CASkell: EDSL para el manejo simbólico de expresiones matemáticas

1. Instalación y uso del proyecto

Para correr el proyecto, es necesario tener instalado Stack, Make y ghc-8.10.7.

Una vez instalados, el comando:

```
make setup
```

Ejecutará todos los comandos para instalar las librerías necesarias.

El comando:

```
make all
```

Compila el proyecto, crea la documentación y corre todos los tests incluidos en la misma.

El comando:

```
stack exec -- ghci
```

Ejecuta `ghci` con todos los módulos del proyecto cargados. Los mismos se pueden importar usando la sentencia 'import'.

Para cargar un archivo que use las librerías del proyecto, usar:

```
stack exec -- ghci <dirección del archivo>
```

Para abrir la documentación del proyecto, usar:

```
make open-docs
```

2. Manejo básico de expresiones

Todo uso del EDSL necesita del tipo `Expr`, el cual se importa con la librería homónima.

```
import Expr
```

2.1 Crear expresiones matemáticas

Todas las `Expr` se construyen a partir de 2 elementos base, números y símbolos:

Números

Los números se pueden crear a partir de la función 'fromNumber' o haciendo un casting explícito al tipo `Expr`:

```
entero_dos = fromNumber 2
entero_tres = (3 :: Expr)
fraccion = fromNumber (21/19) -- fromNumber tiene mayor precedencia que '/' o cualquier operador matemático
```

¿Por qué es necesario el casting?

El casting es necesario debido a que Haskell convierte los números a `Integer` o `Double` de manera predeterminada en vez de al tipo `Expr`. Aunque a veces no es necesario si ya se está operando con `Expr`:

```
x = symbol "x" -- x:: Expr
u = x+2 -- El casting no es necesario
```

Una solución es mediante el uso de `default`:

```
default (Expr,Expr) -- Todas las expresiones numéricas serán casteadas automáticamente a Expr
```

Pero esto hará que cualquier expresión dentro del contexto de ejecución/módulo de trabajo sea casteada automáticamente a Expr, lo cual puede ser no deseable.

En general, si la expresión a utilizar solo números o funciones de números, entonces es necesario hacer un casting.

Números reales

También hay soporte para números reales, pero serán tratados como fracciones si la misma no es muy grande. Los números reales tienen una precisión fija y son sensibles a problemas de precisión.

```
(0.33 :: Expr) ==> 33/10 -- fracción pequeña
(0.3333333 :: Expr) ==> 0.3333333 -- la fracción 3333333/10000000 es muy grande
(0.11111222223333344444 :: Expr) ==> 0.11111222223333345 -- número redondeado
```

Símbolos

Creación

Los símbolos se pueden crear utilizando la función `symbol`, que toma una cadena de texto como argumento y devuelve una expresión simbólica. El resultado de `symbol` puede asignarse a un identificador y ser combinado con otras expresiones.

```
x = symbol "x"
y = symbol "y"
x+x+y -- 2*x+y
```

Los símbolos se identifican por el string pasado a symbol, **NO** por el identificador asignado:

```
x = symbol "qk"
y = symbol "qk"
x*y ==> qk^2 -- identificadores distintos, mismo símbolo
```

Suposiciones

Por defecto, se desconoce la naturaleza de los símbolos creados mediante 'symbol', solo se sabe que son números reales. Se pueden realizar suposiciones sobre los símbolos (ejemplo, es positivo o es entero) usando la función `assume`.

```
x = assume (symbol "x") ["even"]
y = assume (symbol "y") ["positive"]
n = assume (symbol "n") ["negative", "integer"]
```

Ciertas suposiciones pueden hacer que se ejecuten o no se ejecuten ciertas simplificaciones:

```
(2*x)**n ==> 2**n * x**n -- Distribución de potencias con exponentes enteros
(2*x)**y ==> (2*x)**y -- No hay distribución ya que 'y' no es entero

0 ** y ==> 0 -- 0^x = 0 para y > 0
0 ** n ==> Undefined: división por cero -- n < 0
0 ** x ==> 0^x -- No se sabe el signo de x, no se modifica la expresión
```

Las suposiciones no son retroactivas:

```
u = 0**y -- 0
y = assume (symbol "y") ["negative"] -- y ahora es negativo
0**y -- Undefined: división por cero
v = u -- v = 0
```

Las suposiciones afectan a todas las expresiones, no solo a los símbolos. Las operaciones involucradas y las suposiciones sobre los operadores involucrados afectan a la suposición de la expresión final:

```
-- x positivo, y negativo
x+2 -- positivo, x y 2 son positivos
x-y -- positivo, x y (-y) positivos
x+y -- se desconoce el signo, ya que x es positivo e y negativo
```

También es posible consultar si una expresión cumple una cierta suposición usando las funciones del tipo `is{suposición}`. Estas devolverán 3 posibles valores `T`(Verdadero), `F`(Falso) o `U`(Desconocido).

```
isPositive ((99::Expr)) ==> T
isEven ((pi::Expr)) ==> F
isNegative ((symbol "x")) ==> U -- Todos los símbolos se crean con suposiciones desconocidas
```

T, F y U son valores de verdad de lógica ternaria, por lo que los operadores booleanos definidos en Haskell no pueden usarse. El proyecto incluye operadores especiales para trabajar con estos valores:

```
-- And lógico
T &&& T = T
F &&& T = F
T &&& U = U

-- Or lógico
F ||| U = U
T ||| U = T  -- U puede ser True o False, para cualquier valor posible el or devuelve True
U ||| U = U
U ||| True = T -- Pueden combinarse con los booleanos de Haskell, pero el resultado siempre será un valor de lógica
ternaria.

-- Not lógico
not3 T = F
not3 F = T
not3 U = U
```

La siguiente tabla contiene un listado de suposiciones soportadas:

Suposición	Asumir sobre un símbolo	Preguntar suposición
positive	`assume _ ["positive"]`	isPositive
negative	`assume _ ["negative"]`	isNegative
zero	`assume _ ["zero"]`	isZero
even	`assume _ ["even"]`	isEven
integer	`assume _ ["integer"]`	isInteger
odd	`assume _ ["odd"]`	isOdd

El símbolo pi

`pi` es un símbolo con suposiciones predefinidas. Haskell por defecto intentará convertir 'pi' en un Double, por lo que puede ser necesario realizar un casting. Al ser un símbolo, es tenido en cuenta para ciertas simplificaciones.

```
(0**pi :: Expr) = 0 -- pi es positivo, no es entero por lo que no es ni par ni impar
(sin(pi) :: Expr) = 0
```

2.2 Combinando expresiones

`Expr` es una instancia de las clases `Num`, `Fractional` y `Floating`, por lo que soporta las expresiones matemáticas básicas y la aplicación de ciertas funciones.

```
x = symbol "x"
y = symbol "y"

-- Sumas y restas
x+2 -- 2 es automáticamente casteado a un Expr, por lo que no es necesario fromNumber
x-9

-- Productos y divisiones
2*x
x/y

-- Potencias
x^2 -- para exponentes positivos de tipo 'Integer'
x^^5 -- para exponentes 'Integer' de cualquier signo
x**y -- potencia entre 'Expr'

-- Aplicar funciones
sin(x)
tan(9)+y
exp((4::Expr)) -- casting necesario, sino evaluaría a un Double
```

```
log(x-12*pi)

-- También hay soporte para funciones anónimas, solo hay que pasar una lista con los argumentos
f = function "f"
f[x]+f[x] = 2*f(x)
```

Los elementos de tipo `Expr` cumplen todos los axiomas de cuerpo, excepto el de la propiedad distributiva:

```
x*2 == 2*x -- True
(x+y)+9 == x+(y+9) -- True
2*(x+y) == 2*x+2*y -- True, los números se distribuyen
(x+y)*z == x*z + y*z -- False, los términos no numéricos no se distribuyen

1/0 == log(-1)
=>(autosimplifican a)
Undefined: división por cero == Undefined: logaritmo de un número negativo
=>(la comparación evalúa a)
True -- Todas las expresiones indefinidas, son iguales entre sí
```

La siguiente tabla muestra las funciones que se pueden aplicar a las expresiones

Función en haskell	Función matemática
sin	seno
cos	coseno
tan	tangente
sec	secante
csc	cosecante
cot	cotangente
exp	exponencial
log	logaritmo natural
asin	arcoseno
acos	arcocoseno
atan	arcotangente
sinh	seno hiperbólico
cosh	coseno hiperbólico
tanh	tangente hiperbólica
asinh	arcoseno hiperbólico
acosh	arcocoseno hiperbólico
atanh	arcotangente hiperbólica
sqrt	raíz cuadrada

Autosimplificación

Las operaciones básicas ejecutan el proceso de **autosimplificación**, el cual realiza ciertas simplificaciones de manera automática

```
x+x ==> 2*x
x*x ==> x**2
(x**2)**3 ==> x**6
x + sin(pi/2) ==> x+1 -- sin(pi/2) = 1
```

La autosimplificación también se encarga de manejar expresiones que contengan términos indefinidos:

```
u = (1/0)::Expr -- Undefined: división por 0
v = symbol "v"
w = undefinedExpr "Undefined explícito"

sin(w)+1 ==> Undefined: Undefined explícito
u**u ==> Undefined: división por cero
u+v+w ==> Undefined: división por cero
w+v+u ==> Undefined: Undefined explícito -- Los indefinidos de más a la izquierda tienen prioridad
```

Las funciones encargadas de realizar el procedimiento de autosimplificación se encuentran en el archivo `src/Expr/Simplify.hs`. Estas funciones son utilizadas por los operadores matemáticos y no se usan con el tipo `Expr`.

Detección de expresiones indefinidas

La autosimplificación permite detectar ciertas expresiones prohibidas, por ejemplo, aquellas que incluyen una división por 0

```
1/(x-x) ==> Undefined: división por cero
1/(log(x/x)) ==> Undefined: división por cero
```

Límites de la autosimplificación

La **autosimplificación** no realiza todas las simplificaciones posibles, primero porque la lista de reglas de simplificación puede ser muy larga y segundo porque una autosimplificación con muchas reglas podría interferir con el funcionamiento de otras funciones (ejemplo, si la autosimplificación aplicara la propiedad distributiva siempre que pudiera, sería imposible crear una función para factorizar polinomios):

Esto hace que algunas expresiones queden sin simplificar:

```
sin(x)**2 + cos(x)**2 -- la expresión no cambia
1/(exp(2*x) - exp(x)**2) -- división por cero no reconocida
(x+1)**3 / (2*x**2+4*x+2) -- la expresión no cambia
```

Aun así, muchas de estas simplificaciones pueden ser aplicadas usando los módulos especializados para simplificación.

```
trigSimplify (sin(x)**2 + cos(x)**2) ==> 1
expExpand (1/(exp(2*x) - exp(x)**2)) ==> Undefined: división por 0
cancel ((x+1)**3 / (2*x**2+4*x+2)) ==> x/2 + 1/2
```

2.3 Pattern Matching sobre Expr

El tipo `Expr` soporta *Pattern Matching*, esto permite analizar una expresión en base a su estructura y modificarla de la manera que sea necesaria:

```
-- Extrae el primer operando de una expresión
primerOperando :: Expr -> Expr
primerOperando (Add (x :|| _)) = x
primerOperando (Mul (x :|| _)) = x
primerOperando (Pow x _) = x
primerOperando (Fun (x :| _)) = x
primerOperando x = x
```

Tipos TwoList y NonEmpty

Las expresiones en funciones se devuelven como un `NonEmpty Expr`, `NonEmpty a` representa una lista de elementos de tipo `a` que garantiza la existencia de al menos un elemento.

```
data NonEmpty a = a :| [a]
```

Las expresiones en sumas y productos se devuelven en una `TwoList`, una `TwoList` es análoga a una `NonEmpty` pero garantiza la existencia de al menos 2 elementos.

```
data TwoList a = a :|| NonEmpty a
```

El tipo `TwoList` se define en el archivo `src/Data/TwoList.hs`.

Para el tipo `NonEmpty`: consultar la documentación en Hackage

Patrones derivados

Los patrones básicos son `Number`, `Symbol`, `Add`, `Mul`, `Pow`, `Fun` y `Undefined`. Existen patrones adicionales que se derivan a partir de los básicos.

Para una implementación de los patrones, ver el archivo `Expr/Structure.hs`.

Listado de patrones implementados

Patrón	Descripción
`Number x`	Matchea cualquier número `x`.
`Symbol s`	Matchea cualquier símbolo de nombre `s`.

Patrón	Descripción
`Add (x :|| y :| xs )`	Matchea una suma de dos o más expresiones. `x` es el primer elemento, `y` el segundo y `xs` es una lista con el resto de los argumentos.
`Mul (x :|| y :| xs )`	Matchea un producto de dos o más expresiones. `x` es el primer elemento, `y` el segundo y `xs` es una lista con el resto de los argumentos.
`Pow x y`	Matchea una potencia de base `x` y un exponente `y`.
`Fun f (x :| xs)`	Matchea una función aplicada a una lista de uno o más argumentos. `x` es el primer argumento y `xs` una lista con los argumentos restantes
`Undefined e`	Matchea cualquier expresión indefinida, donde `e` es el error correspondiente
`Pi`	Matchea el símbolo pi
`Neg x`	Matchea una expresión `x` multiplicada por un número negativo.
`MonomialTerm u n`	Matchea expresiones de la forma `u**n`, donde `u` es una expresión cualquiera y `n` es un número natural mayor a 1
`Sqrt u`	Matchea una expresión `u` elevada a `1/2`
`Div n d`	Matchea una división de numerador `n` y denominador `d`.
`Exp x`	Matchea una expresión exponencial con base `e` y exponente `x`.
`Log x`	Matchea una expresión logarítmica con base `e` y argumento `x`.
`Sin x`	Matchea la función seno aplicada a `x`.
`Cos x`	Matchea la función coseno aplicada a `x`.
`Tan x`	Matchea la función tangente aplicada a `x`.
`Asin x`	Matchea la función arco seno aplicada a `x`.
`Acos x`	Matchea la función arco coseno aplicada a `x`.
`Atan x`	Matchea la función arco tangente aplicada a `x`.
`Derivative u x`	Matchea la derivada sin evaluar de `u` con respecto a `x`.
`Integral u x`	Matchea la integral indefinida sin evaluar de `u` con respecto a `x`.
`DefiniteIntegral u x a b`	Matchea la integral definida sin evaluar de `u` con respecto a `x` en el intervalo `[a, b]`.

Orden de las expresiones en los patrones

En patrones que representan operaciones conmutativas como `Add` y `Mul` las expresiones se colocan en un orden específico. Esto facilita cosas como la comparación de expresiones, ya que `x+1` y `1+x` internamente siempre tendrán el mismo orden. Sin embargo, esto puede resultar una complicación a la hora de hacer pattern matching.

```
match1 (Add ((Pow (Symbol x) 2) :|| 1 :| [])) = True
match1 _ = False

match2 (Add (1 :|| (Pow (Symbol x) 2) :| [])) = True
match2 _ = False

match1 (x**2+1) = False
match1 (1+x**2) = False

match2 (x**2+1) = True
match2 (1+x**2) = True

-- nota: match1 y match2 podrían reemplazarse por la función (==((symbol "x")**2+1)), si es que 'x' no requiere
suposiciones
```

El orden de las expresiones es determinado por la instancia de `Ord` del tipo `PExpr`, ubicado en `src/Expr/PExpr.hs`.

3. Módulos especiales

Los módulos especiales se construyen a partir del tipo `Expr` y permiten realizar las siguientes 6 funcionalidades:

- Evaluación numérica
- Simplificación avanzada
- Derivación
- Integración
- Parseo de expresiones
- PrettyPrinting de expresiones

3.1 Evaluación numérica

Para evaluar numéricamente una expresión, hay que importar el módulo `Evaluate.Numeric`

```
import Evaluate.Numeric
```

Las expresiones podrán evaluarse usando la función `eval`:

```
eval [] (2*sin(pi/4)) = 1.4142135623730951 -- sqrt 2
eval [(x,2.2)] (7.8+x) = 10-- Puedes reemplazar los símbolos por valores numéricos
```

3.2 Simplificación avanzada

Los módulos para simplificación permiten realizar algunas simplificaciones que la autosimplificación por sí sola no puede realizar. Estos módulos pueden operar con:

- Expresiones algebraicas (polinomios y expresiones racionales)
- Expresiones trigonométricas
- Expresiones con exponenciales
- Expresiones con logaritmos

A su vez, todos los módulos (salvo los de expresiones algebraicas) contienen 3 funciones para realizar simplificaciones, una función de expansión, una de contracción y una de simplificación. El funcionamiento exacto varía de módulo en módulo pero por lo general operan de la siguiente forma:

- Función de expansión: Intenta hacer las expresiones más grandes, puede llegar a formar expresiones más pequeñas gracias a la autosimplificación:

  ```
  -- Expansión algebraica
  expand((x + 1)*(x - 2) - (x - 1)*x)
  =>(expande a)
  x**2 - 2*x + x - 2 - x**2 + x
  =>(autosimplifica a)
  -2

  resultado final: -2
  ```

- Función de contracción: Intenta hacer las expresiones más pequeñas:

  ```
  trigContract (2*sin(x)*cos(x)) = sin(2*x) -- Contracción trigonométrica
  expContract (exp(2) * exp(5)) = exp(5) -- Contracción de exponenciales
  ```

- Función de simplificación: Racionaliza la expresión, intenta contraer el numerador y el denominador y cancela términos usando la autosimplificación:

  ```
  cancel ((x+y)*(x-y)/(x**3-x*y**2)) = 1/x -- Simplificación algebraica
  ```

En general, la expansión no es la inversa ni de la contracción ni de la simplificación, debido al proceso de autosimplificación:

```
expExpand (expContract (exp(x)**2)) = exp(x)**2
-- La expansión anuló la contracción

expExpand (expContract (exp(x)**2- exp(2*x)))
=>(contrae a)
expExpand (exp(2*x) - exp(2*x))
=>(autosimplifica a)
expExpand 0
=>(expande a)
0
-- La expansión no anuló la contracción
```

Tabla de funciones de simplificación avanzada

Área de simplificación	Función de expansión	Función de contracción	Función de simplificación	Módulo/s
Algebraica	`expand`	No existe	`cancel`	`Simplification.Algebraic` y `Simplification.Rationalize`
Trigonométrica	`trigExpand`	`trigContract`	`trigSimplify`	`Simplification.Trigonometric`
Exponencial	`expExpand`	`expContract`	`expSimplify`	`Simplification.Exponential`
Logarítmica	`logExpand`	`logContract`	`logSimplify`	`Simplification.Logarithm`

3.3 Derivación

Para derivar expresiones, importar el módulo `Calculus.Derivate`:

```
import Calculus.Derivate
```

Y luego usar la función `derivate`:

```
derivate (x**2) x = 2*x
derivate (exp(x)) x = exp(x)
f = function "f"
g = function "g"
derivate (f[x]) = Derivate(f(x), x) -- Derivada desconocida, devuelvo una derivada sin evaluar
derivate (f[g[x]]) x = Derivate(f(x),g(x)) * Derivate(g(x), x) -- Derivada sin evaluar aplicando la regla de la cadena
```

3.4 Integración

Para integrar expresiones, importar el módulo `Calculus.Integrate`:

```
import Calculus.Integrate
```

Y luego usar la función `integrate`:

```
integrate (cos x) x = sin(x) -- Notar que no se agrega la constante de integración
integrate (exp(x)) = exp(x)
integrate (2*sin(x)*cos(x)) = -cos(x)^2
```

Si no se puede encontrar la integral de la función (ya sea porque no es una integral elemental, el algoritmo de integración no puede encontrarla o la misma se desconoce), se devuelve una integral desconocida:

```
integrate (exp(-x**2)) x = Integral(e^(-x^2),x) -- Integral no elemental
integrate (1/(x**2+1)) x = Integral(1/(x^2+1),x) -- Integral elemental, pero no obtenida por el algoritmo
f = function "f"
integrate (f[x]) x = Integral(f(x), x) -- Integral desconocida
```

3.5 Parseo de expresiones

El módulo `Expr` viene incluida con la función `parseExpr` que convierte una cadena de texto en una expresión

```
parseExpr "x" -- Devuelve el símbolo x
parseExpr "x + x + sin(pi / 2)" -- Devuelve 2*x+1, la expresión se evalúa usando autosimplificación
parseExpr "f(x) + g(y)" -- Devuelve f(x)+g(y), puede detectar funciones anónimas

a = symbol "a"
u = parseExpr "b+c"
a+u    -- Devuelve a+u, las expresiones parseadas pueden combinarse con expresiones no parseadas
```

En caso de un error de parseo, se devuelve una expresión indefinida.

```
u = symbol "2++x" -- Undefined: Error de parseo
```

El parser se construye a partir de una gramática de Happy, el archivo de la gramática se encuentra en el módulo `Expr/Parser.y`.

3.6 PrettyPrinting

El prettyprinting de expresiones se realiza en el archivo `Expr/PrettyPrint.hs`, utilizando la librería `PrettyPrinter`:

```
y*2*x + y**2 + x**2 -- se muestra como x^2 + 2*x*y + y^2, los términos se reorganizan
exp(x)+exp(y) -- se muestra como e^x+e^y
2 * x**(-1) * y**(-1) -- se muestra como 2/(x*y)
```

4. Organización de los archivos

La estructura del proyecto es la siguiente:

```
.
|-- imgs               -- Imagenes usadas en el informe
|-- src
|   |-- Calculus
|   |   |-- Derivate.hs -- Derivación de expresiones
|   |   |-- Integrate.hs -- Integración de expresiones
|   |   |-- Utils.hs     -- Funciones de utilidad usada por los módulos en la carpeta Calculus
|   |-- Classes
|   |   |-- Assumptions.hs -- Funciones para suposiciones
|   |   |-- EvalResult.hs  -- Mónada EvalResult
|   |-- Data
|   |   |-- Number.hs  -- Tipo 'Number', utilizado por las expresiones cuando operan con números puros
|   |   |-- TriBool.hs -- Manejo de lógica ternaria
|   |   |-- TwoList.hs -- Tipo 'TwoList', que representa listas con 2 o más elementos
|   |-- Evaluate
|   |   |-- Numeric.hs -- Evaluación numérica
|   |-- Expr
|   |   |-- Expr.hs     -- Junta todos los módulos y los exporta como uno
|   |   |-- ExprType.hs -- Definición del tipo Expr
|   |   |-- Parser.y    -- Parser de expresiones
|   |   |-- PExpr.hs    -- Tipo 'PExpr', para manejar árboles de expansiones
|   |   |-- PolyTools.hs -- Funciones para trabajar con expresiones polinómicas
|   |   |-- PrettyPrint.hs -- Prettyprinting de expresiones
|   |   |-- Simplify.hs -- Autosimplificación
|   |   |-- Structure.hs -- Pattern matching de expresiones
|   |-- Simplification
|   |   |-- Algebraic.hs      -- Expansión algebraica
|   |   |-- Exponential.hs    -- Simplificación de exponenciales
|   |   |-- Logarithm.hs      -- Simplificación de logaritmos
|   |   |-- Rationalize.hs    -- Simplificación algebraica
|   |   |-- Trigonometric.hs  -- Simplificación de funciones trigonométricas
|-- CASkell.cabal
|-- README.md
|-- .gitignore
|-- stack-yaml
|-- stack.yaml.lock
|-- makefile
|-- README.md
|-- Informe.pdf
```

5. Decisiones de diseño

5.1 EDSL por sobre DSL

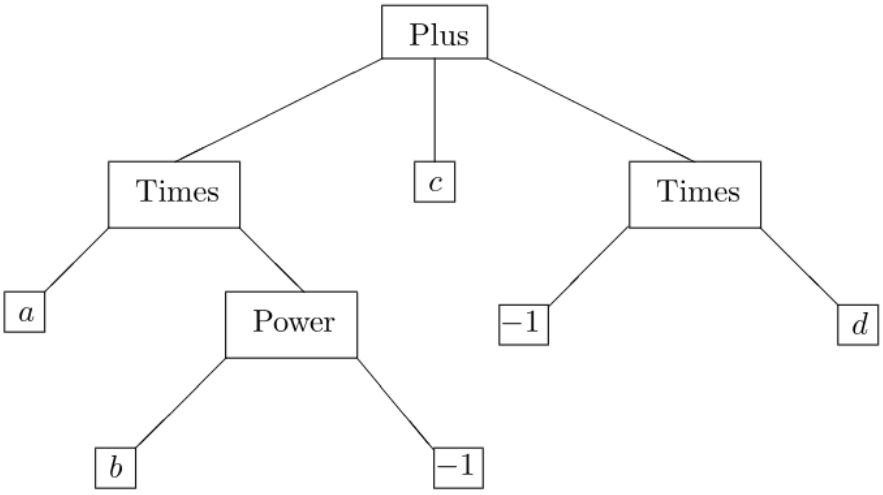
La decisión inicial fue si crear un DSL (Domain Specific Language) o un EDSL (Embedded Domain Specific Language). Un DSL es un lenguaje de programación especializado en un dominio particular, mientras que un EDSL es un DSL que se construye dentro de un lenguaje de programación general, aprovechando su sintaxis y funcionalidades.

Hay 2 razones por las que terminé implementando un EDSL:

1. **Pattern Matching**: El pattern matching a la hora de trabajar con expresiones matemáticas es fundamental y es utilizado en la mayoría de las funciones del proyecto. El soporte de Haskell para realizar Pattern Matching, junto con las extensiones `PatternSynonyms` y `ViewPatterns`, resultó fundamental para simplificar y hacer más legible el código. En un DSL habría que implementar alguna forma de Pattern Matching desde 0, la cual sería potencialmente inferior a la de Haskell.
2. **Reutilización de la infraestructura de Haskell**: Al construir un EDSL dentro de Haskell, se puede aprovechar toda la infraestructura existente del lenguaje, incluyendo su sistema de tipos, funciones de alto orden, y librerías estándar. Esto reduce el esfuerzo de implementación y permite utilizar módulos especializados como `Happy` o `PrettyPrinter`.

5.2 Representación de expresiones

Internamente, las expresiones se representan como árboles de expresiones



Ejemplo de representación de $a/b+c-d$, sacada de *Computer algebra and symbolic computation: Elementary Algorithms*

Estos árboles de expresiones se representan en código mediante el tipo `PExpr`, definido en `src/Expr/PExpr.hs`

```
data PExpr = Number Number
               | SymbolWithAssumptions String AssumptionsEnviroment
               | Mul [PExpr]
               | Add [PExpr]
               | Pow PExpr PExpr
               | Fun String [PExpr]
```

La expresión $a/b+c-d$ en `PExpr` sería similar a la siguiente

```
a/b+c-d = Add [
        Mul [
            Symbol "a",
            Pow (Symbol "b") (Number (-1))
        ],
        Symbol "c",
        Symbol "d"
    ]
```

Las funciones de autosimplificación operan con tipos `PExpr` y evalúan a una `Expr`:

```
simplifyPow :: PExpr -> PExpr -> Expr -- Autosimplificación de potencias
```

5.3 Manejo de errores con mónadas

El tipo `PExpr` no realiza el manejo de expresiones indefinidas(`Undefined: ***`), sino que el mismo se realiza mediante el uso de mónadas.

En particular se utiliza la mónada `EvalResult`, la cual es una mónada de error encapsulada.

```
newtype EvalResult a = EvalResult { runEvalResult :: Either Error a }
```

El tipo `Expr` es simplemente una `PExpr` encapsulada en una mónada `EvalResult`.

```
type Expr = EvalResult PExpr
```

Los operadores matemáticos se construyen utilizando la notación `do` y las funciones de autosimplificación:

```
(**): Expr -> Expr -> Expr -- Potencia de expresiones
a ** b = do
            a' <- a -- PExpr
            b' <- b -- PExpr
            simplifyPow a b
```

Esto permite a los operadores matemáticos detectar cuando trabajan con operadores indefinidos y propagar el error hacia futuros cálculos.

`EvalResult` también soporta la operación de choice(`<|>`), lo cual es útil para cambiar el resultado de una operación con respuesta indefinida.

```
integrate :: Expr -> Expr -> Expr
integrate f x = integralTable f x
                    <|> -- si integralTable devuelve undefined, evaluar la siguiente función
                linearProperties f x
                    <|> -- si linearProperties devuelve undefined, evaluar la siguiente función
                substitutionMethod f x
                    <|> -- y así...
                let g = Algebraic.expand f
                in if f /= g
                    then integrate g x
                    else makeUnevaluatedIntegral f x
```

5.4 Suposiciones con lógica ternaria

La autosimplificación necesita poder realizar suposiciones sobre las expresiones para hacer o no hacer simplificaciones. Estas suposiciones pueden ser verdaderas o falsas, pero no siempre se puede asignar alguno de estos dos valores.

Por ejemplo, en la expresión `0^x`. `x` es un símbolo con valor desconocido, por lo que no es posible asignar una suposición de positivo o negativo a `x`. Es decir los valores de verdad de `x>=0` o `x<=0` son desconocidos.

La lógica ternaria aborda este problema, introduciendo un tercer valor de verdad a las expresiones booleanas, **Desconocido(U)**.

Esto permite que las suposiciones como `¿x es positivo?` tengan 3 posibles respuestas Verdadero(`T`), Falso(`F`) o Desconocido(`U`).

Las suposiciones se guardan directamente en el tipo `PExpr`, específicamente en las hojas de tipo `Symbol`. Para determinar el valor de verdad de una suposición sobre una expresión, se analiza el árbol de expresiones y se construye la suposición en base a los operandos involucrados (Ejemplo, si todos los operandos de una suma son positivos, la suma debe ser positiva).

Las operaciones para trabajar con valores de lógica ternaria se encuentran en el archivo `Data/TriBool.hs`.

Más información sobre lógica ternaria: https://en.wikipedia.org/wiki/Three-valued_logic

6. Testing y documentación

Las funciones dentro del código cuentan con comentarios explicando la funcionalidad y el propósito. Junto con los comentarios se encuentran ejemplos de cómo se comporta la función.

```
{-|
      Utiliza las reglas de derivación para calcular la derivada de una expresión con respecto a una variable dada.

      Las reglas de derivación utilizadas son:

      * Derivada de una constante: \(\dfrac{d}{dx}c = 0\)
      * Regla de la suma: \(\dfrac{d}{dx}\sum u_i = \sum \dfrac{du_i}{dx}\)

      ...

      === Ejemplos:
      >>> derivate (x**2 + 2*x + 1) x
      2*x+2
      >>> derivate (sin x) x
      Cos(x)

      ...
-}
derivate u x = ...
```

Estos comentarios pueden usarse para generar documentación del proyecto con `haddock`. Usando `make docs` se puede generar la documentación del código sin abrirla y usando `make open-docs` se puede generar la documentación y abrirla en el navegador web.

```
derive :: Expr -> Expr -> Expr                                                    #
```

Utiliza las reglas de derivación para calcular la derivada de una expresión con respecto a una variable dada.

Las reglas de derivación utilizadas son:

- Derivada de una constante: $\frac{d}{dx}c = 0$
- Regla de la suma: $\frac{d}{dx}\sum u_i = \sum \frac{du_i}{dx}$
- Regla del producto: $\frac{d}{dx}(u \cdot v) = u \cdot \frac{dv}{dx} + v \cdot \frac{du}{dx}$, mas especificamente, la forma general de dicha regla, $\frac{d}{dx}\left(\prod u_i\right) = \left(\prod u_i\right) \cdot \sum \frac{du_i}{dx} \cdot \frac{1}{u_i}$
- Regla de la potencia: $\frac{d}{dx}v^w = wv^{w-1}\frac{dv}{dx} + \frac{dw}{dx}v^w \log v$
- Regla de la cadena: $\frac{d}{dx}f(g(x)) = f'(g(x)) \cdot g'(x)$

Ademas se utliza la funcion `derivateTable` para calcular la derivada de funciones matemáticas comunes.

Si al aplicar las reglas de derivación no se puede calcular la derivada, se devuelve una derivada sin evaluar.

Ejemplos

```
>>> derivate (x**2 + 2*x + 1) x
2*x+2
>>> derivate (sin x) x
cos(x)
>>> derivate (exp(x**2)) x
2*e^(x^2)*x
>>> derivate (x*log(x)-x) x
log(x)
>>> derivate (f[x]) x
Derivate(f(x),x)
>>> derivate (f[x]*exp(x)) x
Derivate(f(x),x)*e^x+e^x*f(x)
>>> derivate (f[g[x]]) x
Derivate(f(g(x)),g(x))*Derivate(g(x),x)
```

La página de documentación para el ejemplo anterior

Además, los ejemplos en la documentación se pueden usar para testear el funcionamiento correcto del proyecto. El comando `make test` lee los ejemplos de la documentación y los ejecuta para ver si obtienen el resultado esperado.

7. Bibliografía, librerías externas y referencias

Bibliografía

- *Cohen, J. (2003). Computer Algebra and Symbolic Computation: Elementary Algorithms. A K Peters/CRC Press.*
- *Cohen, J. (2003). Computer Algebra and Symbolic Computation: Mathematical Methods. A K Peters/CRC Press.S. R., & Labahn, G. (1992).*
- *Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy: symbolic computing in Python. PeerJ Computer Science 3:e103 https://doi.org/10.7717/peerj-cs.103*

Librerías externas

- `base`: Provee las funcionalidades básicas del lenguaje Haskell.
- `pretty`: Provee herramientas para pretty-printing.
- `exact-combinatorics`: Provee funciones para hacer cálculos combinatorios.
- `matrix`: Provee herramientas para trabajar con matrices.
- `happy`: Provee un generador de analizadores sintácticos para Haskell, utilizado para construir el parser de expresiones.
- `haddock`: Generación de la documentación del proyecto.
- `doctest`: Realiza el testeo de las funciones a partir de los casos de prueba de la documentación.

Referencias

- Happy User Guide: https://www.haskell.org/happy/doc/html/
- Documentación de PrettyPrinter: https://hackage.haskell.org/package/pretty
- Wikipedia, Three-valued logic: https://en.wikipedia.org/wiki/Three-valued_logic
- Sympy, un CAS implementado en Python. Usado como inspiración para el sistema de suposiciones además de una referencia de cómo deberían comportarse las funciones: https://www.sympy.org/en/index.html