Evaluador de Expresiones Matemáticas con Diferenciación Automática

Informe Técnico Completo

Luciano David Duarte
lucianoduarte.wow@gmail.com

Diciembre 2025

Índice

1. Resumen Ejecutivo

Este proyecto implementa un evaluador de expresiones matemáticas completo en Haskell, con capacidad de calcular derivadas automáticamente mediante diferenciación automática usando números duales. El sistema incluye un parser robusto y un evaluador optimizado.

1.1. Características Principales

- Evaluación de expresiones aritméticas complejas
- Cálculo automático de derivadas mediante números duales
- Soporte para funciones trigonométricas, hiperbólicas, logarítmicas y exponenciales
- Parser con manejo de números negativos y notación matemática estándar
- Optimizaciones algebraicas automáticas
- Validación robusta de dominios matemáticos

2. Arquitectura del Sistema

2.1. Módulos Principales

El proyecto está organizado en los siguientes módulos:

1. **Expr.hs**: Define el AST (Abstract Syntax Tree)
2. **Parser.hs**: Implementa el analizador sintáctico
3. **Evaluator.hs**: Realiza la evaluación y diferenciación
4. **FileReader.hs**: Maneja la lectura de archivos
5. **Main.hs**: Punto de entrada del programa

2.2. Diagrama de Flujo

```
Input (String) → Parser → AST (Expr) → Optimizer
    → Evaluator → Result (Double + Derivative)
```

3. Representación de Expresiones (AST)

3.1. Definición del Tipo de Datos

```haskell
data Expr
  = Lit Double            -- Literales num ricos
  | Var String            -- Variables (x, pi, e)
  | Add Expr Expr         -- Suma
  | Sub Expr Expr         -- Resta
  | Mul Expr Expr         -- Multiplicaci n
  | Div Expr Expr         -- Divisi n
  | Pow Expr Expr         -- Potenciaci n
  | Sin Expr              -- Seno
  | Cos Expr              -- Coseno
  | Tan Expr              -- Tangente
  | Sinh Expr             -- Seno hiperb lico
  | Cosh Expr             -- Coseno hiperb lico
  | Tanh Expr             -- Tangente hiperb lica
  | Arsinh Expr           -- Arcoseno hiperb lico
  | Arcosh Expr           -- Arcocoseno hiperb lico
  | Artanh Expr           -- Arcotangente hiperb lica
  | Log Expr              -- Logaritmo natural
  | Exp Expr              -- Exponencial (e^x)
  | Sqrt Expr             -- Ra z cuadrada
  deriving (Eq, Show)
```

3.2. Ventajas del Enfoque

- Representación abstracta independiente de la sintaxis
- Facilita la composición de expresiones
- Permite optimizaciones mediante transformaciones de árboles
- Simplifica la implementación de evaluadores

4. Parser (Analizador Sintáctico)

4.1. Tecnología Utilizada

El parser está implementado usando la biblioteca **Parsec**, un parser combinator monádico que permite:

- Composición declarativa de parsers
- Manejo automático de backtracking
- Mensajes de error informativos
- Código limpio y mantenible

4.2. Precedencia de Operadores

La precedencia de operadores sigue las convenciones matemáticas estándar:

Operador	Precedencia	Asociatividad
Funciones unarias	4	N/A
Potencia (^)	3	Derecha
Multiplicación/División	2	Izquierda
Suma/Resta	1	Izquierda

Cuadro 1: Precedencia de operadores

5. Diferenciación Automática

5.1. Números Duales

La diferenciación automática se implementa mediante **números duales**, definidos como:

$$\mathbb{D} = \{a + b\varepsilon \mid a, b \in \mathbb{R}, \varepsilon^2 = 0\} \tag{1}$$

5.1.1. Implementación en Haskell

```haskell
data Dual = Dual
  { primal :: Double  -- Valor de la funci n
  , deriv  :: Double  -- Valor de la derivada
  }
```

5.2. Instancias de Type Classes

Para hacer los números duales operables, se implementan las siguientes instancias:

5.2.1. Num

```haskell
instance Num Dual where
  Dual p1 d1 + Dual p2 d2 = Dual (p1+p2) (d1+d2)
  Dual p1 d1 * Dual p2 d2 = Dual (p1*p2) (p1*d2 + p2*d1)
  -- Regla del producto
```

5.2.2. Fractional

```haskell
instance Fractional Dual where
  Dual p1 d1 / Dual p2 d2 =
    Dual (p1/p2) ((p2*d1 - p1*d2)/(p2^2))
  -- Regla del cociente
```

5.2.3. Floating

Implementa funciones trascendentes:

```haskell
instance Floating Dual where
  sin (Dual p d) = Dual (sin p) (cos p * d)
  exp (Dual p d) = Dual (exp p) (exp p * d)
  log (Dual p d) = Dual (log p) (d / p)
  -- ... m s funciones
```

5.3. Reglas de Derivación Implementadas

Función	Derivada	Implementación
$f + g$	$f' + g'$	Suma de derivadas
$f \cdot g$	$f'g + fg'$	Regla del producto
f/g	$(f'g - fg')/g^2$	Regla del cociente
f^g	$f^g(g' \ln f + gf'/f)$	Regla de potencias
$\sin(f)$	$\cos(f) \cdot f'$	Regla de la cadena
$\ln(f)$	f'/f	Derivada logarítmica
e^f	$e^f \cdot f'$	Derivada exponencial

Cuadro 2: Reglas de derivación

6. Evaluador

6.1. Tipo de Resultado

```haskell
type EvalResult = Either ErrorType Double

data ErrorType
  = DivideByZero
  | UndefinedVariable String
  | DomainError String
```

6.2. Validación de Dominios

El evaluador implementa validaciones exhaustivas:

- **División**: $x/0 \to$ Error
- **Logaritmo**: $\log(x)$ requiere $x > 0$
- **Raíz cuadrada**: $\sqrt{x}$ requiere $x \geq 0$
- **Potencias**:

- 0^0 y $0^{n<0} \rightarrow$ Error
- a^b con $a < 0$ y b no entero $\rightarrow$ Error

- **arcosh**: Requiere $x \geq 1$
- **artanh**: Requiere $|x| < 1$

7. Optimizaciones Algebraicas

7.1. Reglas Implementadas

```haskell
optimize :: Expr -> Expr
optimize expr = case expr of
  Add e (Lit 0) -> optimize e       -- x + 0 = x
  Add (Lit 0) e -> optimize e       -- 0 + x = x
  Mul e (Lit 1) -> optimize e       -- x * 1 = x
  Mul (Lit 1) e -> optimize e       -- 1 * x = x
  Mul _ (Lit 0) -> Lit 0            -- x * 0 = 0
  Mul (Lit 0) _ -> Lit 0            -- 0 * x = 0
  Sub e (Lit 0) -> optimize e       -- x - 0 = x
  Div e (Lit 1) -> optimize e       -- x / 1 = x
  Pow _ (Lit 0) -> Lit 1            -- x^0 = 1
  Pow e (Lit 1) -> optimize e       -- x^1 = x
  Pow (Lit 1) _ -> Lit 1            -- 1^x = 1
  -- Recursi n en subexpresiones
  Add e1 e2 -> Add (optimize e1) (optimize e2)
  -- ... m s casos
```

8. Procesamiento de Archivos

8.1. Formato de Entrada

```
expresión @ valor_x
-- Los comentarios comienzan con --
```

Ejemplos:

```
x^2 + 2*x + 1 @ 2
sin(x) * cos(x) @ 0.5
-- Este es un comentario
log(x) + exp(x) @ 1
```

8.2. Tipo de Resultado Completo

```haskell
data EvaluacionCompleta = EvaluacionCompleta
  { exprEval :: Expr
  , valorEval :: Double
  , resultadoValor :: Either ErrorType Double
  , resultadoDerivada :: Either ErrorType Dual
  }
```

9. Técnicas Avanzadas de Haskell

9.1. Uso de Mónadas

9.1.1. Either Monad para Manejo de Errores

```haskell
eval (Div e1 e2) x = do
  v1 <- eval e1 x
  v2 <- eval e2 x
  if v2 == 0
    then Left DivideByZero
    else return (v1 / v2)
```

9.1.2. Applicative Style

```haskell
eval (Add e1 e2) x = (+) <$> eval e1 x <*> eval e2 x
-- M s conciso que do-notation
```

9.2. Type Classes

Las instancias de `Num`, `Fractional` y `Floating` permiten usar `Dual` como un número normal, simplificando dramáticamente el código del evaluador.

9.3. Parser Combinators

Uso de Parsec para composición declarativa:

```haskell
parseExpr = spaces' >> parseAddSub
parseAddSub = chainl1 parseMulDiv (addOp <|> subOp)
parseMulDiv = chainl1 parsePow (mulOp <|> divOp)
```

10. Casos Límite y Manejo de Errores

10.1. Casos Numéricos Especiales

- **Infinito**: Evaluación de e^{1000} maneja overflow

- **NaN**: Detectado en potencias con bases negativas
- **Underflow**: Números muy pequeños tratados correctamente

10.2. Validaciones Exhaustivas

Cada operación valida su dominio antes de ejecutarse, produciendo mensajes de error descriptivos.

11. Referencias

1. Wadler, P. (1995). *Monads for functional programming*
2. Elliott, C. (2009). *Beautiful differentiation*
3. Leijen, D. & Meijer, E. (2001). *Parsec: Direct Style Monadic Parser Combinators*
4. Haskell Documentation: `https://www.haskell.org/documentation/`