Universidad Nacional Autónoma de México

Facultad de Ciencias

Lenguajes de Programación

Proyecto 2:
Inferencia lógica y sistemas de tipos

Integrantes:

Elizalde Maza Jesús Eduardo
Navarro Fierro Michelle Alanis
Peredo López Citlalli Abigail

Profesor: Manuel Soto Romero

Diego Méndez Medina
José Alejandro Pérez Márquez
Erick Daniel Arroyo Martínez
Mauro Emiliano Chávez Zamora

Fecha: 15 de diciembre de 2025

CDMX

Índice

1. Introducción

El estudio de los lenguajes de programación ha estado siempre acompañado por la necesidad de establecer mecanismos formales que permitan describir, controlar y verificar el comportamiento de los programas. Entre estos mecanismos, los sistemas de tipos han adquirido un papel central, pues proporcionan una estructura rigurosa para clasificar expresiones y garantizar que operen de manera coherente. La *inferencia de tipos*, entendida como la capacidad de un sistema o compilador para deducir automáticamente el tipo de una expresión, ha sido fundamental para el desarrollo tanto teórico como práctico de los lenguajes modernos.

Históricamente, la inferencia de tipos surgió de la intersección entre la teoría de lenguajes y la lógica matemática. Desde los primeros trabajos de Curry, Church y Turing en la década de 1930, la relación entre funciones, tipos y proposiciones comenzó a tomar forma [1,2].Más tarde, durante los años sesenta y setenta, investigaciones como las de Hindley y Milner consolidaron sistemas de inferencia que permitieron diseñar lenguajes expresivos y seguros sin exigir anotaciones de tipos exhaustivas. Paralelamente, los avances en la lógica de predicados de primer orden ofrecieron un marco formal para razonar sobre la validez de enunciados, convirtiéndose en una herramienta fundamental en la verificación y en la semántica formal.

La conexión entre estos campos no es accidental: los sistemas de tipos pueden interpretarse como sistemas lógicos, y los procesos de inferencia de tipos comparten estructuras con los mecanismos de inferencia lógica en sistemas formales. Esta relación ha dado lugar a principios que han guiado el diseño de lenguajes modernos, permitiendo que conceptos provenientes de la lógica se traduzcan en herramientas prácticas como la verificación automática, la prevención de errores en tiempo de ejecución y la creación de lenguajes más robustos y expresivos.

Comprender la historia y el trasfondo lógico de la inferencia de tipos no sólo ofrece una visión más profunda de los lenguajes de programación actuales, sino que también permite apreciar cómo los fundamentos teóricos continúan influyendo en su evolución. La relevancia de este tema radica en que constituye un puente entre la teoría matemática y la ingeniería de software, revelando cómo los métodos formales han moldeado, y continúan moldeando, la manera en que construimos y razonamos sobre programas. [1, 4]

2. Objetivos

Este proyecto tiene como objetivo general responder la pregunta central: *¿Qué relación existe entre los sistemas de tipos y la inferencia lógica en sistemas formales de primer orden?* Para abordar esta cuestión, se propone el lenguaje AURA el cual busca establecer un puente conceptual que permita comprender de manera rigurosa cómo los sistemas de tipos pueden interpretarse como mecanismos de comprobación lógica, donde cada expresión del programa funciona como una proposición y cada tipo como una propiedad que debe ser demostrada. A partir de esta perspectiva, se pretende mostrar cómo la inferencia de tipos realizada por los compiladores se fundamenta en principios lógicos capaces de derivar, verificar y garantizar la validez formal de las expresiones del lenguaje.

Con base en este objetivo general, el proyecto se plantea los siguientes objetivos específicos:

- Explicar el papel de los sistemas de tipos en la estructura y seguridad de los lenguajes de programación, destacando su relevancia para el control de errores y la verificación anticipada.
- Describir los fundamentos de la inferencia lógica en sistemas formales de primer orden y su importancia en el razonamiento matemático y computacional.
- Analizar de manera conceptual cómo la inferencia de tipos se corresponde con procesos de deducción lógica, mostrando los paralelos entre reglas de tipado y reglas de inferencia lógica.
- Identificar los beneficios prácticos y teóricos que surgen al comprender esta relación, especialmente en el diseño de lenguajes modernos, la construcción de compiladores robustos y el desarrollo de herramientas de verificación formal.
- Proporcionar un marco integrador que permita entender por qué nociones como tipificación estática, inferencia automática y pruebas formales no son elementos independientes, sino manifestaciones de un mismo fundamento lógico.

Al alcanzar estos objetivos, el proyecto busca no sólo responder la pregunta planteada, sino también ofrecer una comprensión más profunda de la conexión entre teoría matemática y práctica de programación. De este modo, se pretende resaltar la importancia de dicha relación para el diseño disciplinado, seguro y confiable de software en la ingeniería computacional contemporánea.

3. Marco Teórico

3.1. Sistemas de Tipos

Un sistema de tipos consiste en un conjunto de reglas de inferencia que establecen restricciones sobre cómo se pueden construir los programas. Mediante los tipos, se determina la clasificación de las expresiones del lenguaje, indicando de qué manera pueden combinarse entre sí. De manera intuitiva, el tipo de una expresión permite anticipar la naturaleza de su resultado: por ejemplo, si se suman dos expresiones numéricas, el resultado también deberá ser numérico. Por el contrario, intentar sumar un valor numérico con uno booleano, como en `true + 7`, debe considerarse inválido porque provoca un error de tipos.

En términos generales, un tipo puede entenderse como una descripción abstracta de un conjunto de valores posibles. Aunque esta idea funciona en casos simples —como tipos representando números enteros o valores flotantes— no resulta adecuada cuando se trata de tipos más complejos, como los tipos de funciones o los tipos polimórficos. En realidad, la interpretación formal de los tipos suele basarse en estructuras matemáticas como los órdenes parciales, especialmente dentro del marco de la semántica denotacional.

La incorporación de un sistema de tipos en el diseño de un lenguaje de programación proporciona diversas ventajas:

- Permite detectar errores de programación en etapas tempranas.
- Aporta seguridad, ya que un programa bien tipado evita fallos de ejecución relacionados con tipos.
- Facilita la abstracción, fundamental para definir interfaces.
- Ayuda a documentar el código de forma más clara y manejable que los comentarios.
- Contribuye a una implementación del lenguaje más eficiente y ordenada [1, 5]

3.2. Importancia de los Sistemas de Tipos

Los sistemas de tipos desempeñan un papel fundamental en el diseño y funcionamiento de los lenguajes de programación. Su objetivo principal es garantizar que los valores se usen correctamente dentro de un programa, evitando errores y mejorando la calidad del software.

- Seguridad y corrección

Un programa bien tipado no se detiene inesperadamente por errores de tipo, lo que asegura un comportamiento correcto durante su ejecución.

La seguridad en los sistemas de tipos vincula la semántica estática con la semántica dinámica, y su demostración suele dividirse en dos propiedades principales:

- **Progreso:** Todo programa que cumple con las reglas de tipado puede continuar evaluándose sin quedar bloqueado.
- **Preservación:** Si un programa bien tipado avanza en su evaluación, la expresión resultante también mantiene un tipo válido; en muchos casos, conserva exactamente el mismo tipo.

- Prevención de errores en compilación

Los sistemas de tipos permiten detectar fallos antes de ejecutar el programa, reduciendo los *bugs* y proporcionando una descripción formal del lenguaje.

- Documentación implícita

Los tipos sirven como guía para entender el código, facilitando el mantenimiento y la productividad del desarrollador sin necesidad de comentarios adicionales.

- Optimización del compilador

La información de tipos habilita generación de código más eficiente, al eliminar comprobaciones innecesarias en tiempo de ejecución.

- Soporte al desarrollo a gran escala

Al definir interfaces claras y minimizar dependencias, los sistemas de tipos permiten la colaboración organizada entre múltiples desarrolladores. [3–5]

3.3. Correspondencia de Curry–Howard

La correspondencia de Curry–Howard es una relación profunda y explícita entre los sistemas de lógica formal (demostraciones matemáticas) y los sistemas formales de computación (programas). En esta relación, las proposiciones lógicas se interpretan como tipos en un sistema de tipado; las pruebas de esas proposiciones como programas (o términos); y la demostración como la construcción de un valor del tipo correspondiente.

Más formalmente, según esta correspondencia: [16, 17]

A nivel de fórmulas y tipos

La relación con la lógica de primer orden aparece mediante el siguiente paralelismo:

- Tipos polimórficos $\leftrightarrow$ cuantificación universal ($\forall$).
- Variables de tipo $\leftrightarrow$ variables lógicas.
- Instanciación de tipos $\leftrightarrow$ eliminación de cuantificadores.

Ejemplo:

$$\forall \alpha.\ \alpha \to \alpha$$

corresponde al tipo de la función identidad, análogo a una fórmula universal en lógica de primer orden. [17, 18]

A nivel de demostraciones y programas

Proveer una prueba de una proposición equivale a definir un programa de un tipo determinado: las reglas de introducción y eliminación de los conectivos se interpretan como operaciones de construcción y deconstrucción de valores en el sistema de tipos (por ejemplo, abstracción y aplicación en el cálculo lambda).

En síntesis: **una prueba es un programa; una proposición es un tipo**. [14, 18]

3.3.1. Importancia de la Correspondencia de Curry–Howard

La correspondencia de Curry–Howard no es una mera curiosidad matemática o filosófica; tiene consecuencias fundamentales para la lógica, la teoría de la computación y el diseño de lenguajes de programación. Entre sus implicaciones más destacadas:

- **Puente entre lógica y computación / programas verificados.** Permite garantizar correctitud por construcción: si un programa pasa la verificación de tipos, entonces existe una prueba válida de su corrección.
- **Fundamento de lenguajes funcionales y teoría de tipos.** Conceptos como tipos algebraicos, polimorfismo y tipos dependientes encuentran su base en principios lógicos.
- **Soporte a la verificación formal y asistentes de prueba.** Con tipado dependiente, las proposiciones pueden representarse como tipos y las pruebas como programas certificados.
- **Unificación de lógica intuicionista y cálculo lambda.** Los sistemas de tipos pueden verse como sistemas lógicos; los programas, como pruebas construidas mediante cálculo lambda.

Por estas razones, la correspondencia de Curry–Howard se considera un pilar conceptual en la teoría de la computación, la semántica de lenguajes y la verificación formal. [14, 17]

3.3.2. Relación con los lenguajes de programación

Este enfoque influye fuertemente en el diseño de sistemas de tipos avanzados, en la inferencia automática de tipos (como en Haskell), y en lenguajes destinados a la formalización matemática de programas, pues en este caso la correspondencia de Curry–Howard proporciona un fundamento teórico sólido con consecuencias prácticas: lenguajes más seguros, sistemas de tipos más expresivos y la posibilidad de verificar formalmente propiedades de programas.

Podemos verlo como que las proposiciones lógicas corresponden a tipos, y las pruebas de esas proposiciones corresponden a programas.

Esta correspondencia se expresa en el siguiente paralelismo:

Lógica	**Lenguajes de programación**
Proposición	Tipo
Prueba	Programa
Demostración válida	Programa bien tipado
Eliminación / introducción	Aplicación / abstracción

Así, escribir un programa bien tipado equivale a construir una demostración de una proposición lógica. [2, 12]

3.4. Juicios de tipado como inferencias lógicas

En un lenguaje de programación tipado, el objeto central no es el programa en sí, sino el *juicio de tipado*. Este se escribe como:

$$\Gamma \vdash e : \tau$$

y se lee:

"Bajo el contexto Γ, la expresión e tiene tipo τ".

Aquí:

- Γ es un contexto que asigna tipos a variables libres,
- e es una expresión del lenguaje,
- τ es un tipo.

Esto no es simplemente una notación elegante para decorar documentos: es lógica pura. Tiene exactamente la misma forma que un juicio de deducción en un sistema formal. [11, 12]

Analogía directa con la lógica

En lógica proposicional o de primer orden, un juicio típico es:

$$\Gamma \vdash \varphi$$

lo cual significa:

"φ es deducible a partir de las hipótesis Γ".

Bajo la correspondencia de Curry–Howard se establece el siguiente paralelismo:

Lógica	**Tipos**
Hipótesis	Variables tipadas
Proposición	Tipo
Prueba	Término
Deducción	Derivación de tipado

El contexto de tipos Γ cumple el mismo papel que el conjunto de suposiciones lógicas.

Reglas de tipado como reglas de inferencia

Un sistema de tipos se define mediante reglas, escritas como reglas de inferencia lógica. Un ejemplo clásico es la regla de *aplicación de funciones*:

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash f\, x : \tau_2}$$

Esta regla es estructuralmente idéntica a una regla lógica como el *modus ponens*:

$$\frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$

El patrón es exactamente el mismo. La única diferencia es que, en el sistema de tipos, las "proposiciones" reciben el nombre de tipos. [5, 12, 17]

Derivaciones de tipado como árboles de prueba

Un programa bien tipado no es solamente una expresión correcta: es una prueba completa. Dicha prueba puede representarse como un árbol de derivación, de manera análoga a los árboles de prueba en lógica formal.

Por ejemplo, en el cálculo lambda tipado, cada término bien tipado corresponde a un árbol de inferencia que justifica formalmente su tipo.

En lo que sigue, se introduce un lenguaje de expresiones aritméticas y booleanas, y se analiza detalladamente su semántica formal,tanto estática como dinámica, empleando las herramientas matemáticas previamente descritas. [4, 5]

$$\frac{\Gamma, x : \tau_1 \vdash x : \tau_1}{\Gamma \vdash \lambda x.\, x : \tau_1 \to \tau_1}\ (\textsc{var})$$

Esto constituye una demostración formal de que $\lambda x.\, x$ tiene tipo $\tau_1 \to \tau_1$. No es nada distinto de una prueba lógica construida paso a paso.

Si el compilador no puede construir este árbol de derivación, entonces el programa no es demostrable. El juicio falla. [12]

Introducción y eliminación: la simetría lógica

Cada constructor del lenguaje posee dos caras lógicas, correspondientes a reglas de introducción y eliminación:

Lenguaje	**Lógica**
Abstracción $\lambda x.\, e$	Introducción de implicación
Aplicación $f\, x$	Eliminación de implicación
Par (e_1, e_2)	Introducción de conjunción
Proyecciones *fst*, *snd*	Eliminación de conjunción
`case` / `match`	Eliminación de disyunción

Esta la razón por la cual el lenguaje *tiene sentido*.

4. Lenguaje AURA

El lenguaje **AURA** se propone como un lenguaje diseñado para ilustrar la relación entre los sistemas de tipos y la inferencia lógica en sistemas formales de primer orden. Su propósito es ofrecer un marco simplificado en el cual se puedan observar de manera clara los principios teóricos que sustentan la inferencia de tipos [1, 3].

4.1. Motivación

La creación de AURA responde a la necesidad de contar con un entorno controlado que permita demostrar cómo las reglas de tipado pueden interpretarse como reglas de inferencia lógica [2]. Al definir un lenguaje mínimo, se facilita la exposición de conceptos como la correspondencia Curry–Howard [1, 4] y la aplicación del algoritmo W [3, 9], evitando la complejidad de otros lenguajes.

4.2. Sintaxis básica

AURA se construye sobre una sintaxis inspirada en el cálculo λ [5], con las siguientes expresiones fundamentales:

- **Variables:** x, y, z
- **Abstracción:** $\lambda x.e$ (función que recibe un argumento x y devuelve la expresión e)
- **Aplicación:** $(e_1\ e_2)$ (aplicación de la función e_1 al argumento e_2)
- **Constantes básicas:** números enteros $(0, 1, 2, \dots)$ y booleanos (`true`, `false`)

Ejemplos de expresiones en AURA:

- $\lambda x.x$ (función identidad)
- $\lambda x.\lambda y.x$ (función que ignora su segundo argumento)
- $(\lambda x.x)(\lambda y.y)$ (aplicación de la identidad a otra identidad)

4.3. Sistema de tipos

El sistema de tipos de AURA se basa en el sistema Hindley–Milner [3, 7, 15], que permite inferencia polimórfica. Los tipos básicos incluyen:

- `Int` (enteros)
- `Bool` (booleanos)
- Tipos de función: $\tau_1 \to \tau_2$
- Variables de tipo: $\alpha, \beta, \gamma, \dots$

Ejemplo de tipado:

- $\lambda x.x \quad \Rightarrow \quad \alpha \to \alpha$
- $\lambda x.\lambda y.x \quad \Rightarrow \quad \alpha \to \beta \to \alpha$
- $(\lambda x.x)(\lambda y.y) \quad \Rightarrow \quad \gamma \to \gamma$

4.4. Relación con la lógica

Cada regla de tipado en AURA puede interpretarse como una regla de inferencia lógica [2, 6]:

- La **abstracción** corresponde a la introducción de la implicación en lógica.
- La **aplicación** corresponde al modus ponens.
- Las **variables de tipo** representan cuantificadores universales en lógica de primer orden.

De esta manera, AURA se convierte en un lenguaje que materializa la correspondencia Curry–Howard [1, 4], mostrando cómo los programas son pruebas y los tipos son proposiciones.

4.5. Correspondencia de Curry-Howard en AURA

Anteriormente ya hemos hablado de la correspondencia de Curry-Howard y su importancia en la lógica y en los lenguajes de programación. Ahora veamoslo de una forma un poco más concreta, abordando más su aplicación a nuestro lenguaje AURA, pues esto es escencial como fundamento sustento teórico de nuestro trabajo.

El isomorfismo de Curry–Howard establece una correspondencia profunda entre tipos, lógica y teoría de categorías:

Tipos	**Lógica**	**Categorías**
$A \to B$	Implicación	Exponencial
$A \times B$	Conjunción	Producto
$A + B$	Disyunción	Coproducto
$\forall a.\, \tau$	Universal	Límite / Fin
$\exists a.\, \tau$	Existencial	Colímite / Cofin

En AURA:

- Un programa bien tipado es una prueba constructiva.
- Ejecutar un programa equivale a simplificar una prueba.
- Un error de tipo corresponde a una prueba inválida.

5. Algoritmo W

5.0.1. Descripción general

El **algoritmo W** es el procedimiento clásico para realizar inferencia de tipos dentro del sistema Hindley–Milner [3]. Propuesto por Luis Damas y Robin Milner en 1982, este algoritmo permite obtener automáticamente el tipo más general de una expresión sin requerir anotaciones de tipo explícitas. Su impacto ha sido decisivo en el diseño de lenguajes funcionales como Haskell y la familia ML, al ofrecer una combinación efectiva entre tipado estático seguro y una sintaxis flexible.

5.0.2. El algoritmo en el sistema Hindley–Milner

El sistema Hindley–Milner se caracteriza por soportar polimorfismo paramétrico y por garantizar que toda expresión bien formada posea un *tipo principal*. Dicho tipo principal es el más general posible, de modo que cualquier otro tipo admisible puede obtenerse mediante instanciación [1]. En este marco, el algoritmo W actúa como el método operativo que calcula ese tipo de manera sistemática, preservando tanto la corrección como la completitud del sistema de tipos [4].

5.0.3. Mecanismo de operación

El algoritmo W trabaja esencialmente en tres etapas:

1. **Generación de restricciones:** Se examina la estructura de la expresión y se producen ecuaciones que describen cómo deben relacionarse los tipos de sus subexpresiones.

2. **Unificación:** Las restricciones generadas se resuelven mediante unificación, encontrando sustituciones para las variables de tipo que satisfacen todas las ecuaciones simultáneamente.

3. **Obtención del tipo principal:** Aplicadas las sustituciones, se obtiene el tipo final de la expresión. Este tipo puede contener variables polimórficas si la expresión lo permite. [3]

Si la expresión es tipable, el algoritmo produce el tipo más general posible. En caso contrario, indica la presencia de un error de tipos.

5.0.4. Correspondencia lógica

El algoritmo W puede interpretarse también desde la lógica de primer orden, pues sus pasos reflejan reglas de inferencia lógicas:

- La **abstracción** ($\lambda x.e$) corresponde a la regla de *introducción de la implicación*: asumir que x tiene tipo α para concluir que e tiene tipo β permite obtener $\alpha \to \beta$.
- La **aplicación** ($e_1\ e_2$) se relaciona con la regla de *modus ponens*: a partir de una función de tipo $\alpha \to \beta$ y un argumento de tipo α, se deduce un resultado de tipo β.
- La construcción **let** se vincula con la *generalización* o cuantificación universal, que permite definir expresiones con tipos polimórficos reutilizables mediante instanciación.

En suma, el algoritmo W no solo es un procedimiento de inferencia, sino una expresión concreta de la correspondencia Curry–Howard [4], donde determinar el tipo de un programa equivale a construir una prueba en un sistema lógico.

5.1. El algortimo W en AURA

El algoritmo W puede entenderse como un procedimiento de búsqueda de pruebas en un sistema de inferencia de tipos. Cada constructor del lenguaje corresponde a una regla de inferencia, y la unificación resuelve las ecuaciones generadas por dichas reglas.

Por ejemplo:

- **Variables**: usan $\forall$-eliminación (instanciación).
- **Abstracciones** λ: introducen supuestos en el entorno.
- **Aplicaciones**: generan ecuaciones de tipos que son resueltas por unificación.
- **Let**: combina $\forall$-introducción (generalización) y $\forall$-eliminación.

De esta manera, la inferencia de tipos se reduce a un problema de inferencia lógica sobre ecuaciones de primer orden, donde el unificador más general garantiza la maximalidad del tipo inferido.

6. Componentes clave del algoritmo W

6.1. Sustituciones (*Subst*)

Las sustituciones son funciones parciales de variables de tipo a tipos:

```
type Subst = Map.Map TVar Type
```

Corresponden a instanciaciones en lógica. Cuando se unifica a con `Int`, se establece que en todas las fórmulas donde aparece a, debe reemplazarse por `Int`.

Composición de sustituciones.

```
compose s1 s2 = Map.map (apply s1) s2 `Map.union` s1
```

Esto refleja la composición de transformaciones: aplicar s_2 y luego s_1 es equivalente a aplicar `compose` s_1 s_2.

6.2. Unificación

La unificación constituye el núcleo algorítmico del sistema:

```
unify :: Type -> Type -> Infer Subst
```

La unificación resuelve ecuaciones de tipos. Por ejemplo, al escribir:

```
f x = x + 1
```

el algoritmo genera las siguientes restricciones:

- $x : a$ (variable fresca),
- $(+) : \texttt{Int} \to \texttt{Int} \to \texttt{Int}$,

por lo tanto $a = \texttt{Int}$.

La unificación encuentra el *unificador más general* (mgu), es decir, la sustitución mínima que hace iguales dos tipos.

Reglas de unificación (análogas a lógica de primer orden).

- **Constantes:** $\texttt{Int} \sim \texttt{Int} \Rightarrow \emptyset$.
- **Variables:** $a \sim \tau \Rightarrow [a \mapsto \tau]$ si $a \notin \text{ftv}(\tau)$.
- **Funciones:**

$$(\tau_1 \to \tau_2) \sim (\sigma_1 \to \sigma_2) \Rightarrow \text{compose}(\text{unify } \tau_1 \ \sigma_1)(\text{unify } \tau_2 \ \sigma_2).$$

6.3. Variables frescas

```
fresh :: Infer Type
fresh = do
  i <- get
  put (i + 1)
  return (TVar ("a" ++ show i))
```

Corresponden a variables libres en lógica. Cada variable fresca evita colisiones de nombres, de manera análoga a la conversión α en el cálculo lambda.

6.4. Instanciación

La instanciación elimina cuantificadores universales de un esquema de tipos, reemplazando cada variable cuantificada por una variable de tipo fresca.

```
instantiate :: Scheme -> Infer Type
```

Si se tiene el esquema:

$$\forall a_1 \dots a_n.\ \tau$$

la instanciación produce un tipo τ' donde cada a_i es reemplazada por una variable de tipo fresca distinta.

La instanciación corresponde a la regla de eliminación del cuantificador universal ($\forall$-eliminación) en lógica de predicados:

$$\frac{\forall x.\ P(x)}{P(t)}$$

En el contexto de inferencia de tipos, esta regla permite usar una función polimórfica con tipos concretos distintos en cada uso, evitando dependencias entre aplicaciones independientes.

6.5. Generalización

La generalización introduce cuantificadores universales sobre aquellas variables de tipo que no aparecen libres en el entorno de tipos.

```
generalize :: TypeEnv -> Type -> Scheme
```

Formalmente, si Γ es el entorno y τ el tipo inferido:

$$\text{generalize}(\Gamma, \tau) = \forall \alpha_1 \dots \alpha_n.\ \tau$$

donde:

$$\{\alpha_1, \dots, \alpha_n\} = FV(\tau) - FV(\Gamma)$$

Este proceso implementa la regla de introducción del cuantificador universal ($\forall$-introducción):

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall \alpha.\ \tau}$$

La generalización se aplica exclusivamente en construcciones `let`, lo que permite definir funciones polimórficas reutilizables sin comprometer la consistencia del sistema de tipos.

6.6. Entornos de tipos

El entorno de tipos mantiene la asociación entre variables del lenguaje y sus esquemas de tipo:

```
type TypeEnv = Map.Map String Scheme
```

El entorno Γ corresponde a un conjunto de hipótesis en un sistema de deducción:

$$\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$$

Cada juicio de tipos tiene la forma:

$$\Gamma \vdash e : \tau$$

Durante la inferencia, el entorno se extiende al introducir variables ligadas (`lambda`, `let`) y se transforma al aplicar sustituciones, preservando la coherencia de las hipótesis.

7. Reglas de inferencia del algoritmo W

7.1. Variables (VAR)

$$\frac{x : \sigma \in \Gamma \qquad \tau = \text{instantiate}(\sigma)}{\Gamma \vdash x : \tau}$$

Implementación.

```
EVar x -> do
  case Map.lookup x env of
    Nothing -> error $ "Variable no ligada: " ++ x
    Just scheme -> do
      t <- instantiate scheme
      return (nullSubst, t)
```

Instanciar un esquema es análogo a usar un axioma en una prueba lógica.

7.2. Literales (LIT)

$$\Gamma \vdash n : \texttt{Int} \qquad\qquad \Gamma \vdash b : \texttt{Bool}$$

Implementación.

```
ENum _  -> return (nullSubst, TInt)
EBool _ -> return (nullSubst, TBool)
```

Los literales actúan como axiomas con tipos conocidos.

7.3. Abstracción lambda (ABS)

$$\frac{\Gamma, x : \alpha \vdash e : \tau \qquad (\alpha \text{ fresca})}{\Gamma \vdash \lambda x.e : \alpha \to \tau}$$

Implementación.

```
ELam x body -> do
  tv <- fresh
  let env' = Map.insert x (Forall [] tv) env
  (s1, t1) <- infer env' body
  return (s1, TFun (apply s1 tv) t1)
```

Corresponde a la introducción de la implicación ($\to I$).

7.4. Aplicación (APP)

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \sim \tau_2 \to \alpha}{\Gamma \vdash e_1\, e_2 : \alpha}$$

Implementación.

```
EApp e1 e2 -> do
  (s1, t1) <- infer env e1
  (s2, t2) <- infer (apply s1 env) e2
  tv <- fresh
  s3 <- unify (apply s2 t1) (TFun t2 tv)
  return (compose s3 (compose s2 s1), apply s3 tv)
```

Equivale al *modus ponens* ($\to E$).

7.5. Let polimórfico (LET)

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \sigma = \mathrm{Gen}(\Gamma, \tau_1) \quad \Gamma, x : \sigma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau_2}$$

Implementación.

```
ELet x e1 e2 -> do
  (s1, t1) <- infer env e1
  let env' = apply s1 env
      scheme = generalize env' t1
      env'' = Map.insert x scheme env'
  (s2, t2) <- infer env'' e2
  return (compose s2 s1, t2)
```

Implementa la generalización universal ($\forall I$).

8. Generalización e instanciación

8.1. Generalización

```
generalize :: TypeEnv -> Type -> Scheme
generalize env t =
```

```
  Forall vars t
  where
    vars = Set.toList (ftv t `Set.difference` ftv env)
```

Corresponde a la cuantificación universal sobre variables libres.

8.2. Instanciación

```
instantiate :: Scheme -> Infer Type
instantiate (Forall vars t) = do
  freshVars <- mapM (const fresh) vars
  let s = Map.fromList (zip vars freshVars)
  return (apply s t)
```

Equivale a la eliminación del cuantificador universal ($\forall E$).

8.2.1. Propiedades del algoritmo W

Corrección
Si $W(\Gamma, e) = (S, \tau)$, entonces $S(\Gamma) \vdash e : S(\tau)$ es derivable.
Completitud
Si existe una derivación $\Gamma \vdash e : \tau$, entonces el algoritmo W termina exitosamente.
Tipo principal
El tipo inferido es el más general posible. Por ejemplo:

$$\lambda x.x : \forall a.\ a \to a$$

9. Conclusiones y resultados

En el presente proyecto se estudió e implementó el algoritmo W como mecanismo de inferencia de tipos para un lenguaje funcional mínimo, con soporte para enteros, booleanos, abstracciones lambda, aplicación de funciones y expresiones `let`. A través de este desarrollo se demostró que, aun en lenguajes pequeños, la inferencia de tipos constituye una herramienta poderosa tanto desde el punto de vista teórico como práctico, al permitir razonar formalmente sobre los programas sin requerir anotaciones explícitas por parte del programador.

Uno de los principales aportes del trabajo es la interpretación de la inferencia de tipos como un proceso de inferencia lógica. Los juicios de tipado se analizaron como fórmulas dentro de un sistema formal, mientras que la unificación se entendió como un procedimiento para resolver ecuaciones de primer orden. Bajo esta perspectiva, el sistema de tipos deja de verse únicamente como un mecanismo de detección de errores y pasa a concebirse como un sistema de razonamiento formal automatizado.

El sistema implementado se fundamenta en el esquema de Hindley–Milner, incorporando conceptos esenciales como sustituciones, variables de tipo frescas, esquemas de tipo, instanciación y generalización. Estos mecanismos permitieron inferir tipos polimórficos de forma automática, decidible y correcta, evidenciando la importancia del polimorfismo paramétrico en el diseño de lenguajes funcionales modernos. En particular, se mostró que la generalización e instanciación corresponden directamente a la introducción y eliminación del cuantificador universal en la lógica de predicados, reforzando la relación entre sistemas de tipos y lógica formal.

Asimismo, el análisis de las propiedades del algoritmo W confirmó su solidez teórica. Se discutieron las propiedades de corrección, completitud y existencia de tipo principal, mostrando que el algoritmo no solo produce tipos válidos cuando termina exitosamente, sino que además genera el tipo más general posible para cada expresión. Estas propiedades son fundamentales para garantizar la consistencia del sistema de tipos y su utilidad práctica en compiladores e intérpretes reales.

No obstante, también se identificaron las limitaciones inherentes al sistema Hindley–Milner. En particular, el algoritmo W no soporta polimorfismo de rango superior, subtipado ni tipos dependientes, lo que restringe su expresividad frente a sistemas de tipos más avanzados. Estas limitaciones no invalidan su relevancia, pero sí delimitan claramente el alcance del modelo implementado.

Por otro lado, una limitación adicional del proyecto es que el lenguaje desarrollado depende de Haskell como lenguaje anfitrión, tanto a nivel de implementación como de semántica subyacente. Si bien esta elección facilitó el desarrollo y permitió concentrarse en el estudio del algoritmo W y del sistema de tipos, una posible mejora futura sería diseñar el lenguaje de manera independiente, con una semántica operacional y un sistema de tipos completamente propios. No obstante, se reconoce que avanzar en esta dirección implica un estudio más riguroso y profundo de temas como semántica formal, diseño de lenguajes y construcción de compiladores, lo cual representa un desafío considerable. Aun así, lograr esta independencia sería altamente beneficioso, ya que permitiría una comprensión más completa del lenguaje y una evaluación más precisa de las decisiones de diseño tomadas.

A partir de este trabajo, se abren diversas líneas de mejora y trabajo futuro. Entre las extensiones más naturales al lenguaje se encuentran la incorporación de tipos algebraicos, clases de tipos y tipos existenciales, lo cual permitiría modelar estructuras de datos más

complejas y patrones de abstracción más ricos. Asimismo, la inclusión de mecanismos para el manejo de efectos, por ejemplo mediante mónadas, ampliaría el lenguaje hacia escenarios más cercanos a lenguajes funcionales de uso real.

Desde un punto de vista más teórico, una posible extensión relevante sería el estudio de sistemas de tipos más expresivos, como aquellos con polimorfismo de rango superior o tipos dependientes. Esto permitiría profundizar aún más en la relación entre la lógica de predicados y los lenguajes de programación, así como en la correspondencia de Curry–Howard en contextos más generales.

En conclusión, este proyecto demuestra que la inferencia de tipos no solo es una técnica esencial en la implementación de lenguajes de programación, sino también un puente conceptual profundo entre la lógica y la computación. El algoritmo W ejemplifica cómo es posible automatizar la construcción de pruebas dentro de un sistema formal, reafirmando la relevancia de los sistemas de tipos como una herramienta central en el diseño y análisis de lenguajes de programación modernos.

Referencias

[1] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[2] P. Hinman. *Fundamentals of Mathematical Logic*. A K Peters, 2005.

[3] R. Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences, 1978.

[4] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[5] Universidad Nacional Autónoma de México, Facultad de Ciencias. (s. f.). *Material de Lenguajes de Programación*. Recuperado de https://sites.google.com/ciencias.unam.mx/lengprog/material?authuser=0

[6] Type systems and logic. (s. f.). *Codewords Recurse*. Recuperado de: https://codewords.recurse.com/issues/one/type-systems-and-logic

[7] Wikipedia. *Hindley–Milner type system*. (s. f.). Recuperado de https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system

[8] Stimsina. (s. f.). *Implementing a Hindley-Milner type system — Part 1*. Recuperado de https://blog.stimsina.com/post/implementing-a-hindley-milner-type-system-part-1

[9] Bernsteinbear. (2024, 15 octubre).*Damas-Hindley-Milner inference two ways*. Recuperado de https://bernsteinbear.com/blog/type-inference/

[10] Bernstein, M. (2024, Octubre 15). *Damas-Hindley-Milner inference two ways*. Recuperado de https://bernsteinbear.com/blog/type-inference/

[11] Stimsina. (s. f.). *Implementing a Hindley-Milner Type System (Part 1)*. Recuperado de https://blog.stimsina.com/post/implementing-a-hindley-milner-type-system-part-1

[12] Recurse Center. (2020). *Type systems and logic*. Codewords. Recuperado de https://codewords.recurse.com/issues/one/type-systems-and-logic

[13] Wikipedia. (s. f.). *Hindley–Milner type system*. Recuperado de https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system

[14] Byorgey, B. (2021, Septiembre 8). *Implementing Hindley-Milner with the unification-fd library*. Recuperado de https://byorgey.wordpress.com/2021/09/08/implementing-hindley-milner-with-the-unification-fd-library/

[15] Microsoft Research. (2016). *HMF: Hindley–Milner Type System*. Recuperado dehttps://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/hmf.pdf

[16] Cornell University. (s. f.). *The Curry-Howard Correspondence*. CS 3110 — Data Structures and Functional Programming. Recuperado de https://courses.cs.cornell.edu/cs3110/2021sp/textbook/adv/curry-howard.html:contentReference[oaicite:0]{index=0}

[17] Academia-Lab. (s. f.). *Correspondencia Curry-Howard*. Recuperado de https://academia-lab.com/enciclopedia/correspondencia-curry-howard/

[18] Harvard School of Engineering and Applied Sciences. (2021, Marzo 18). *Curry-Howard Isomorphism; Existential types* (Lecture 16). Recuperado de https://courses.seas.harvard.edu/courses/cs152/2024sp/lectures/lec15-curryhoward.pdf:contentReference[oaicite:1]{index=1}