpretes. Aprenderás a aprovechar la metaprogramación para automatizar tareas tediosas y expandir los horizontes expresivos de Agda. Y explorarás cómo la criptografía funcional puede ser tu aliada en la construcción de sistemas más seguros y verificables.

Estos temas representan la vanguardia del conocimiento en la programación funcional. Son las herramientas y técnicas que los expertos emplean para abordar los desafíos más complejos e interesantes en el desarrollo de software. Al dominar estos conceptos, no solo ampliarás tu comprensión de Agda, sino que también abrirás nuevas puertas de posibilidades en tu viaje como programador funcional.

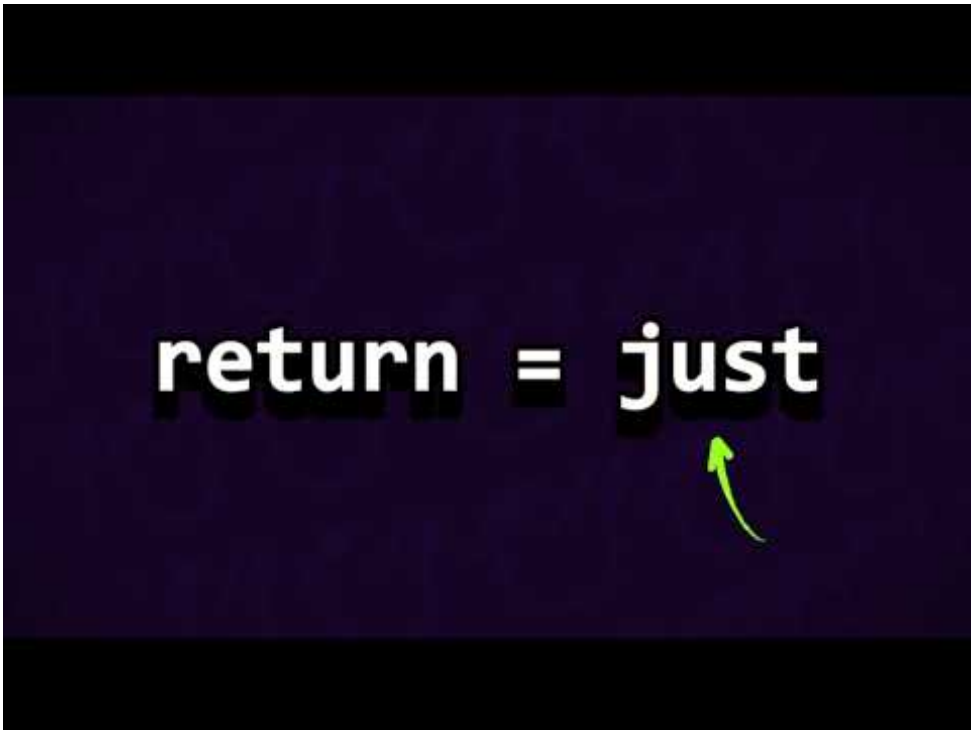
8.2. Abstracciones avanzadas

8.2.1. Mónadas

En capítulos anteriores, introdujimos el concepto de mónadas como una poderosa abstracción para estructurar y secuenciar computaciones con efectos. Vimos cómo las mónadas nos permiten encapsular operaciones impuras, como la entrada/salida o el estado mutable, de una manera segura y componible. Sin embargo, las mónadas son mucho más que una simple herramienta para manejar efectos; son un patrón fundamental de diseño que captura una amplia gama de estructuras y comportamientos computacionales.

Recordemos que una mónada se define formalmente como un tipo de datos `M` equipado con dos operaciones: `return` (también conocida como `pure`) y `_>>=_` (conocida como "bind"). Estas operaciones deben satisfacer ciertas leyes de coherencia conocidas como las leyes de mónadas.

En Agda, la interfaz de una mónada se puede representar utilizando un registro parametrizado:



Video 07: Definición de monada Maybe

```
record Monad (M : Set → Set) : Set₁ where
  field
    return : {A : Set} → A → M A
    _>>=_  : {A B : Set} → M A → (A → M B) → M B
    left-id  : {A B : Set} (a : A) (f : A → M B) →
               (return a >>= f) ≡ f a
    right-id : {A : Set} (ma : M A) →
               (ma >>= return) ≡ ma
    assoc    : {A B C : Set} (ma : M A) (f : A → M B) (g : B → M C) →
               ((ma >>= f) >>= g) ≡ (ma >>= (λ a → f a >>= g))
```

Aquí, `return` es una función que toma un valor de tipo `A` y lo envuelve en el contexto monádico `M`, mientras que `_>>=_` (pronunciado "bind") es una función que toma un valor monádico `M A` y una función `A → M B`, extrae el valor `A` del contexto, aplica la función y devuelve el resultado envuelto en `M B`.

Las leyes de mónadas están codificadas como los campos `left-id`, `right-id` y `assoc`. Estas leyes aseguran que las mónadas se comporten de manera coherente y componible. La ley de identidad izquierda establece que `return` es una identidad para `_>>=_` por la izquierda, la ley de identidad derecha establece que `return` es una identidad para `_>>=_` por la derecha, y la ley asociativa establece que `_>>=_` es asociativa.

Un ejemplo canónico de mónada es la mónada `Maybe`, que representa computaciones que pueden fallar:

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
instance
  MaybeMon : Monad Maybe
  MaybeMon = record
    { return = just
    ; _>>=_  = maybe′ (λ _ → nothing)
    }
    where
      maybe′ : {A B : Set} → (A → Maybe B) → Maybe A → Maybe B
      maybe′ f (just a)  = f a
      maybe′ f nothing = nothing
```

Aquí, `return` simplemente envuelve un valor en el constructor `just`, mientras que `_>>=_` desenvuelve un valor `Maybe A`, aplica la función si es `just`, o propaga `nothing` si la computación ya ha fallado.

La belleza de las mónadas radica en su capacidad para componer y secuenciar operaciones de manera clara y legible. Por ejemplo, podemos encadenar una serie de cómputos `Maybe` utilizando `_>>=_`:

```
validateLength : String → Maybe String
validateLength s =
  if length s >= 8 then just s else nothing
validateUppercase : String → Maybe String
validateUppercase s =
  if any isUppercase s then just s else nothing
validatePassword : String → Maybe String
validatePassword s =
  validateLength s >>= validateUppercase
```

Aquí, `validatePassword` primero verifica que la longitud de la contraseña sea de al menos 8 caracteres utilizando `validateLength`, y luego verifica que contenga al menos una letra mayúscula utilizando `validateUppercase`. El uso de `_>>=_` nos permite secuenciar estas operaciones de manera concisa, propagando automáticamente los fallos a través de la cadena de cómputos.

Esta capacidad de composición es lo que hace a las mónadas tan poderosas. Nos permiten construir programas complejos a partir de piezas más pequeñas y reutilizables, abstrayendo los detalles de manejo de errores, estado, no determinismo, etc. Algunos otros ejemplos comunes de mónadas incluyen:

- `List` para representar cómputos no deterministas o con múltiples resultados.
- `State` para pasar estado mutable a través de una secuencia de operaciones.
- `Reader` para proporcionar un entorno o contexto a un cómputo.
- `Writer` para generar un registro o traza junto con el resultado de un cómputo.
- `IO` para realizar operaciones de entrada/salida.

Una técnica poderosa en la programación con mónadas es el uso de transformadores de mónadas. Un transformador de mónada es un tipo que toma una mónada como parámetro y devuelve una nueva mónada que agrega alguna funcionalidad extra. Esto nos permite construir mónadas más complejas y especializadas componiendo transformadores.

En Agda, los transformadores de mónadas se pueden definir utilizando registros de tipo de orden superior. Por ejemplo, aquí se muestra la definición del transformador de estado `StateT`:

```
record StateT (S : Set) (M : Set → Set) (A : Set) : Set where
  field
```

```
    runStateT : S → M (A × S)
module _ {S : Set} {M : Set → Set} ⦃ MonM : Monad M ⦄ where
  open Monad MonM
  mapStateT : {A B : Set} → (M (A × S) → M (B × S)) → StateT S M A → StateT S M B
  mapStateT f ma = record { runStateT = λ s → f (runStateT ma s) }
  instance
    MonadStateT : Monad (StateT S M)
    MonadStateT = record
      { return = λ a → record { runStateT = λ s → return (a , s) }
      ; _>>=_  = λ ma f → record { runStateT = λ s →
          runStateT ma s >>= λ{ (a , s') → runStateT (f a) s' }
        }
      }
```

Aquí, `StateT S M A` representa un cómputo que puede leer y escribir un estado de tipo `S` mientras interactúa con la mónada base `M`. La función `runStateT` desenvuelve el cómputo, tomando un estado inicial y devolviendo un valor `M` que contiene tanto el resultado como el estado final.

La función `mapStateT` nos permite transformar la computación interna, mientras que la instancia `MonadStateT` proporciona las operaciones estándar de `return` y `_>>=_` para `StateT`. Luego, `StateT` se puede usar para agregar capacidades de estado a otras mónadas:

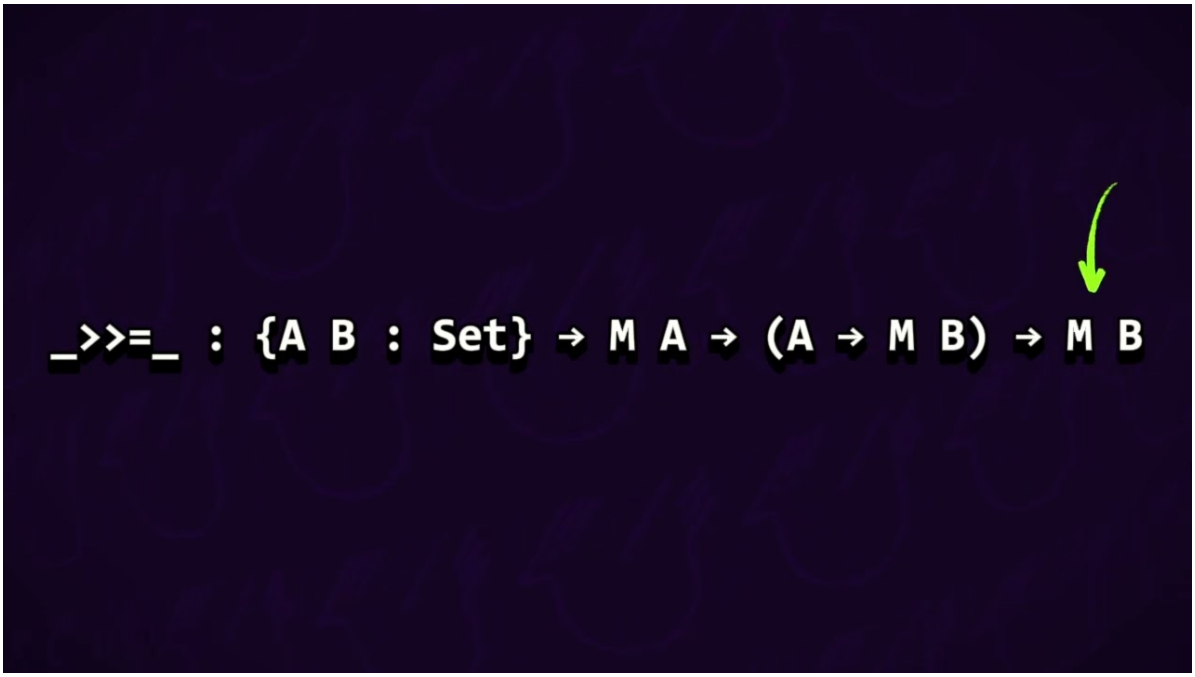
```
type Parser A = StateT String Maybe A
satisfy : (Char → Bool) → Parser Char
satisfy p = do
  s ← get
  case uncons s of λ where
    nothing → lift nothing
    (just (c , s')) → if p c then put s' >> return c
                              else lift nothing
  where
    get : Parser String
    get = record { runStateT = λ s → return (s , s) }
    put : String → Parser ⊤
    put s = record { runStateT = λ _ → return (tt , s) }
```

Aquí, definimos un parser simple utilizando `StateT` sobre la mónada `Maybe`. El parser `satisfy` intenta consumir y devolver el siguiente carácter de la entrada si cumple con el predicado dado `p`. Utiliza las

operaciones `get` y `put` para leer y escribir el estado de la cadena de entrada, y `lift nothing` para fallar si no se puede analizar ningún carácter.

Las mónadas y los transformadores de mónadas son herramientas esenciales en el toolkit del programador funcional. Nos permiten estructurar y razonar sobre efectos y cómputos complejos de una manera clara y modular. A medida que nuestra codificación en Agda se vuelve más sofisticada, dominar el uso de mónadas se vuelve cada vez más importante.

Sin embargo, las mónadas son solo la punta del iceberg cuando se trata de abstracciones expresivas en Agda. A continuación, exploraremos un tema aún más general y fundamental: la teoría de categorías.



Video 09: Definición del registro monada

8.2.2. Teoría de Categorías



La teoría de categorías es como un lenguaje universal para hablar sobre estructuras y relaciones en matemáticas, pero no te asustes, ¡también tiene aplicaciones prácticas en programación! Imagina que estás organizando tu colección de libros. Podrías agruparlos por género, autor, o año de publicación. Cada una de estas formas de organización es como una "categoría" diferente.

En el mundo de la programación, una categoría podría ser el conjunto de todos los tipos de datos y las funciones entre ellos. Por ejemplo, en la categoría de los tipos de Agda, los "objetos" serían tipos como `Nat`, `Bool`, o `List A`, y los "morfismos" serían funciones entre estos tipos.

Formalmente, una categoría consta de:

1. Objetos (como nuestros tipos)
2. Morfismos entre objetos (como nuestras funciones)
3. Una forma de componer morfismos (como componer funciones)
4. Un morfismo identidad para cada objeto (como la función identidad para cada tipo)

Veamos cómo podríamos representar esto en Agda:

```
1  record Category : Set₁ where
2    field
3      Obj : Set
4      _⇒_ : Obj → Obj → Set
5      id : ∀ {A} → A ⇒ A
6      _∘_ : ∀ {A B C} → (B ⇒ C) → (A ⇒ B) → (A ⇒ C)
```

Aquí, `Obj` representa nuestros objetos, `_⇒_` nuestros morfismos, `id` la identidad, y `_∘_` la composición.

Ahora, ¿por qué nos importa todo esto? Bueno, la teoría de categorías nos permite ver patrones comunes en diferentes áreas de las matemáticas y la programación. Por ejemplo, el concepto de "funtor" en teoría de categorías es la base de la idea de "Functor" en programación funcional.

Un funtor es como un traductor entre categorías. Imagina que tienes dos formas diferentes de organizar tus libros (dos categorías). Un funtor te permitiría traducir sistemáticamente una organización a la otra, preservando la estructura.

En Agda, podríamos representar un funtor así:

```
record Functor (C D : Category) : Set where
  field
    F₀ : Category.Obj C → Category.Obj D
    F₁ : ∀ {A B} → C [ A , B ] → D [ F₀ A , F₀ B ]
    Fid : ∀ {A} → F₁ (C.id {A}) ≡ D.id {F₀ A}
    F∘ : ∀ {A B X} (f : C [ B , X ]) (g : C [ A , B ]) →
         F₁ (f C.∘ g) ≡ F₁ f D.∘ F₁ g
```

Aquí, `F₀` mapea objetos, `F₁` mapea morfismos, y `Fid` y `F∘` aseguran que se preserven la identidad y la composición.

En programación, los funtores nos permiten aplicar una función a cada elemento de una estructura de datos, manteniendo la estructura. Por ejemplo, `List` es un funtor: podemos aplicar una función a cada elemento de una lista, obteniendo una nueva lista.

```
mapList : ∀ {A B} → (A → B) → List A → List B
mapList f []       = []
mapList f (x ∷ xs) = f x ∷ mapList f xs
```

La teoría de categorías nos proporciona un marco para entender y trabajar con estas estructuras abstractas. Nos ayuda a ver conexiones profundas entre diferentes áreas de las matemáticas y la programación, y nos proporciona un lenguaje para hablar sobre estas conexiones.

En Agda, estos conceptos no son solo teoría: nos permiten escribir código genérico y reutilizable, trabajando con estructuras abstractas de una manera que se puede aplicar a muchos tipos concretos diferentes. Por ejemplo, podríamos escribir funciones que trabajan con cualquier funtor, no solo con listas o árboles específicos.

8.3. Análisis y Compilación

8.3.1. Parsing

El parsing es un proceso fundamental en la implementación de muchas herramientas de software basadas en lenguajes, como compiladores, intérpretes y validadores. Consiste en analizar una secuencia de símbolos, como

un programa o un formato de datos, para determinar su estructura gramatical con respecto a una gramática formal dada.

En Agda, los parsers se pueden modelar como funciones que toman una lista de tokens y devuelven un resultado sintáctico junto con los tokens restantes. Para manejar el no determinismo y la posibilidad de fallar, se envuelve el resultado en un tipo de mónada apropiado, como `Maybe` o `List`:

```
Parser : Set → Set → Set
Parser Tok A = List Tok → Maybe (A × List Tok)
```

Una técnica común para definir parsers de gramáticas libres de contexto es utilizar combinadores. La idea es construir parsers complejos componiendo parsers más simples mediante operaciones como alternación (`_<|>_`), secuenciación (`_*>_` y `_<*_`), y mapeo (`_<$>_`):

```
_<|>_ : {A : Set} → Parser Tok A → Parser Tok A → Parser Tok A
p₁ <|> p₂ = λ toks → maybe (p₂ toks) just (p₁ toks)
_<$>_ : {A B : Set} → (A → B) → Parser Tok A → Parser Tok B
f <$> p = λ toks → maybe′ (map f) (p toks)
_<*_ : {A B : Set} → Parser Tok A → Parser Tok B → Parser Tok A
p₁ <* p₂ = (λ a _ → a) <$> p₁ <*> p₂
_*>_ : {A B : Set} → Parser Tok A → Parser Tok B → Parser Tok B
p₁ *> p₂ = (λ _ b → b) <$> p₁ <*> p₂
_<*>_ : {A B : Set} → Parser Tok (A → B) → Parser Tok A → Parser Tok B
pf <*> pa = uncurry _$_ <$> (pf <*> pa)
```

Estos combinadores permiten especificar la estructura gramatical de manera declarativa. Veamos un ejemplo de un parser simple para expresiones aritméticas:

```
data Expr : Set where
  lit : ℕ → Expr
  add : Expr → Expr → Expr
  sub : Expr → Expr → Expr
  mul : Expr → Expr → Expr
expr : Parser Tok Expr
expr = add <|> sub <|> mul <|> lit
lit : Parser Tok Expr
lit = lit <$> tok number
add : Parser Tok Expr
add = _+_ <$> expr <* tok "+" <*> expr
sub : Parser Tok Expr
sub = _-_ <$> expr <* tok "-" <*> expr
```

```
mul : Parser Tok Expr
mul = _*_ <$> expr <* tok "*" <*> expr
```

Aquí, `lit`, `add`, `sub` y `mul` son parsers para los diferentes casos de la gramática, que se combinan en el parser general `expr` mediante alternación. Los parsers individuales utilizan `tok` para coincidir con tokens específicos, y `<$>`, `<*>`, `<*` y `*>` para secuenciar y transformar los resultados del parsing.

El parsing es un excelente caso de estudio tanto para la programación funcional en general como para la programación con tipos dependientes en particular. Con Agda, no solo podemos definir parsers elegantes utilizando combinadores, sino que también podemos razonar formalmente sobre su correctitud y propiedades.

Por ejemplo, podríamos querer probar que el parser es completo (puede analizar todas las cadenas válidas) y que es inequívoco (no hay dos árboles de parsing diferentes para la misma cadena). Esto se puede lograr generalizando el tipo de parsing para incluir una prueba de que el parser se corresponde con la gramática:

```
Parser : Grammar Tok NT → NT → Set
Parser G A = ∀ {as bs} → A ∈ NT → as ++ bs ∈ G A → (as : List Tok) → Maybe (bs : List Tok)
```

Esto requiere que, para cada regla gramatical `A → as ++ bs`, el parser para `A` consuma los tokens `as` y deje los tokens `bs`. Al indexar el tipo de parser en la gramática `G` y el no terminal `A`, podemos hacer un seguimiento de la relación entre los parsers y las reglas gramaticales en el sistema de tipos, lo que abre posibilidades interesantes para pruebas automatizadas y generación de parsers.

Ventaja de Tipos Dependientes para Parsing	Explicación
Especificar Gramáticas	Los tipos pueden codificar directamente las reglas gramaticales
Verificar Parsers	Los parsers pueden probarse contra su especificación gramatical
Generar Parsers	Los parsers pueden derivarse automáticamente de una especificación gramatical
Parsers Eficientes	Los parsers generados pueden optimizarse based en conocimiento gramatical
Parsers Extensibles	Las gramáticas pueden extenderse y componerse modularmente

Además, con los tipos dependientes, podemos representar gramáticas más expresivas que las gramáticas libres de contexto, como gramáticas indexadas, gramáticas dependientes del contexto e incluso gramáticas de tipos completos. Esto nos permite escribir parsers verificados para una gama más amplia de lenguajes y formatos de datos.

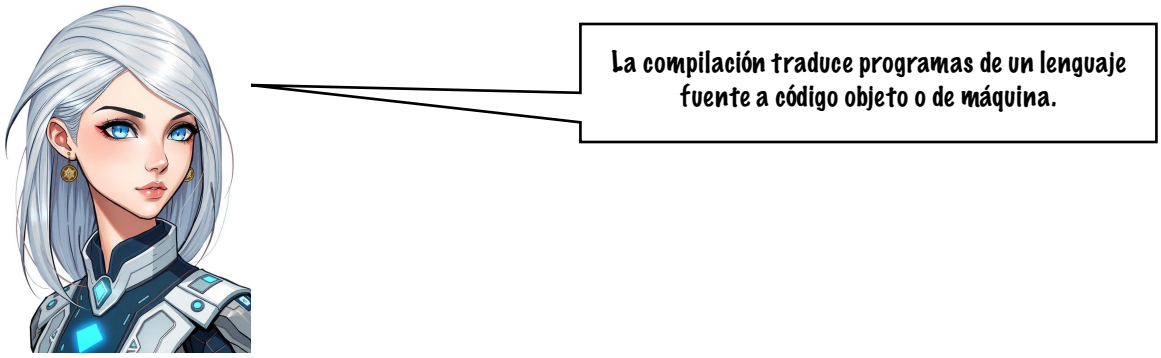
Otro aspecto importante del parsing que podemos abordar en Agda es la interacción con la semántica. Podemos definir una relación de evaluación para la sintaxis y probar que el parser respeta esta semántica de manera apropiada. Por ejemplo, podemos demostrar que el árbol sintáctico producido por el parser evalúa al mismo valor que la cadena de tokens original.

El parsing es solo una de las muchas fascinantes aplicaciones de la programación funcional dependientemente tipada en el dominio de los lenguajes de programación.

Aplicación	Explicación	Ventajas de los Tipos Dependientes
Compiladores	Transforman código fuente a código objeto o bytecode	Permite especificar y verificar la preservación de tipos, optimizaciones, y propiedades semánticas
Intérpretes	Ejecutan código fuente directamente sin compilación previa	Permite especificar y verificar la corrección de la evaluación con respecto a la semántica formal
Analizadores Estáticos	Analizan código fuente para encontrar errores, optimizaciones, o propiedades sin ejecutarlo	Permite expresar y comprobar análisis sensibles al flujo, al contexto, y a los tipos
DSLs Embebidos	Lenguajes específicos de dominio implementados dentro de un lenguaje anfitrión	Permite capturar invariantes y restricciones del dominio directamente en el sistema de tipos del host

Esto sienta las bases para explorar temas aún más emocionantes como la compilación, la interpretación y la verificación de lenguajes. Con la potencia de Agda, ¡el único límite es nuestra imaginación!

8.3.2. Compilación



La compilación es el proceso de traducir un programa de un lenguaje de programación (el lenguaje fuente) a otro lenguaje (el lenguaje objeto), generalmente con el propósito de ser ejecutado en hardware real o virtual. Los compiladores son algunos de los programas más complejos e importantes en ciencias de la computación, y desempeñan un papel fundamental en hacer que los lenguajes de alto nivel sean prácticos y utilizables.

Un compilador típico consta de varios pasos o fases:

Paso	Descripción
1.	Análisis léxico: División del programa fuente en una secuencia de tokens.
2.	Análisis sintáctico: Construcción de un árbol sintáctico abstracto (AST) a partir de los tokens.
3.	Análisis semántico: Comprobación de tipos, resolución de nombres, etc.
4.	Transformaciones de nivel intermedio: Optimizaciones, desugaring, etc.
5.	Generación de código: Traducción del AST a código objeto.

Cada una de estas fases es un dominio fascinante en sí mismo, que presenta muchos desafíos y oportunidades para la formalización y verificación.

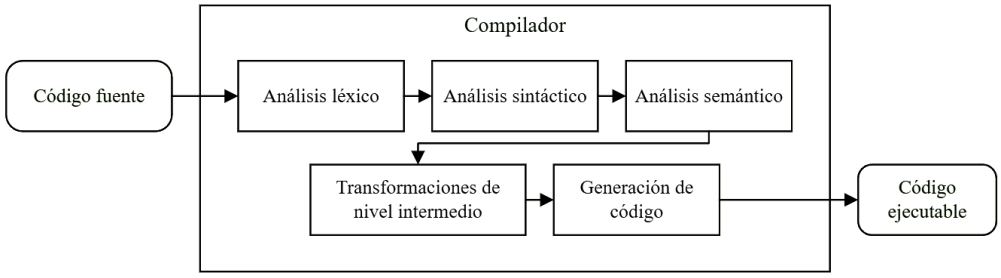


Imagen 22 - Flujo de Compilación: Del Código Fuente al Ejecutable.
(Fuente: Propia)

Con Agda, podemos representar los programas como términos de un tipo de datos inductivo y las transformaciones de programas como funciones sobre estos términos. Por ejemplo, podemos definir un pequeño lenguaje de expresiones aritméticas:

```
1  data Expr : Set where
2    num : ℕ → Expr
3    _+_ : Expr → Expr → Expr
4    _-_ : Expr → Expr → Expr
5    _*_ : Expr → Expr → Expr
```

Construcción	Explicación
num	Representa un número literal
_+_	Representa la suma de dos expresiones
_-_	Representa la resta de dos expresiones
_*_	Representa la multiplicación de dos expresiones
eval	Función de evaluación que reduce una Expr a un valor
compile	Función de compilación que traduce una Expr a una lista de instrucciones de máquina

Podemos definir una función de evaluación para este lenguaje de manera directa:

```
1  eval : Expr → ℕ
2  eval (num n)    = n
3  eval (e₁ + e₂)  = eval e₁ + eval e₂
4  eval (e₁ - e₂)  = eval e₁ - eval e₂
5  eval (e₁ * e₂)  = eval e₁ * eval e₂
```

Pero también podemos definir una función de compilación que traduce `Expr` a, por ejemplo, código de máquina para una máquina virtual de pila:

```
1  data Instr : Set where
2    PUSH : ℕ → Instr
```

```
  ADD  : Instr
  SUB  : Instr
  MUL  : Instr
compile : Expr → List Instr
compile (num n)    = [ PUSH n ]
compile (e₁ + e₂)  = compile e₁ ++ compile e₂ ++ [ ADD ]
compile (e₁ - e₂)  = compile e₁ ++ compile e₂ ++ [ SUB ]
compile (e₁ * e₂)  = compile e₁ ++ compile e₂ ++ [ MUL ]
```

Aquí, `compile` traduce una expresión a una secuencia de instrucciones de máquina que, cuando se ejecutan en una máquina de pila, calculan el valor de la expresión.

Lo realmente emocionante es que podemos probar formalmente que `compile` es correcta con respecto a `eval`. Es decir, podemos demostrar que ejecutar el código compilado siempre produce el mismo resultado que evaluar la expresión original:

```
correctness : (e : Expr) → exec (compile e) [] ≡ [ eval e ]
correctness (num n)   = refl
correctness (e₁ + e₂) =
  begin
    exec (compile e₁ ++ compile e₂ ++ [ ADD ]) []
  ≡⟨ cong (λ s → exec (compile e₁ ++ s) []) (correctness e₂) ⟩
    exec (compile e₁ ++ [ PUSH (eval e₂) , ADD ]) []
  ≡⟨ correctness e₁ ⟩
    exec [ ADD ] (eval e₂ :: eval e₁ :: [])
  ≡⟨⟩
    eval e₁ + eval e₂ :: []
  ∎
correctness (e₁ - e₂) = ...
correctness (e₁ * e₂) = ...
```

En este caso, `exec` es un intérprete para el código de máquina, y la prueba procede por inducción en la estructura de la expresión `e`, utilizando las propiedades de `exec` y algunas manipulaciones algebraicas simples.

Este es solo un ejemplo ilustrativo, pero demuestra el poder de Agda para razonar sobre compiladores. Con tipos dependientes, podemos especificar y verificar propiedades mucho más ricas, como:

- Preservación de tipos: si `e : τ`, entonces `compile y` produce código que también tiene tipo `τ`.
- Seguridad de la memoria: `compile y` nunca accede a memoria fuera de sus límites.

- Optimización correcta: una optimización `optimize` no cambia el comportamiento del programa (`exec (optimize (compile e)) ≡ exec (compile e)`).

Lo hermoso es que estas no son solo especificaciones en papel, sino afirmaciones reales que pueden ser formalmente verificadas por el verificador de tipos de Agda.

Además, utilizando técnicas como tipos de orden superior, podemos definir lenguajes y compiladores parametrizados. Esto nos permite reutilizar la lógica común entre diferentes lenguajes y targets, y derivar automáticamente compiladores para todo un espacio de posibles lenguajes simplemente instanciando estos parámetros.

Algunos ejemplos impresionantes del mundo real de la aplicación de Agda (y sus enfoques hermanos como Coq e Idris) a la compilación incluyen:

- Verificación completa de un compilador realista para un subconjunto de C en la tesis doctoral de Jeremey Koenig
- El compilador verificado CertiCoq para un subconjunto de Coq
- El compilador verificado CakeML para un subconjunto de ML
- El lenguaje de programación Plutus, utilizado para contratos inteligentes en la blockchain de Cardano

Si bien la verificación completa de compiladores realistas de la industria como GCC o LLVM puede seguir siendo un objetivo a largo plazo, la tecnología para la verificación a gran escala de compiladores está avanzando rápidamente, con Agda a la vanguardia. Con su combinación única de expresividad, automatización y base matemática sólida, Agda es una herramienta inmensamente poderosa para abordar el desafío de construir compiladores realmente correctos y dignos de confianza.

Ya sea que nuestro interés esté en la optimización, el targetting de máquina, la seguridad o cualquiera de los otros innumerables aspectos fascinantes de la compilación, Agda nos proporciona un marco para explorar estas ideas de una manera rigurosa y basada en principios. Es una invitación a sumergirse, experimentar y descubrir el emocionante mundo de la compilación verificada.

8.4. Metaprogramación

La metaprogramación es como tener un asistente programador que escribe código por ti. Imagina que pudieras decirle a tu computadora "Escribe un programa que haga X" y ella lo hiciera automáticamente. Eso, en esencia, es la metaprogramación.

En términos más técnicos, la metaprogramación es la práctica de escribir programas que generan, manipulan o analizan otros programas. Es como si nuestro código tuviera superpoderes y pudiera crear más código.

En Agda, la metaprogramación nos permite hacer cosas asombrosas. Podemos automatizar tareas repetitivas, generar código basado en tipos, e incluso crear nuestras propias extensiones del lenguaje. Todo esto mientras mantenemos la seguridad y las garantías que nos da el sistema de tipos de Agda.

Empecemos con un ejemplo simple. Supongamos que queremos una función que tome un número n y genere una lista de n elementos, cada uno siendo ese mismo número n. Podríamos escribirlo manualmente:

```
replicate : {A : Set} → A → (n : ℕ) → Vec A n
replicate x zero = []
replicate x (suc n) = x ∷ replicate x n
```

Pero ¿y si pudiéramos generar esta función automáticamente para cualquier tipo? Aquí es donde entra la metaprogramación:

```
genReplicate : (A : Set) → Term
genReplicate A = def (quote replicate)
  [ arg (argInfo visible relevant) (quoteTerm A)
  , arg (argInfo visible relevant) (var 0 [])
  , arg (argInfo visible relevant) (var 1 [])
  ]
macro
  mkReplicate : (A : Set) → Term → TC ⊤
  mkReplicate A goal = do
    let body = genReplicate A
    ty ← quoteTC ((a : A) → (n : ℕ) → Vec A n)
    defineFun (quote replicate) [ clause [] ty body ]
    unify goal (quoteTerm replicate)
```

Este código parece complejo, pero lo que hace es generar automáticamente la función `replicate` que escribimos manualmente antes. La macro `mkReplicate` toma un tipo `A` y genera la función `replicate` para ese tipo.

La metaprogramación en Agda se basa en la capacidad de manipular términos del lenguaje como datos. El tipo `Term` representa la sintaxis de Agda, y funciones como `quoteTC` y `unquote` nos permiten convertir entre código Agda normal y su representación como `Term`.

Otro ejemplo común de metaprogramación es la generación automática de instancias para tipos de datos. Por ejemplo, podríamos generar automáticamente funciones de igualdad para nuestros tipos de datos:

```
macro
  deriveEq : Name → Term → TC ⊤
  deriveEq name hole = do
    type ← getType name
    cons ← getConstructors name
    -- Aquí iría el código para generar la función de igualdad
    -- basada en los constructores del tipo
    -- ...
    unify hole generatedEqFunction
data Tree (A : Set) : Set where
```

```
  leaf : A → Tree A
  node : Tree A → Tree A → Tree A
deriveEq Tree
```

Este código generaría automáticamente una función de igualdad para el tipo `Tree`, sin que tengamos que escribirla manualmente.

La metaprogramación es una herramienta poderosa, pero también puede ser compleja. Es importante usarla con cuidado y asegurarse de que el código generado sea correcto y eficiente. Sin embargo, cuando se usa adecuadamente, la metaprogramación puede hacer que nuestro código sea más conciso, más fácil de mantener y menos propenso a errores.

En Agda, la metaprogramación se integra de manera segura con el sistema de tipos. Esto significa que podemos usar toda la potencia de los tipos dependientes de Agda para asegurarnos de que nuestro código generado sea correcto por construcción.

8.4.1. Macros y templating

La metaprogramación es una técnica poderosa que nos permite escribir programas que manipulan otros programas, incluyendo a sí mismos. Su objetivo es ampliar y automatizar el proceso de programación, facilitando la creación de nuevas abstracciones, la reducción de código repetitivo y la adaptación de los lenguajes a necesidades específicas.

Uno de los principales mecanismos de metaprogramación es el sistema de macros. Las macros son esencialmente funciones que operan sobre la sintaxis del programa en lugar de los valores en tiempo de ejecución. Se ejecutan durante la compilación, tomando fragmentos de código como entrada y generando código transformado como salida, que luego se compila de manera regular.

Agda soporta un potente sistema de macros basado en la reflexión, el cual nos permite inspeccionar la propia estructura de nuestros programas en tiempo de ejecución. Esto se logra a través de un universo de tipos `Term` que representa la sintaxis de Agda, junto con una serie de primitivas para manipular estos términos:

```
data Term : Set where
  var : (x : Nat) → List (Arg Term) → Term
  con : (c : Name) → List (Arg Term) → Term
  def : (f : Name) → List (Arg Term) → Term
  lam : (v : Visibility) (t : Abs Term) → Term
  pat-lam  : (cs : List Clause) (args : List (Arg Term)) → Term
  pi : (a : Arg Type) (b : Abs Type) → Term
  sort : (s : Sort) → Term
  lit : (l : Literal) → Term
  meta : (x : Meta) → List (Arg Term) → Term
  unknown : Term
```

Cada constructor representa un nodo diferente en el AST (Abstract Syntax Tree) de Agda. Por ejemplo, `var` representa variables, `con` representa constructores de datos, `def` representa funciones definidas, entre otros.

Con el tipo `Term`, podemos escribir funciones que generen o transformen código Agda. Sin embargo, para utilizar esto en metaprogramación, necesitamos una forma de ejecutar código Agda en tiempo de compilación y enlazar los términos resultantes de vuelta en el programa. Esto se logra mediante el tipo `TC`:

```
postulate
  TC : ∀ {a} → Set a → Set a
  unquote : ∀ {A : Set} → Term → TC A
  quoteTerm : ∀ {A : Set} → A → Term
```

`TC A` representa una computación que produce un valor de tipo `A` en tiempo de compilación. La función `unquote` toma un término y lo ejecuta como un programa Agda, mientras que `quoteTerm` realiza la operación inversa, convirtiendo un valor de Agda en su representación como término.

Utilizando estas primitivas, podemos definir macros como funciones que toman términos, los manipulan y devuelven nuevos términos. Por ejemplo, la siguiente macro calcula una expresión aritmética en tiempo de compilación:

```
macro
  eval : Term → Term → TC _
  eval t hole = do
    t ← unquote t
    case t of λ where
      (num n) → unify hole (lit (nat n))
      (t₁ + t₂) → do
        v₁ ← eval t₁ unknown
        v₂ ← eval t₂ unknown
        case v₁ , v₂ of λ where
          (lit (nat n₁)) , (lit (nat n₂)) → unify hole (lit (nat (n₁ + n₂)))
          _ , _ → typeError (strErr "Expected naturals" :: [])
      _ → typeError (strErr "Expected natural or addition" :: [])
```

Esta macro recibe un término `t` que representa una expresión aritmética y un término `hole` que indica dónde insertar el resultado. Descompone `t` de forma recursiva, evalúa cada subexpresión y unifica el resultado final con `hole`.

Aspecto	Tipos Nominales	Tipos Estructurales
Definición	Se definen explícitamente usando la palabra clave `data` y dándoles un nombre.	Se definen implícitamente por su estructura, sin darles un nombre explícito.

Igualdad	Dos valores son iguales si tienen el mismo constructor y sus argumentos son iguales.	Dos valores son iguales si tienen la misma estructura, independientemente de cómo fueron construidos.
Compatibilidad	Los tipos nominales no son compatibles entre sí, incluso si tienen la misma estructura.	Los tipos estructurales son compatibles si tienen la misma estructura, incluso si fueron definidos en diferentes lugares.

Podemos utilizarla de la siguiente manera:

```
1  _ : eval (1 + 2 + 3) ≡ 6
2  _ = refl
```

Durante la compilación, la llamada a `eval` será reemplazada por el valor `6`, que luego se verificará contra el tipo `eval (1 + 2 + 3) ≡ 6`.

Las macros son esencialmente funciones parciales, por lo que pueden fallar o no terminar. Agda maneja esto permitiendo que las macros generen errores de tipo o no terminen. Sin embargo, si una macro se ejecuta exitosamente, el término resultante se verifica completamente, garantizando que las macros no comprometan la seguridad de tipos.

Una aplicación clave de las macros es la generación automática de código. Por ejemplo, podemos definir una macro para generar funciones de igualdad estructural para tipos de datos:

```
1  macro
2    deriveEq : Name → TC _
3    deriveEq n = do
4      qn ← quoteTerm n
5      declareEq qn
6      d ← getDefinition n
7      let as = dataParameters d
8      forEachConstructor d λ c → do
9        qc ← quoteTerm c
10       declareCEq qn qc as
11   where
12     declareEq : Term → TC _
13     declareEq qn = do
14       defineFun (qn ++ ":Eq") (clause [] (sort (lit nat)))
15     declareCEq : Term → Term → List (Arg Term) → TC _
16     declareCEq qn qc as = defineFun (qn ++ ":Eq:" ++ qc) (clause as unknown)
```

Esta macro toma el nombre de un tipo de datos `n`, genera una función de igualdad `n:Eq` y luego genera una función de igualdad para cada constructor `c` del tipo `n:Eq:c`. Podemos utilizarla de la siguiente manera:

```
1  data Tree : Set where
2    leaf : Tree
```

```
  node : Tree → ℕ → Tree → Tree
deriveEq Tree
```

Esto generará automáticamente las siguientes definiciones:

```
Tree:Eq : Tree → Tree → Bool
Tree:Eq:leaf : Tree.leaf ≡ Tree.leaf
Tree:Eq:node : (a b : Tree) → (c : ℕ) → (d e : Tree) → Tree.node a c d ≡ Tree.node b c e
Tree:Eq : Tree → Tree → Bool
```

Incluso podemos combinar esto con pruebas para derivar automáticamente propiedades de estas funciones, como reflexividad, simetría y transitividad.

Propiedad	Enunciado Formal	Explicación
Reflexividad	∀ x. x ≡ x	Todo valor es igual a sí mismo.
Simetría	∀ x y. x ≡ y → y ≡ x	Si x es igual a y, entonces y es igual a x.
Transitividad	∀ x y z. x ≡ y → y ≡ z → x ≡ z	Si x es igual a y, y y es igual a z, entonces x es igual a z.
Congruencia	∀ f x y. x ≡ y → f x ≡ f y	Funciones aplicadas a argumentos iguales producen resultados iguales.
Sustituibilidad	∀ P x y. x ≡ y → P x → P y	Si x es igual a y, entonces cualquier propiedad P que vale para x también vale para y.

Otra técnica relacionada con las macros es el templating. Los templates son esencialmente funciones Agda parametrizadas sobre tipos o términos, que pueden ser instanciadas con argumentos específicos para generar código. A diferencia de las macros, los templates son completamente verificados y no pueden fallar o divergir.

Por ejemplo, podemos definir un template para pares con una función de proyección:

```
record Pair (A B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B
open Pair
```

Luego, este template se puede instanciar con diferentes tipos para generar pares concretos:

```
IntPair : Set
IntPair = Pair ℕ ℕ
StringPair : Set
StringPair = Pair String String
```

También es posible parametrizar sobre términos:

```
add : ℕ → ℕ → ℕ
add = _+_
mul : ℕ → ℕ → ℕ
mul = _*_
BinOp : Set
BinOp = Pair (ℕ → ℕ → ℕ) String
plus : BinOp
plus = add , "+"
times : BinOp
times = mul , "*"
```

Los templates nos proporcionan una forma poderosa de reutilización y abstracción en Agda, permitiéndonos factorizar patrones comunes y generar instancias específicas según sea necesario. Esto reduce la duplicación de código y mejora la mantenibilidad.

En combinación con tipos dependientes, los templates pueden expresar abstracciones muy generales y ricas en propiedades. Por ejemplo, podemos definir un template para estructuras algebraicas:

```
record Monoid (A : Set) : Set where
  field
    mempty : A
    _<>_ : A → A → A
    assoc : (a b c : A) → ((a <> b) <> c) ≡ (a <> (b <> c))
    identityˡ : (a : A) → (mempty <> a) ≡ a
    identityʳ : (a : A) → (a <> mempty) ≡ a
open Monoid {{...}}
```

Estructura Algebraica	Signatura	Leyes
Semigrupo	$_\cdot_ : A \to A \to A$	Asociatividad: $(x \cdot y) \cdot z \equiv x \cdot (y \cdot z)$
Monoide	$mempty : A _\cdot_ : A \to A \to A$	Identidad a izquierda: $mempty \cdot x \equiv x$Identidad a derecha: $x \cdot mempty \equiv x$Asociatividad: $(x \cdot y) \cdot z \equiv x \cdot (y \cdot z)$
Grupo	$mempty : A _\cdot_ : A \to A \to A _^{-1} : A \to A$	Identidad a izquierda: $mempty \cdot x \equiv x$Identidad a derecha: $x \cdot mempty \equiv x$Asociatividad: $(x \cdot y) \cdot z \equiv x \cdot (y \cdot z)$Inversos: $x \cdot x^{-1} \equiv mempty$ y $x^{-1} \cdot x \equiv mempty$

Luego, este template se puede instanciar para tipos específicos:

```
instance
  NatAddMonoid : Monoid ℕ
  NatAddMonoid = record
```

```
    { mempty = 0
    ; _<>_ = _+_
    ; assoc = +-assoc
    ; identityˡ = +-identityˡ
    ; identityʳ = +-identityʳ
    }
```

La definición de `Monoid` actúa como una especificación de las propiedades que deben cumplir sus instancias. Agda verificará que las pruebas proporcionadas para `NatAddMonoid` efectivamente cumplan con las leyes monoidales. Esto nos permite una forma muy expresiva y segura de programación genérica.

8.4.2. Generación de Código

Una de las aplicaciones más emocionantes de la metaprogramación es la generación automática de código. En lugar de escribir manualmente código repetitivo o propenso a errores, podemos crear programas que generen código por nosotros, basados en especificaciones de alto nivel o fuentes de datos. Esto no solo nos ahorra tiempo y esfuerzo, sino que también mejora significativamente la corrección y mantenibilidad del código resultante.

Agda es excepcionalmente adecuado para la generación de código gracias a su potente sistema de tipos y capacidades de metaprogramación. Con Agda, podemos escribir generadores garantizados para producir código bien tipado, e incluso codificar y verificar las propiedades deseadas del código generado en los tipos de los propios generadores.

Por ejemplo, consideremos la tarea de serializar y deserializar estructuras de datos desde o hacia formatos como JSON o XML. Escribir manualmente el código para convertir entre representaciones internas y externas puede resultar tedioso, propenso a errores y difícil de mantener, especialmente cuando las estructuras de datos evolucionan con el tiempo.

En su lugar, podríamos definir un lenguaje de dominio específico (Lenguaje de programación especializado para un dominio particular) integrado (EDSL) en Agda para especificar el formato de los datos y las reglas de conversión. Luego, escribiríamos un generador que traduzca automáticamente estas especificaciones a código Agda. Veamos un ejemplo simple:

```
data JSONSchema : Set where
  str   : JSONSchema
  num   : JSONSchema
  ✓_    : String → JSONSchema → JSONSchema
  _⊗_   : JSONSchema → JSONSchema → JSONSchema
infixr 4 ✓_
infixr 6 _⊗_
⟦_⟧ : JSONSchema → Set
⟦ str ⟧       = String
```

```
⟦ num ⟧       = Float
⟦ ✓_ s A ⟧    = ⟦ A ⟧
⟦ A ⊗ B ⟧     = ⟦ A ⟧ × ⟦ B ⟧
data JSON : Set where
  str_   : String → JSON
  num_   : Float → JSON
  obj_   : List (String × JSON) → JSON
record ToJSON (A : Set) : Set where
  field
    toJSON : A → JSON
record FromJSON (A : Set) : Set where
  field
    fromJSON : JSON → Maybe A
```

Aquí, `JSONSchema` es un EDSL para especificar formatos JSON. Los constructores `str` y `num` representan cadenas y números primitivos, `✓_` representa campos etiquetados y `_⊗_` representa objetos producto. La función `⟦_⟧` traduce un `JSONSchema` al tipo Agda correspondiente.

`JSON` es un tipo de datos simple para representar valores JSON, mientras que `ToJSON` y `FromJSON` son interfaces para convertir entre tipos Agda y `JSON`.

Con estas definiciones, podemos escribir un generador que tome un `JSONSchema` y produzca instancias de `ToJSON` y `FromJSON` para el tipo Agda correspondiente:

```
genToJSON : (A : JSONSchema) → ToJSON ⟦ A ⟧
genToJSON str         = record { toJSON = str_ }
genToJSON num         = record { toJSON = num_ }
genToJSON (✓_ s A)    = record { toJSON = λ a → str_ (toJSON (genToJSON A) a) }
genToJSON (A ⊗ B)     = record { toJSON = λ (a , b) → obj_ (("fst" , toJSON (genToJSON A) a) ∷ ("snd" , toJSON (genToJSON B) b) ∷ []) }
genFromJSON : (A : JSONSchema) → FromJSON ⟦ A ⟧
genFromJSON str        = record { fromJSON = λ{ (str_ s) → just s ; _ → nothing } }
genFromJSON num        = record { fromJSON = λ{ (num_ x) → just x ; _ → nothing } }
genFromJSON (✓_ s A)   = record { fromJSON = λ{ (str_ s) → fromJSON (genFromJSON A) (obj_ (s , str_ s) ∷ []) ; _ → nothing } }
genFromJSON (A ⊗ B)    = record { fromJSON = λ{ (obj_ (((_ , a) ∷ (_ , b) ∷ []))) → _,_ <$> fromJSON (genFromJSON A) a ⊛ fromJSON (genFromJSON B) b ; _ → nothing } }
```

Las funciones `genToJSON` y `genFromJSON` recorren recursivamente la estructura del `JSONSchema` y generan las implementaciones correspondientes de `toJSON` y `fromJSON`. Para tipos producto, generan código que serializa o deserializa cada componente de forma recursiva y luego combina los resultados.

Ahora podemos definir esquemas JSON para nuestros tipos de datos y obtener automáticamente la serialización y deserialización:

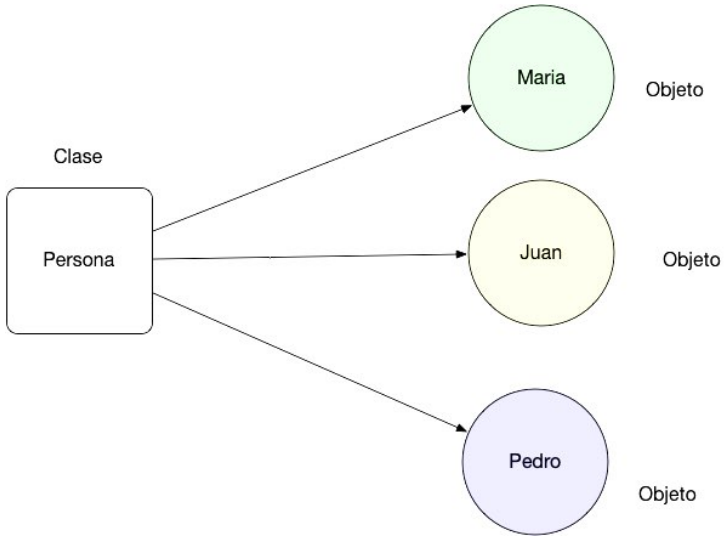


Imagen 23 - Tipos Singleton: Representando Valores a Nivel de Tipo.
(Fuente: Cecilio Álvarez Caules)

```
1  UserSchema : JSONSchema
2  UserSchema = ✓ "name" str ⊗ ✓ "age" num
3  data User : Set where
4    user : String → Float → User
5  userToJSON : ToJSON User
6  userToJSON = genToJSON UserSchema
7  userFromJSON : FromJSON User
8  userFromJSON = genFromJSON UserSchema
```

Este enfoque presenta varias ventajas sobre la codificación manual de la serialización:

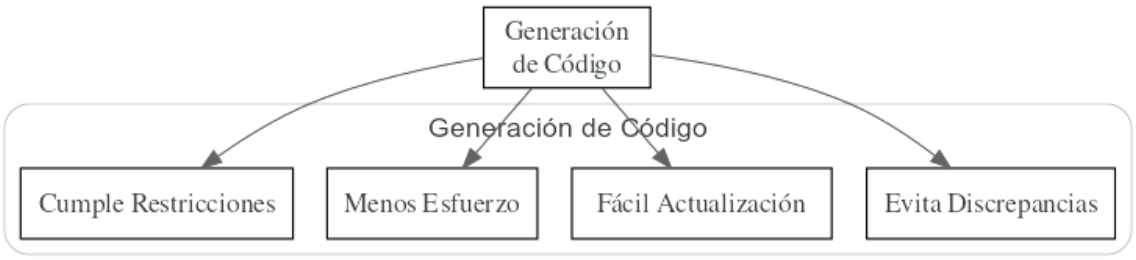


Imagen 24 – Ventajas de la Generación de Código.
(Fuente: Propia)

Punto	Descripción
El formato de los datos solo necesita especificarse una vez en el JSONSchema. El código real para toJSON y fromJSON se genera automáticamente.	Esto elimina la posibilidad de discrepancias, ya que el código se genera automáticamente a partir del esquema, evitando errores de inconsistencia.

Si el formato de los datos cambia (por ejemplo, se agrega o elimina un campo), solo se requiere actualizar el JSONSchema.	El código de serialización se adaptará automáticamente a los cambios en el esquema, lo que simplifica el mantenimiento y reduce el riesgo de errores.
Obtener toJSON y fromJSON se reduce a una simple aplicación de función. No existe la posibilidad de olvidar implementarlos o que se desincronicen.	Esto garantiza consistencia y evita problemas comunes de desincronización entre la implementación y el esquema de datos.
Podemos codificar restricciones o invariantes en el JSONSchema, y el sistema de tipos de Agda garantizará que el código generado las respete.	Al utilizar el sistema de tipos de Agda, podemos garantizar que el código generado cumpla con las restricciones definidas en el esquema.

Si bien este es un ejemplo relativamente simple, el mismo principio se puede extender a esquemas y formatos mucho más complejos. Podemos tener generadores no solo para tipos de datos, sino también para funciones, pruebas, benchmarks, documentación, etc. Incluso es posible crear generadores que tomen otros generadores como argumentos o que estén parametrizados sobre pruebas de ciertas propiedades.

Un área particularmente interesante es la generación de código específico de dominio. En lugar de escribir software de propósito general, podemos crear lenguajes y generadores personalizados para dominios específicos como finanzas, gráficos, aprendizaje automático, etc. Los expertos del dominio pueden entonces especificar el comportamiento deseado en un lenguaje de alto nivel y hacer que la implementación de bajo nivel se genere automáticamente.

Otros ejemplos interesantes de generación de código en Agda incluyen:

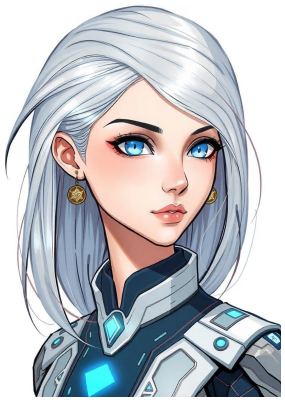
- Derivación automática de instancias para clases de tipos como `Eq`, `Ord`, `Show`, `Generic`, etc.
- Generación de código de bajo nivel (por ejemplo, C o ensamblador) a partir de especificaciones de alto nivel, con pruebas de corrección.
- Generación de API y bindings de lenguajes extranjeros a partir de especificaciones de tipos.
- Generación de arte ASCII, música, gráficos 3D, etc., a partir de descripciones de alto nivel.

Las posibilidades son realmente infinitas. Al aprovechar el poder de Agda para la metaprogramación, podemos automatizar cada vez más aspectos del proceso de desarrollo de software, desde la codificación de bajo nivel hasta la arquitectura de alto nivel. Esto nos permite crear sistemas más concisos, robustos, flexibles y verificables, al mismo tiempo que elevamos el nivel de abstracción en el que trabajamos.

En muchos sentidos, Agda representa el pináculo del sueño de la generación de código. Proporciona una base sólida y principios para la metaprogramación segura de tipos sin sacrificar la expresividad o la capacidad de prueba. Si podemos imaginar una transformación de código, es muy probable que podamos expresarla en Agda de una manera bien tipada.

Por lo tanto, la próxima vez que nos encontremos escribiendo código repetitivo o propenso a errores, detengámonos y preguntémonos: ¿cómo puedo hacer que Agda genere esto por mí? Las respuestas pueden sorprendernos y cambiar fundamentalmente la forma en que abordamos la resolución de problemas.

8.5. Criptografía



La criptografía funcional busca implementar primitivas y protocolos criptográficos de forma segura y verificable.

En el mundo moderno, la criptografía juega un papel fundamental en la protección de información sensible y en habilitar una amplia gama de servicios digitales. Desde la seguridad de las comunicaciones hasta la salvaguarda de datos financieros, pasando por el comercio electrónico y la privacidad individual, la criptografía es el pegamento invisible que mantiene unida gran parte de nuestra infraestructura digital.

Sin embargo, implementar algoritmos criptográficos correctamente es notoriamente difícil. Incluso pequeños errores pueden tener consecuencias catastróficas, como lo demuestran numerosos ejemplos de esquemas aparentemente seguros que luego resultaron fatalmente defectuosos. Aquí es donde las técnicas de la programación funcional, especialmente aquellas con tipos dependientes como Agda, pueden marcar una diferencia real.

La naturaleza de los lenguajes funcionales, con su énfasis en la inmutabilidad, la composición pura y el razonamiento ecuacional, los hace excepcionalmente adecuados para el dominio de la criptografía. Agda, en particular, permite no solo implementar algoritmos criptográficos, sino también especificar y verificar formalmente propiedades de seguridad directamente en los tipos.

Para ilustrar esto, consideremos el cifrado autenticado con datos asociados (AEAD), un esquema que proporciona confidencialidad, integridad y autenticidad.

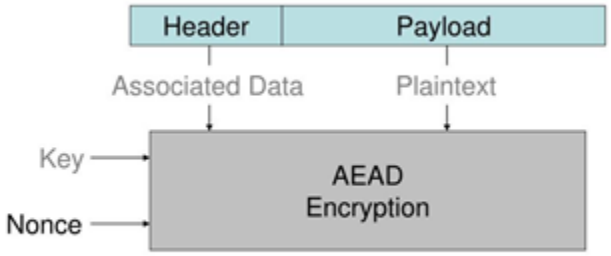
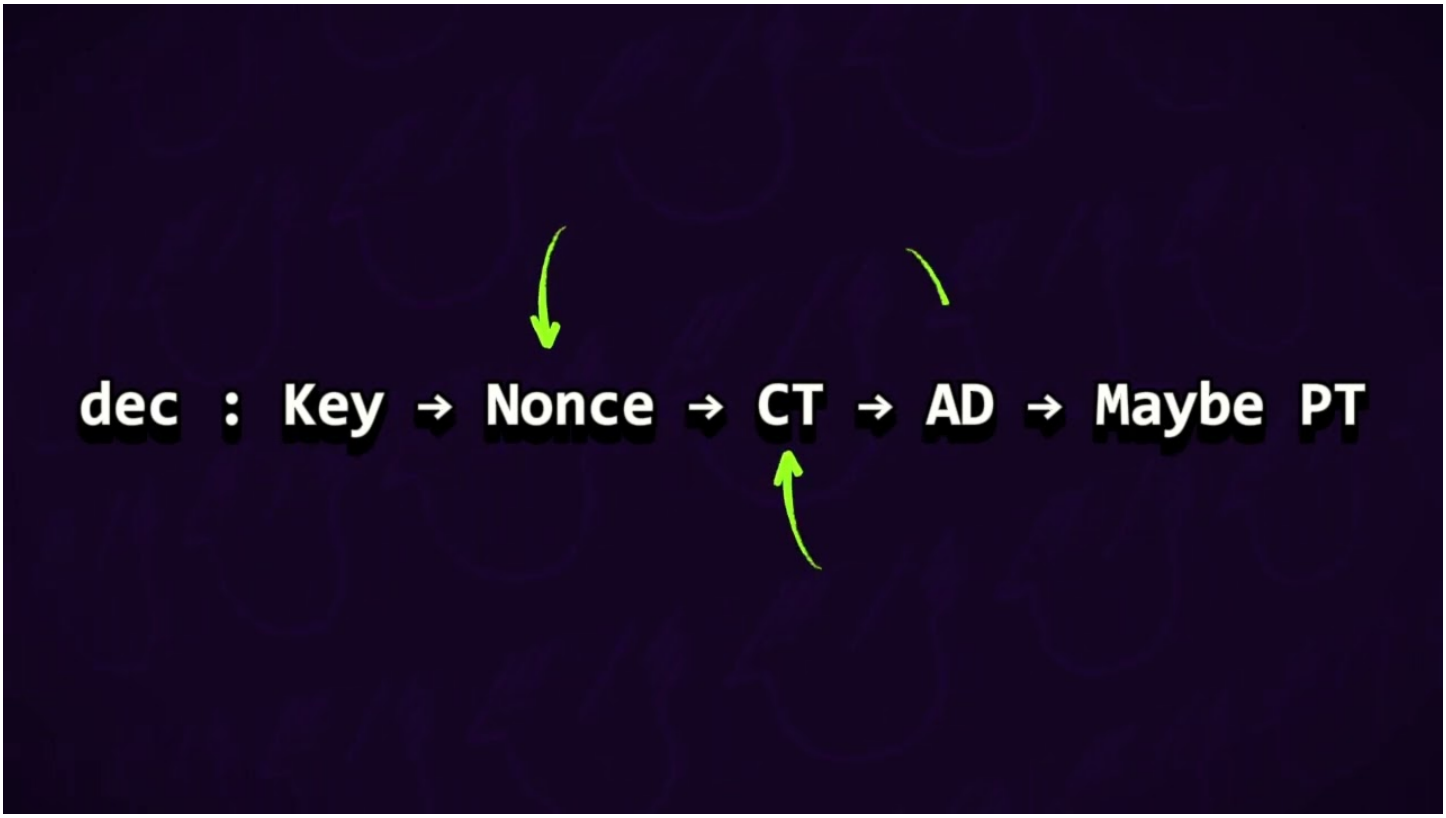


Imagen 25 - Esquema de Cifrado Autenticado con Datos Asociados (AEAD).
(Fuente: Nimantha Fernando)

En Agda, podemos especificar la funcionalidad de AEAD de la siguiente manera:



Video 08: Registro Read simplifica

```
1  record AEAD (Key : Set) (Nonce : Set) (PT : Set) (AD : Set) (CT : Set) : Set₁ where
2    field
3      enc : Key → Nonce → PT → AD → CT
4      dec : Key → Nonce → CT → AD → Maybe PT
5      .correctness : ∀ {k n pt ad} → dec k n (enc k n pt ad) ad ≡ just pt
6      .cpa-security : {- La propiedad de indistinguibilidad bajo ataque de texto plano escogido -}
7      .int-ctxt : {- La propiedad de integridad del texto cifrado -}
```

Aquí, cada componente tiene un significado preciso:

Componente	Explicación
Key	Tipo que representa las claves criptográficas
Nonce	Tipo que representa valores únicos usados para asegurar frescura y prevenir ataques de repetición
PT	Tipo que representa el texto plano (datos sin cifrar)
AD	Tipo que representa datos asociados (información no confidencial que se autentica pero no se cifra)
CT	Tipo que representa el texto cifrado (datos cifrados)

enc	Función de cifrado que toma una clave, un nonce, texto plano y datos asociados y produce un texto cifrado
dec	Función de descifrado que toma una clave, un nonce, texto cifrado y datos asociados y produce un Maybe texto plano
correctness	Prueba de que descifrar con los mismos parámetros que se usaron para cifrar da el texto plano original
cpa-security	Postulado que representa la seguridad bajo ataques de texto plano escogido (confidencialidad)
int-ctxt	Postulado que representa la integridad del texto cifrado (autenticidad)

Esta especificación captura la esencia de AEAD de una manera clara y concisa. Las funciones `enc` y `dec` representan las operaciones centrales, mientras que los campos `correctness`, `cpa-security` e `int-ctxt` codifican las propiedades de seguridad deseadas.

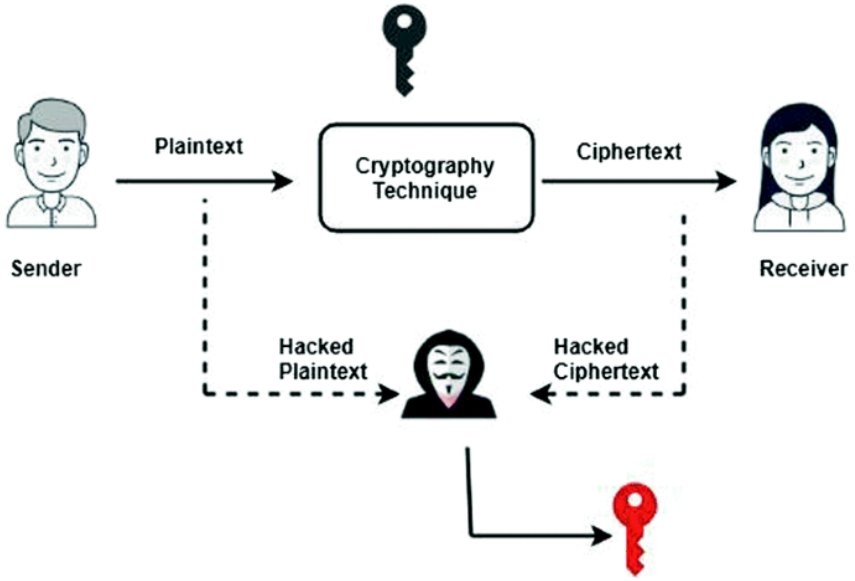


Imagen 26 - Ataque de Texto Plano Escogido en Criptografía.
(Fuente: B. Bazeer Ahamed)

Podemos entonces definir instancias concretas de AEAD implementando estos campos. Por ejemplo, he aquí una implementación simplificada basada en el cifrado de flujo XOR y el código de autenticación de mensajes GHASH:

```
1  module AESGCM where
2    open import Data.Vec
3    open import Data.Bool
4    open import Data.Product
5    open import Data.String
6    open import Relation.Binary.PropositionalEquality
7    postulate
8      AES : Vec Bit 128 → String → String  -- una función postulada para AES
9    GHASH : ∀ {ℓ} {A : Set ℓ} → (A → Vec Bit 128) → A → Vec Bit 128 → Vec Bit 128
10   GHASH {A = A} embed m c = foldr _⊕_ c (map embed m)
```

```
AESGCM : AEAD (Vec Bit 128) (Vec Bit 96) String String String
AESGCM = record
  { enc = λ k n pt ad →
      let H = AES k (replicate 128 false)
          c = GHASH (λ s → AES k (s ++ as97 n)) ad (replicate 128 false)
      in xor (AES k (as128 n)) pt , xor c (AES k (inc (as128 n)))
  ; dec = λ k n ct ad →
      let c , t = ct
          H = AES k (replicate 128 false)
          pt = xor (AES k (as128 n)) c
          c′ = GHASH (λ s → AES k (s ++ as97 n)) ad (replicate 128 false)
          t′ = xor c′ (AES k (inc (as128 n)))
      in if t == t′ then just pt else nothing
  ; correctness = λ {k} {n} {pt} {ad} → refl
  ; cpa-security = {! omitidas las pruebas !}
  ; int-ctxt = {! de seguridad por brevedad !}
  }
  Where
    as97 : ∀ {n} → Vec Bit n → Vec Bit 97
    as97 {n} v = take 97 (v ++ replicate 128 false)
    as128 : ∀ {n} → Vec Bit n → Vec Bit 128
    as128 {n} v = take 128 (v ++ replicate 128 false)
    inc : Vec Bit 128 → Vec Bit 128
    inc = {! incremento de 128 bits !}
    xor = <_,_>
      Where
        xor : List Bit → List Bit → List Bit
        xor = zipWith _⊕_
```

Aunque los detalles pueden parecer un poco densos al principio, este código demuestra cómo podemos especificar y razonar sobre propiedades criptográficas directamente en Agda. La función `correctness` es demostrable por reflexividad, ya que simplemente invierte el proceso de cifrado. Las pruebas reales de `cpa-security` e `int-ctxt` se omiten por brevedad, pero en principio podrían completarse dentro del asistente de pruebas de Agda.

El verdadero poder de este enfoque surge cuando se combina con otras características de Agda, como los tipos refinados. Por ejemplo, en lugar de simplemente usar `Vec Bit 128` para claves AES, podríamos definir un tipo que imponga la aleatoriedad como un invariante:

```
record AESKey : Set where
  constructor mk
  field
    key     : Vec Bit 128
    .random : TrueRandom key
```

Ahora, el sistema de tipos garantiza que no se puedan crear claves AES sin una prueba de aleatoriedad.

De manera similar, podemos definir tipos para textos cifrados y etiquetas de autenticación que codifiquen sus relaciones:

```
record Ciphertext (PT : Set) : Set where
  constructor mk
  Field
    ct : String
    .tagged : Tag ct
record Tag (CT : String) : Set where
  constructor mk
  Field
    tag : String
    .matches : CT == GHASH tag
```

Aquí, un `Ciphertext` válido siempre debe tener una `Tag` coincidente, según lo verificado por el campo `matches`.

Podemos llevar esto aún más lejos y definir un marco completo para especificar y razonar sobre protocolos criptográficos en Agda, capturando elementos clave como:

```
record AEAD (Key : Set) (Nonce : Set) (PT : Set) (AD : Set) : Set₁ where
  field
    enc : (k : Key) (n : Nonce) (pt : PT) (ad : AD) → Ciphertext PT
    dec : (k : Key) (n : Nonce) (ct : Ciphertext PT) (ad : AD) → Maybe PT
```

De esta forma, el sistema de tipos impone protocolos criptográficos válidos. Intentar descifrar algo que no sea un `Ciphertext` adecuadamente construido resultaría en un error de tipos.

Lo realmente emocionante es que este enfoque se puede extender no solo a primitivas criptográficas, sino también a protocolos y sistemas completos. Consideremos, por ejemplo, un protocolo simple de intercambio de claves como Diffie-Hellman:

```
module DiffieHellman where
  open import Data.Nat
  open import Data.Maybe
```

```
4   postulate
5     prime : ℕ  -- un primo grande para el intercambio de claves
6     _^_ : ℕ → ℕ → ℕ  -- exponenciación modular
7   aliceStep1 : ℕ → ℕ
8   aliceStep1 a = a ^ prime
9   bobStep1 : ℕ → ℕ
10  bobStep1 b = b ^ prime
11  aliceStep2 : ℕ → ℕ → ℕ
12  aliceStep2 a sb = sb ^ a
13  bobStep2 : ℕ → ℕ → ℕ
14  bobStep2 b sa = sa ^ b
15  protocol : ℕ → ℕ → Maybe ℕ
16  protocol a b with aliceStep1 a | bobStep1 b
17  ... | sa | sb = just (aliceStep2 a sb)
18    where
19      _ : aliceStep2 a sb ≡ bobStep2 b sa
20      _ = {! prueba omitida !}
21  protocol _ _ | _ | _ = nothing
```

Aquí, los pasos de Alice y Bob en el protocolo se modelan como funciones separadas, que luego se combinan en una función `protocol` que representa el intercambio completo. El tipo de `protocol` garantiza que, si ambas partes siguen el protocolo correctamente, terminarán con la misma clave compartida, como se demuestra en la cláusula `where` (aunque la prueba en sí se omite por brevedad).

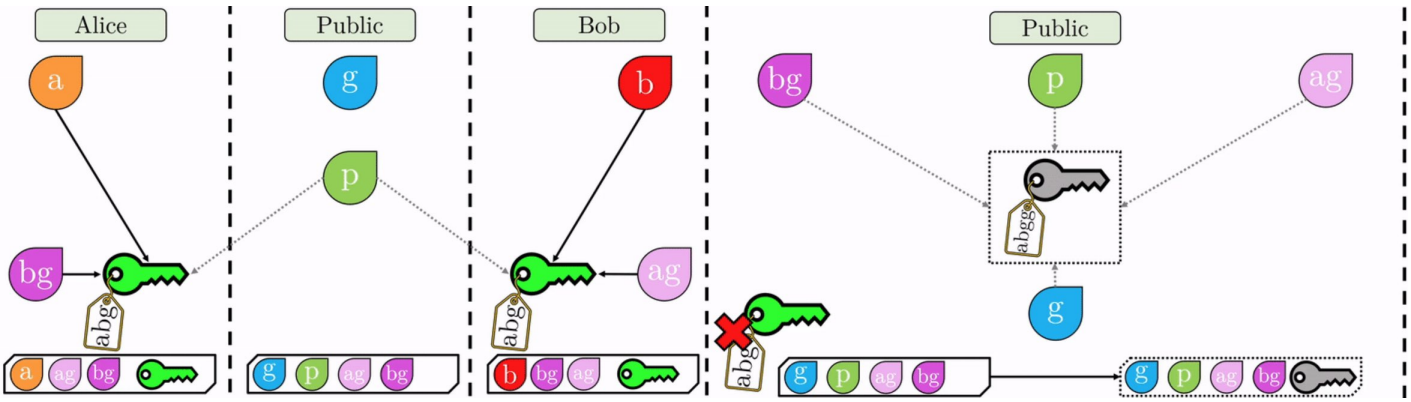


Imagen 27 - Protocolo de Intercambio de Claves Diffie-Hellman.
(Fuente: Alejandro Mantilla)

Podríamos ir un paso más allá y definir un framework completo en Agda para especificar y razonar sobre protocolos criptográficos. Este framework podría incluir tipos para representar aspectos como:

Aspecto	Descripción
Mensajes y formatos de mensaje	Se detallan los mensajes utilizados en el protocolo y sus respectivos formatos.

Funciones de codificación y firmas	Se describen las funciones de codificación utilizadas para procesar los mensajes, así como las firmas digitales si las hay.
Roles de los participantes y sus estados locales	Se explican los roles que desempeñan los participantes en el protocolo y cómo se reflejan estos roles en sus estados locales.
Trazas de ejecución del protocolo y sus propiedades	Se analizan las secuencias de eventos que ocurren durante la ejecución del protocolo y se discuten las propiedades que deben cumplirse en estas trazas para garantizar la seguridad y la integridad del sistema.

Al capturar todo esto en el sistema de tipos, podemos asegurar que las implementaciones sigan los protocolos correctamente por construcción y que se confirmen propiedades clave como el secreto, la autenticación y el forward secrecy.

Este enfoque se extiende también al hardware criptográfico. Con Agda, podríamos especificar y verificar circuitos criptográficos, demostrando que implementan correctamente los algoritmos deseados y son resistentes a ataques de canal lateral. Incluso podríamos sintetizar circuitos criptográficos a partir de especificaciones de alto nivel, asegurando por construcción que cumplan con los requisitos.

Propiedad Criptográfica	Especificación en Agda	Explicación
Correctitud Funcional	correctness : ∀ {n} → (ps : Vec Bit n) → decrypt (encrypt ps) ≡ ps	Encriptar y luego desencriptar un texto plano resulta en el mismo texto plano original
Confidencialidad	confidentiality : ∀ {n m} → (ps qs : Vec Bit n) → (rs : Vec Bit m) → encrypt ps ≡ encrypt qs → ps ≡ qs	Si dos textos planos cifrados son iguales, entonces los textos planos originales también deben ser iguales
Autenticidad	authenticity : ∀ {n} → (ps : Vec Bit n) → (ct : Vec Bit (n + TAG_SIZE)) → decrypt ct ≡ just ps → ct ≡ encrypt ps	Si desencriptar un texto cifrado resulta en un texto plano dado, entonces el texto cifrado debe ser el resultado de encriptar ese mismo texto plano
Integridad	integrity : ∀ {n} → (ps : Vec Bit n) → (ct : Vec Bit (n + TAG_SIZE)) → decrypt (corrupt ct) ≡ nothing	Desencriptar un texto cifrado corrupto (modificado) siempre resulta en nothing, indicando falla
No Maleabilidad	non-malleability : ∀ {n m} → (ps : Vec Bit n) → (f : Vec Bit n → Vec Bit m) → is-valid (f (encrypt ps)) ≡ false	No es posible transformar un texto cifrado en otro texto cifrado válido sin conocer la clave

Agda ya se ha utilizado para verificar e implementar varios sistemas criptográficos del mundo real, incluyendo:

- Una implementación verificada formalmente de la curva elíptica Ed25519 utilizada para firmas digitales
- Verificación del protocolo Off-the-Record (OTR) para mensajería segura
- Verificación de la implementación de la curva elíptica secp256k1 utilizada en Bitcoin

Aunque una verificación formal completa de protocolos criptográficos complejos sigue siendo un desafío significativo, Agda proporciona algunas de las herramientas más poderosas para abordar este reto. Al permitir razonar sobre propiedades criptográficas en los tipos de datos mismos, abre la puerta a una nueva era de software criptográfico más robusto, seguro y confiable.

En un mundo cada vez más dependiente de la criptografía, este es un objetivo de gran importancia. Desde el comercio electrónico hasta las infraestructuras críticas, desde las comunicaciones personales hasta los procesos

democráticos, la criptografía es un componente integral de cualquier sociedad digital funcional. Pero solo es tan buena como las implementaciones que la sustentan.

Con Agda, podemos elevar el listón de lo que es posible en ingeniería criptográfica. Podemos pasar de meramente esperar que los sistemas criptográficos sean seguros a probarlo matemáticamente. Podemos codificar leyes, especificaciones de seguridad y condiciones de corrección en los propios tipos, haciendo que su cumplimiento sea no solo deseado sino obligatorio.

Por supuesto, Agda no es una bala de plata. Verificar sistemas criptográficos reales sigue siendo una tarea compleja y laboriosa, y ninguna cantidad de sofisticación de tipos puede salvarnos de errores en los modelos o especificaciones. Pero proporciona una base sólida sobre la que construir, una que está fundamentalmente alineada con el pensamiento criptográfico en su énfasis en la precisión, la composición segura y el razonamiento matemático riguroso.

Característica	Beneficio para la Criptografía
Funciones Puras	Facilita el razonamiento equacional sobre primitivas criptográficas
Inmutabilidad	Previene errores sutiles de estado y aliasing
Tipos Dependientes	Permite expresar invariantes criptográficos y pre/post condiciones
Pruebas Formales	Proporciona garantías matemáticas de corrección y seguridad
Abstracciones de Alto Nivel	Permite definir y razonar sobre protocolos y sistemas completos
Evaluación Perezosa	Habilita definiciones con estructuras de datos infinitas y recursión no bien fundada
Especificación Ejecutable	Especificaciones son programas que se pueden prototipar y probar

Además, la criptografía es solo una de las muchas áreas donde las técnicas de programación funcional con tipos dependientes pueden tener un impacto. Desde sistemas de control de tráfico aéreo hasta software médico, desde compiladores hasta kernels de sistemas operativos, la capacidad de codificar y hacer cumplir invariantes críticos de seguridad directamente en el sistema de tipos tiene el potencial de revolucionar el desarrollo de software en muchos dominios.

Si te apasiona construir sistemas altamente seguros y correctos, o simplemente te fascina el elegante entrelazamiento del álgebra, la lógica y la computación, vale la pena sumergirte en el mundo de la criptografía funcional con Agda. No solo aprenderás una habilidad increíblemente valiosa, sino que también te unirás a la vanguardia de un nuevo paradigma en cómo abordamos algunos de los desafíos más difíciles y apremiantes en informática. El camino puede ser desafiante a veces, pero las recompensas, tanto intelectuales como prácticas, hacen que valga la pena el esfuerzo.

Conclusiones del Capítulo
En este capítulo, hemos explorado temas avanzados en programación funcional con Agda, desde abstracciones algebraicas hasta técnicas de metaprogramación. Esto amplía nuestras herramientas y horizontes, permitiéndonos escribir código modular y reutilizable. Hemos aprendido sobre mónadas, funtores y teoría de categorías para estructurar efectos y abstracciones comunes. La metaprogramación en Agda, a través de macros y reflexión, nos permite automatizar tareas y elevar el nivel del discurso del programa. Además, vimos cómo Agda puede aplicarse a la criptografía, garantizando seguridad y confianza en algoritmos y protocolos criptográficos.

Ejercicios Propuestos	
1	Define las nociones de categoría, funtor y transformación natural en Agda. Proporciona ejemplos concretos de cada uno.
2	Implementa un pequeño compilador de un lenguaje de expresiones aritméticas a un lenguaje de máquina de pila. Proporciona una prueba formal de la corrección de la compilación.
3	Implementa una macro en Agda que genere automáticamente instancias de la clase `Show` para tipos de datos algebraicos simples.
4	Define un universo que incluya tipos básicos (como `Nat` y `Bool`) y constructores de tipos (como `List` y pares). Implementa funciones genéricas que trabajen sobre este universo.
5	Implementa un sistema de efectos algebraicos que incluya operaciones para estado, excepciones y no determinismo. Proporciona un intérprete para este sistema.
6	Define un sistema de tipos de sesión simple en Agda y úsalo para especificar y verificar un protocolo de comunicación básico entre dos partes.
7	Implementa y verifica formalmente un esquema simple de cifrado (como el cifrado César). Proporciona pruebas de propiedades básicas de seguridad.
8	Implementa un sistema simple de lógica modal en Agda. Define la semántica de Kripke y proporciona un procedimiento de decisión verificado para la validez de fórmulas.
9	Define los tipos de caminos y demuestra algunas propiedades básicas, como la simetría y la transitividad de la igualdad de caminos.
10	Implementa tácticas simples para automatizar partes comunes de las pruebas, como la eliminación de casos triviales o la aplicación de inducción. Demuestra su uso en algunas pruebas complejas.
11	Define la semántica operacional de un pequeño lenguaje imperativo (con asignaciones, condicionales y bucles) en Agda. Prueba algunas propiedades sobre la ejecución de programas en este lenguaje.
12	Implementa un sistema de tipos gradual simple, que permita la coexistencia de tipado estático y dinámico. Demuestra la propiedad de preservación de tipos para este sistema.
13	Implementa una biblioteca básica para trabajar con estados cuánticos y operaciones cuánticas en Agda. Demuestra algunas propiedades fundamentales, como el no-clonado de estados cuánticos.
14	Implementa el isomorfismo de Curry-Howard para la lógica proposicional intuicionista. Demuestra la equivalencia entre pruebas en el sistema lógico y términos bien tipados en el correspondiente sistema de tipos.
15	Especifica y verifica un protocolo distribuido simple (como un protocolo de elección de líder) en Agda. Prueba propiedades de seguridad y vivacidad del protocolo.

Glosario de Términos

Tipos dependientes: Tipos que dependen de valores, permitiendo especificaciones más precisas.

Recursión: Técnica donde una función se llama a sí misma para resolver problemas.

Mónadas: Estructura que encapsula cómputos y efectos secundarios de manera componible.

Pattern matching: Técnica para descomponer datos estructurados y ejecutar código según su forma.

Isomorfismo de Curry-Howard: Correspondencia entre tipos y proposiciones, y programas y pruebas.

Pruebas formales: Demostraciones matemáticas rigurosas de propiedades de programas usando lógica.

Adjunción: Relación entre dos funtores que generaliza la noción de inverso.

Tipos de orden superior: Tipos que pueden tomar otros tipos como parámetros.

Polimorfismo paramétrico: Capacidad de escribir código que funciona con múltiples tipos de datos.

Polimorfismo ad-hoc: Comportamiento diferente de una función según el tipo de sus argumentos.

Singleton: Tipo que contiene exactamente un valor, útil para restricciones estáticas.

Verificación formal: Proceso de probar matemáticamente la corrección de un programa.

Tipos coinductivos: Tipos que representan estructuras potencialmente infinitas o cíclicas.

Lenguaje de dominio específico: Lenguaje de programación especializado para un dominio particular.

Cálculo Lambda: Sistema formal para expresar computación basado en abstracción y aplicación de funciones.

Programación funcional pura: Paradigma que trata la computación como evaluación de funciones matemáticas.

Inmutabilidad: Propiedad de los datos que no pueden ser modificados después de su creación.

Transparencia referencial: Propiedad donde una expresión siempre evalúa al mismo resultado en cualquier contexto.

Efectos secundarios: Modificaciones del estado del programa fuera del valor de retorno de una función.

Currying: Técnica de transformar una función con múltiples argumentos en una secuencia de funciones.

Functores: Estructuras que permiten mapear funciones sobre contenedores de valores.

Tipos algebraicos de datos: Tipos compuestos formados por producto y suma de otros tipos.

Evaluación perezosa: Estrategia que retrasa la evaluación de expresiones hasta que su valor es necesario.

Tipos refinados: Tipos que incluyen predicados para especificar propiedades más precisas de los valores.

Tipos indexados: Tipos parametrizados por valores, permitiendo especificaciones más detalladas.

Proposiciones como tipos: Principio donde los tipos representan proposiciones y los términos representan pruebas.

Teoría de categorías: Estudio matemático abstracto de estructuras y relaciones entre objetos.

Metaprogramación: Técnica de escribir programas que generan o manipulan otros programas.

Macros: Reglas o patrones que especifican cómo cierto input debe ser mapeado a un output.

Argumentos implícitos: Argumentos de función que el compilador puede inferir y no necesitan especificarse explícitamente.

Argumentos irrelevantes: Argumentos de función que no afectan el resultado computacional, solo la tipificación.

Cuantificación universal: Expresión de que una propiedad o relación es válida para todos los elementos.

Inferencia bidireccional: Técnica de inferencia de tipos que combina la propagación hacia adelante y hacia atrás.

Unificación: Proceso de resolver ecuaciones entre términos simbólicos.

Funtores aplicativos: Functores con una estructura adicional que permite levantar funciones sobre efectos.

Transformadores de mónada: Tipos que toman una mónada y devuelven otra, añadiendo nuevas capacidades.

Combinadores: Funciones de orden superior que componen otras funciones de manera estructurada.

Bibliografía

Agda Development Team. (2023). Agda Documentation (Version 2.6.5). https://agda.readthedocs.io/

Benton, N., Hughes, J., & Moggi, E. (2000). Monads and effects. In International Summer School on Applied Semantics (pp. 42-122). Berlin, Heidelberg: Springer Berlin Heidelberg.

Chacón Sartori, C. (2021). Introducción a la programación funcional usando Haskell y Agda. https://doi.org/10.13140/RG.2.2.34182.29764

Geuvers, J. H. (1992). Inductive and coinductive types with iteration and recursion.

Lilis, Y., & Savidis, A. (2019). A Survey of Metaprogramming Languages. ACM Computing Surveys, 52(6), Article 113. https://doi.org/10.1145/3354584

Mosses, P. D. (1990). Chapter 11 - Denotational Semantics. In J. Van Leeuwen (Ed.), Formal Models and Semantics (pp. 575-631). Elsevier. https://doi.org/10.1016/B978-0-444-88074-1.50016-0

Norell, U. (2008). Dependently typed programming in Agda. In Proceedings of the 6th International Conference on Advanced Functional Programming (pp. 230–266). Springer-Verlag.

Schäfer, M., & De Moor, O. (2010, January). Type inference for datalog with complex type hierarchies. In Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 145-156).

Stump, A. (2016). Verified Functional Programming in Agda. Association for Computing Machinery and Morgan & Claypool.

Wadler, P. (1992, February). The essence of functional programming. In Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 1-14).

Wadler, P., & Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (pp. 60–76). Association for Computing Machinery. https://doi.org/10.1145/75277.75283

Wadler, P., Kokke, W., & Siek, J. G. (2022, August). Programming Language Foundations in Agda. https://plfa.inf.ed.ac.uk/22.08/

Xi, H., & Pfenning, F. (1999, January). Dependent types in practical programming. In Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 214-227).

Agradecimientos Finales

Al concluir este libro, me gustaría expresar mi más profunda gratitud a todos aquellos que han hecho posible este viaje. En primer lugar, a mi familia, cuyo constante apoyo y aliento han sido un faro en los momentos más desafiantes de este proceso. Su amor y comprensión han sido fundamentales para mantenerme centrado y motivado.

A los pioneros y contribuidores del lenguaje Agda y su ecosistema, cuya visión y arduo trabajo han abierto nuevos horizontes en lo que es posible en la programación. Parados sobre sus hombros, podemos ver más lejos y aspirar a nuevas alturas.

Y finalmente, a ustedes, queridos lectores, por embarcarse en este viaje conmigo. Espero que las ideas y técnicas exploradas en este libro resuenen con ustedes, iluminen su práctica y enciendan su curiosidad. Su entusiasmo y sed de conocimiento son la razón por la que escritores como yo hacen lo que hacen.

Juntos, sigamos impulsando los límites de lo que es posible con la programación funcional dependiente de tipos. Sigamos buscando elegancia en nuestro código, precisión en nuestros razonamientos y belleza en las matemáticas de la computación.

Con sincera gratitud,

Luis Eduardo Muñoz Guerrero